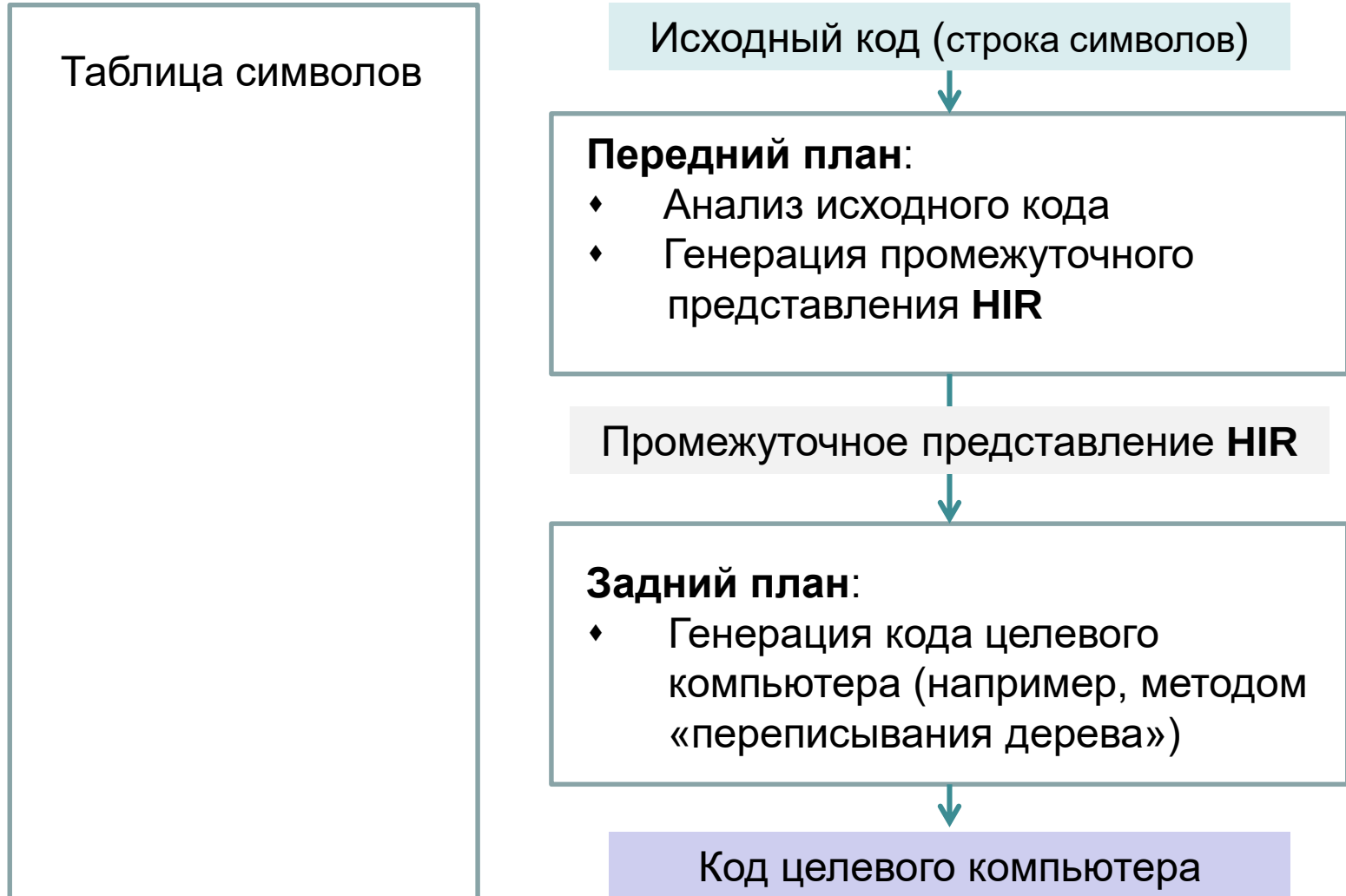


1. Введение. Неоптимизирующий компилятор

1.1 Компиляторы

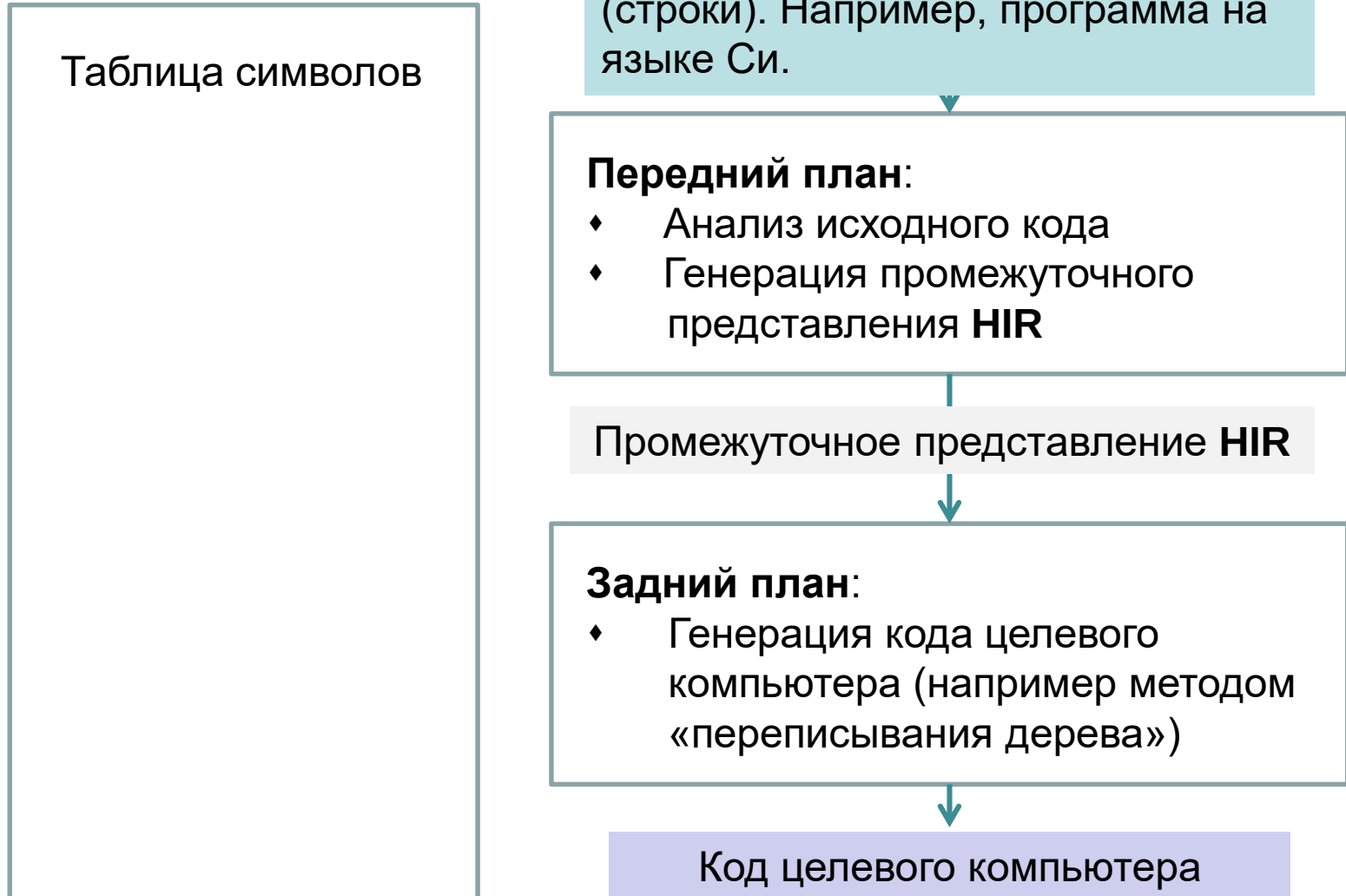
1.1.1. Неоптимизирующий компилятор



Структура неоптимизирующего компилятора

1.1 Компиляторы

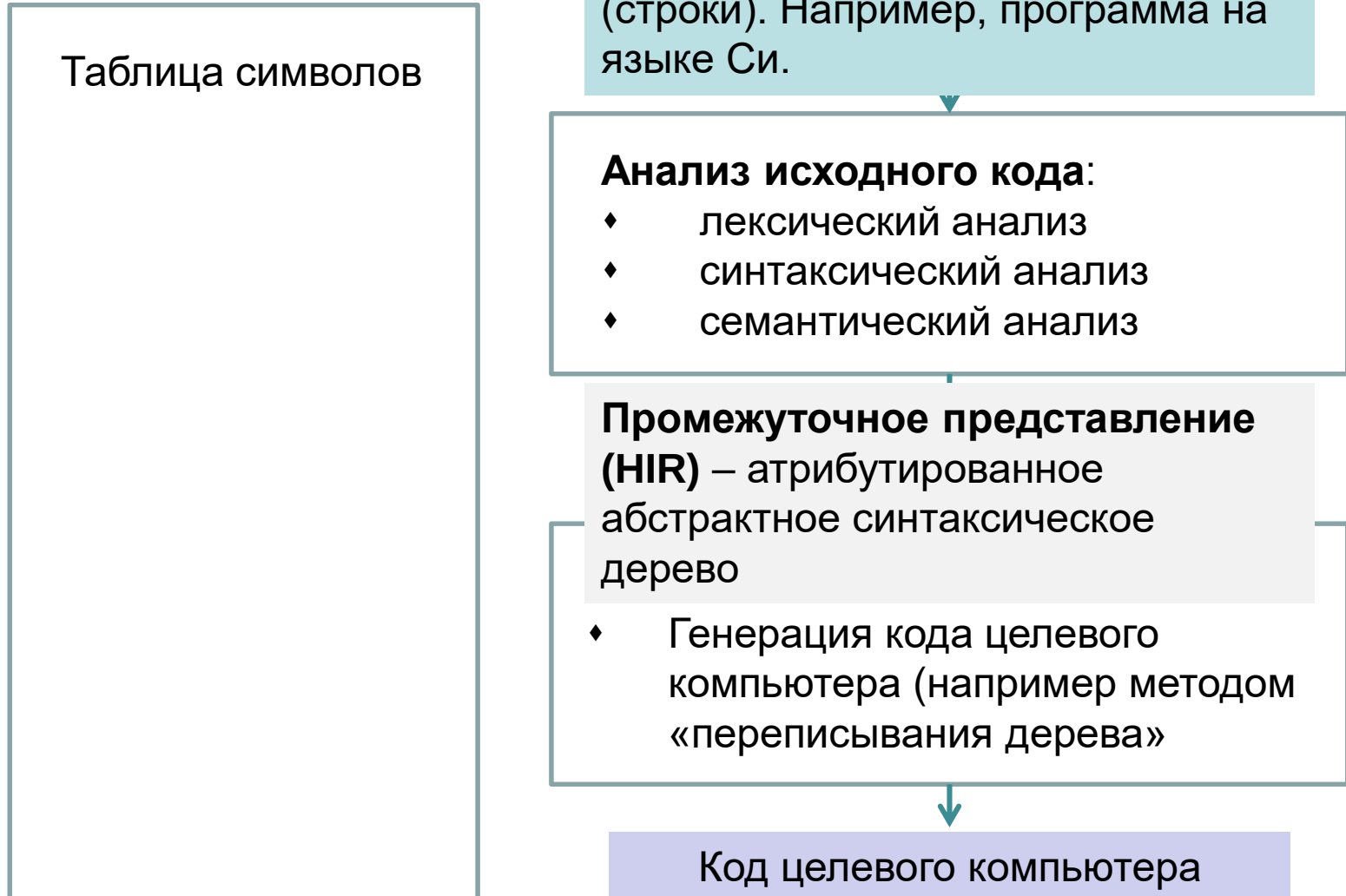
1.1.1. Неоптимизирующий компилятор



Структура неоптимизирующего компилятора

1.1 Компиляторы

1.1.1. Неоптимизирующий компилятор



Структура неоптимизирующего компилятора

1.1 Компиляторы

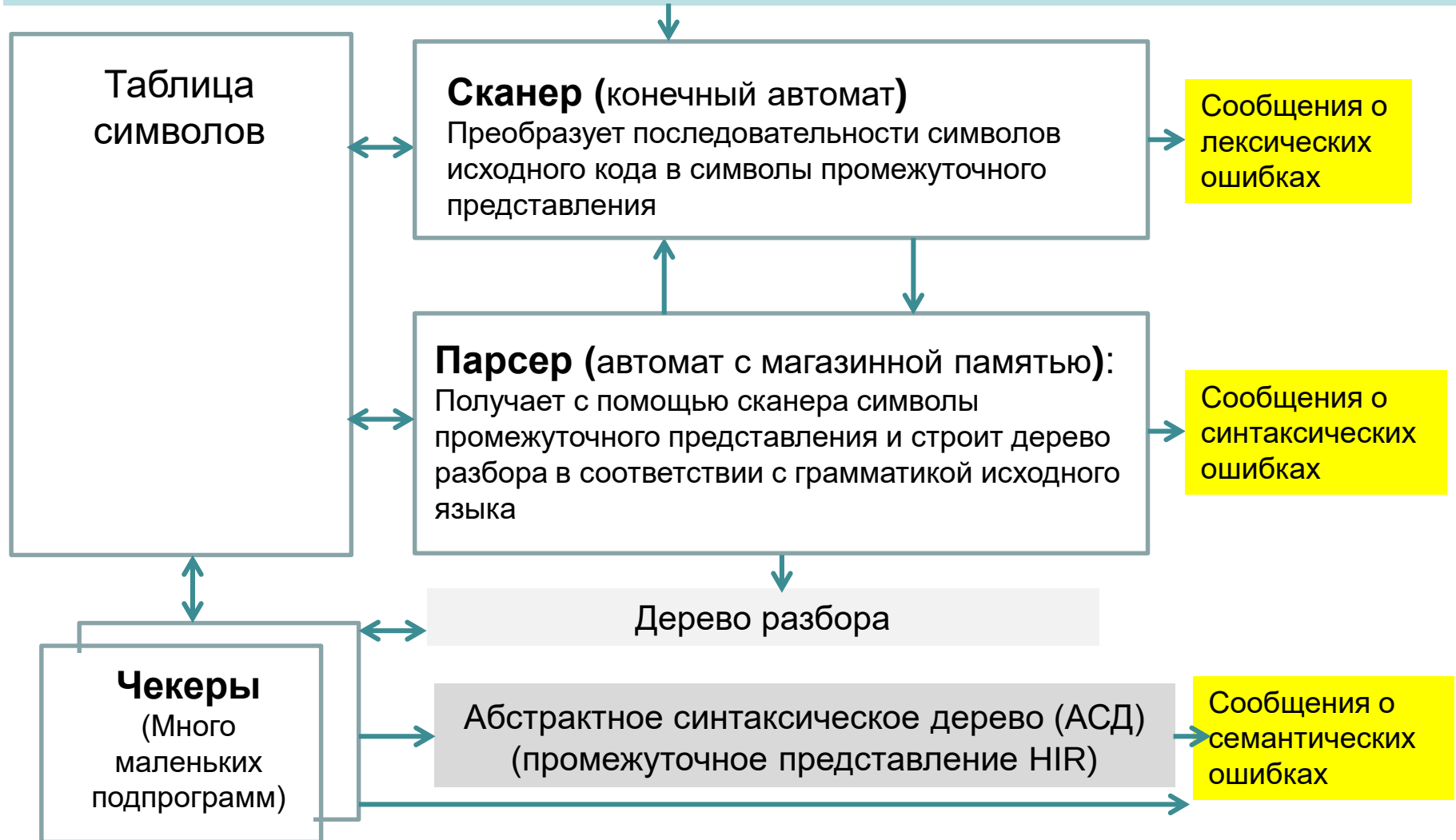
1.1.2. Синтаксически управляемая трансляция



1.1 Компиляторы

1.1.2. Синтаксически управляемая трансляция

Исходный код – текст модуля компиляции в виде строки символов

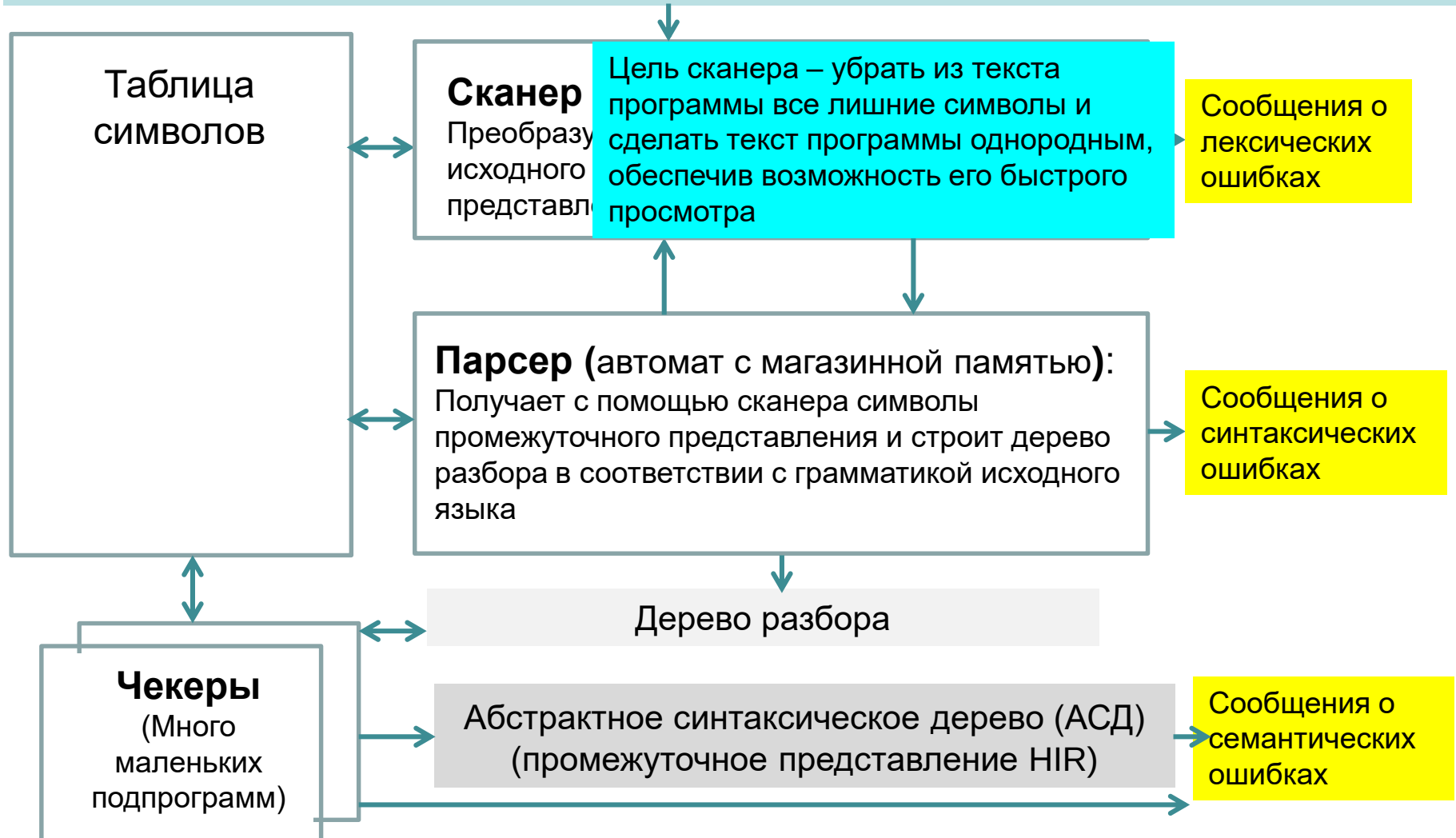


Каждый чекер вычисляет атрибут узлов дерева разбора и проверяет соответствуют ли атрибуты родительских и дочерних узлов; после чего либо строит соответствующую часть АСД, либо выдает сообщение об ошибке

1.1 Компиляторы

1.1.2. Синтаксически управляемая трансляция

Исходный код – текст модуля компиляции в виде строки символов

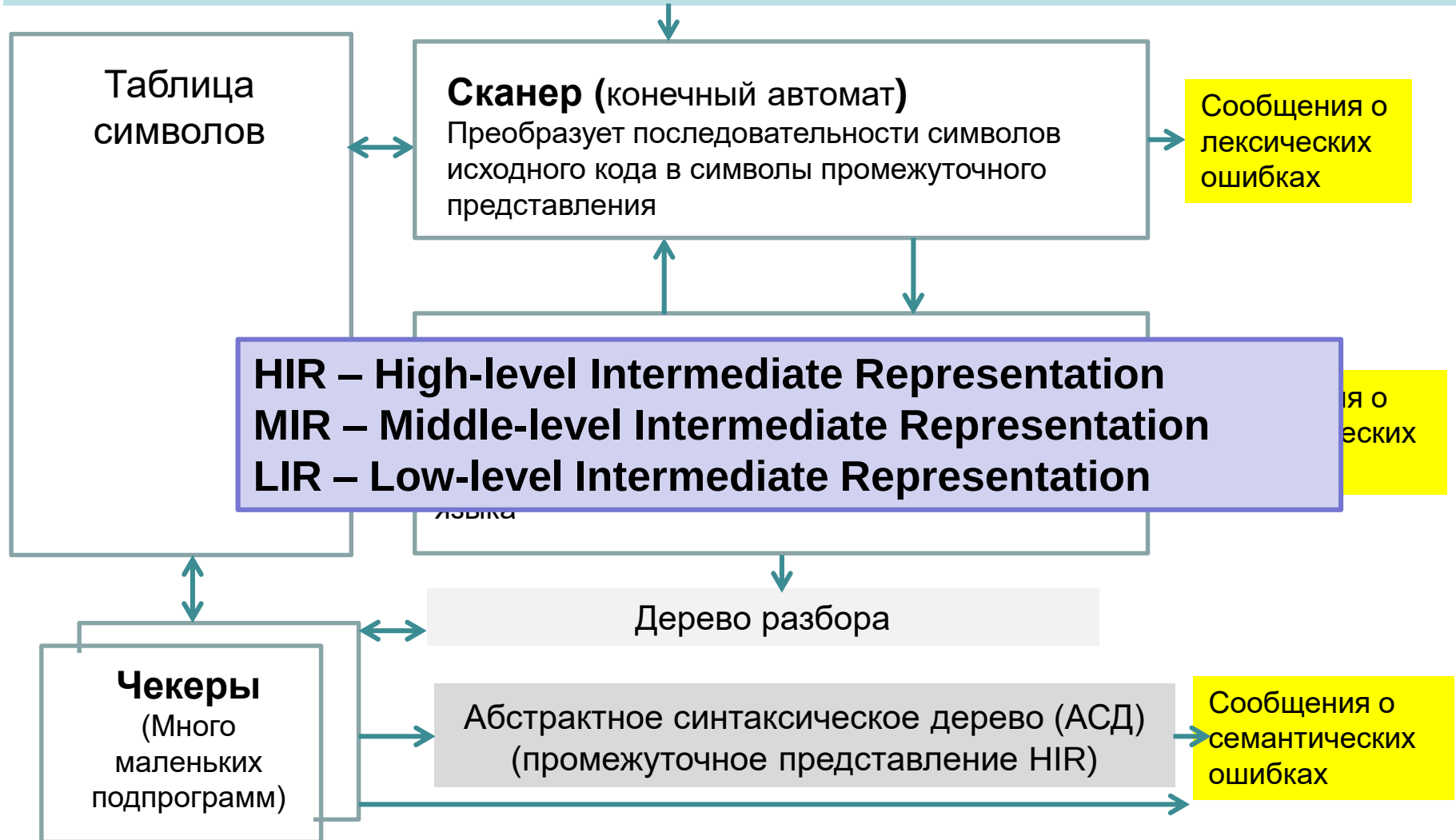


Каждый чекер вычисляет атрибут узлов дерева разбора и проверяет соответствуют ли атрибуты родительских и дочерних узлов; после чего либо строит соответствующую часть АСД, либо выдает сообщение об ошибке (семантической)

1.1 Компиляторы

1.1.2. Синтаксически управляемая трансляция

Исходный код – текст модуля компиляции в виде строки символов



Каждый чекер вычисляет атрибут узлов дерева разбора и проверяет соответствуют ли атрибуты родительских и дочерних узлов; после чего либо строит соответствующую часть АСД, либо выдает сообщение об ошибке

1.1 Компиляторы

1.1.2. Синтаксически управляемая трансляция

- ◇ По АСД (HIR) можно восстановить исходную программу, так как для каждого оператора исходной программы существует узел АСД. Промежуточные представления MIR и LIR, которые будут использоваться в данном курсе таким свойством не обладают даже в том случае, когда оптимизация была отключена.
- ◇ Передний и задний планы компилятора изучаются в курсе «Конструирование компиляторов», а также в таких курсах как «Теория алгоритмических языков», «Теория автоматов» и др. В данном курсе считается, что студенты все это знают. Для тех, кто забыл могу рекомендовать учебное пособие

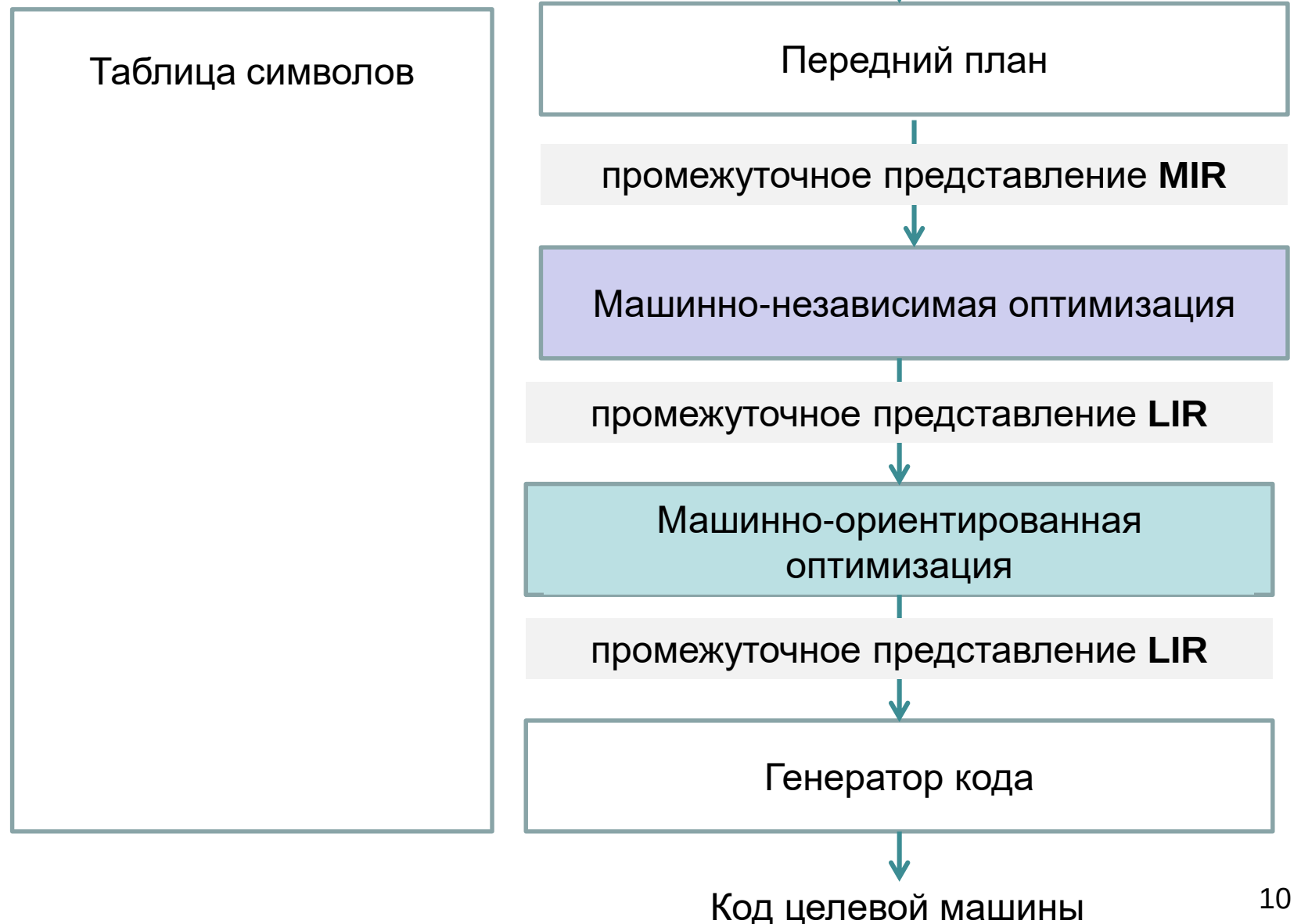
В. А. Серебряков, М. П. Галочкин. Теория и реализация языков программирования, учебное пособие, 2-е изд., доп. и испр.
Москва : МЗ Пресс, 2006. - 350 с.

В Интернете оно есть в формате pdf.

http://trpl7.ru/t-books/_TRYAPBOOK_pdf.pdf

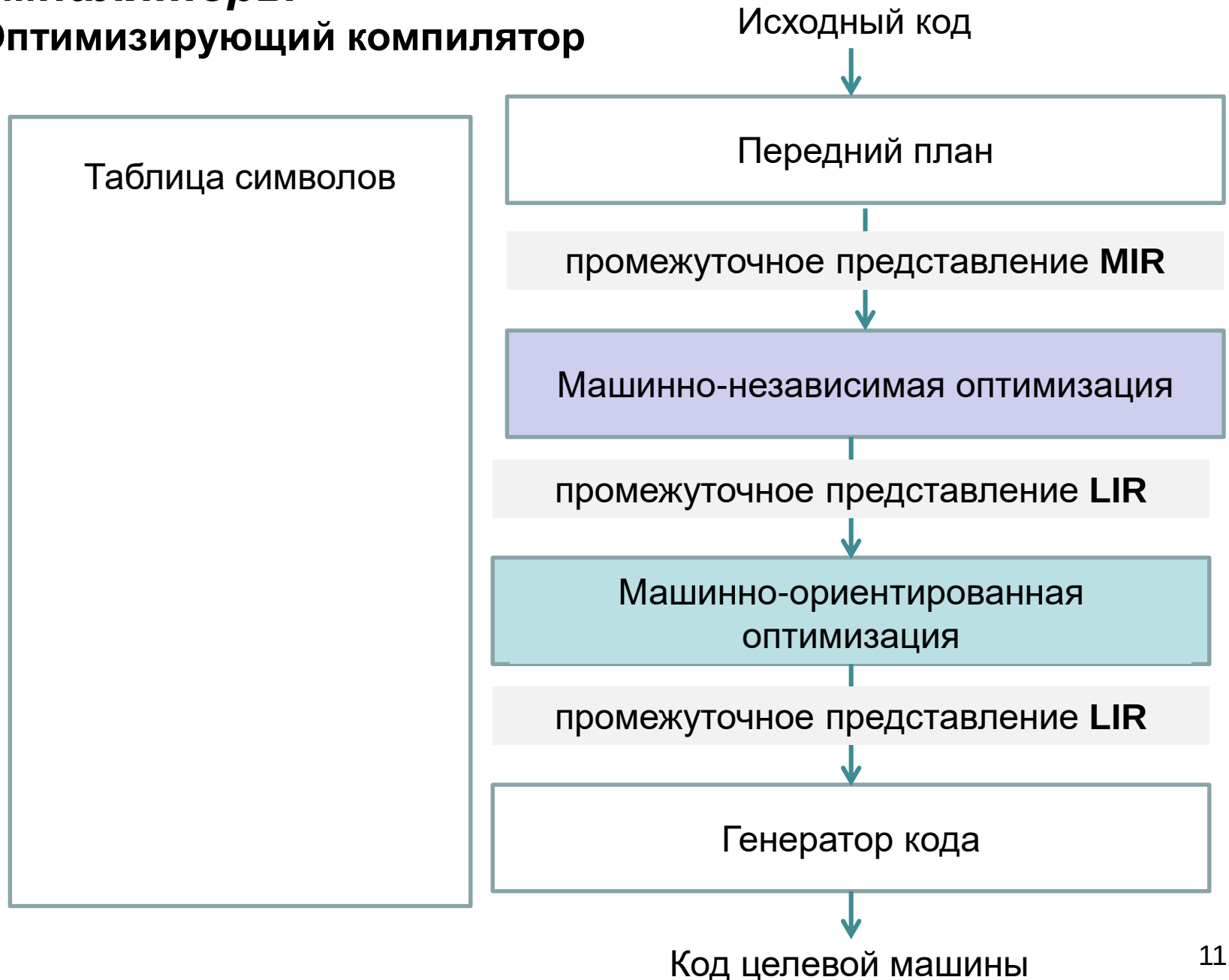
1.1 Компиляторы

1.1.3. Оптимизирующий компилятор



1.1 Компиляторы

1.1.3. Оптимизирующий компилятор



1.2 Цель курса

- ◇ Цель курса
 - ♦ изучение методов оптимизации программ (как машинно-независимой, так и машинно-ориентированной)
 - ♦ изучение принципов конструирования оптимизирующих компиляторов.
- ◇ Основное учебное пособие:
А. В. Ахо, М. С. Лам, Р. Сети, Дж. Д. Ульман. Компиляторы: принципы, технологии и инструменты, 2-е издание.
М.: «И.Д. Вильямс», 2008
(в конце 2014 года был выпущен дополнительный тираж)
- ◇ В тех случаях, когда излагаемый материал в указанном учебном пособии отсутствует, используется еще одно учебное пособие:
Keith D. Cooper, Linda Torczon. Engineering a Compiler (Second Edition) Elsevier, Inc. 2012 (на русский оно пока не переведено)
либо статьи из журналов (как правило, на английском языке).

1.3 Генерация промежуточного представления MIR

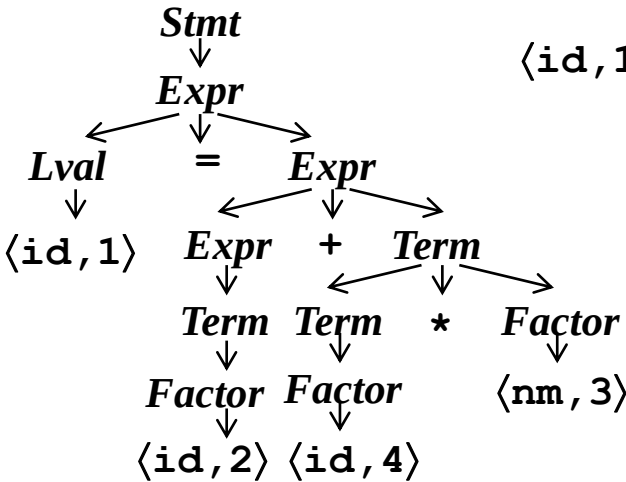
`current_pos=initial_pos+16*scale` Строка исходной программы

Анализатор лексики

`<id,1><=><id,2><+><nm,3><*><id,4>`

Последовательность токенов

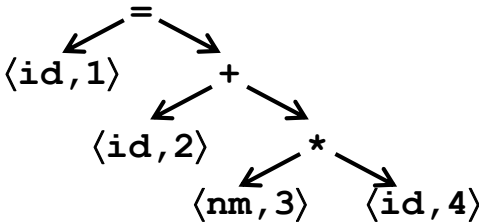
Анализатор синтаксиса



Дерево разбора

Дерево разбора

Анализатор семантики



Абстрактное синтаксическое
дерево
(HIR)

Таблица символов

Внутрен- нее имя	Внешнее имя	Атрибуты
id1	current_pos	int
id2	initial_pos	int
nm3	16	int
id4	step	int

Генератор промежуточного
представления

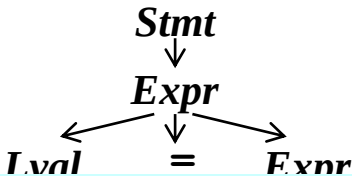
`t1 ← *, nm3, id4`
`t2 ← +, id2, t1`
`id1 ← t2`

MIR

1.3 Генерация промежуточного представления MIR

`current_pos=initial_pos+16*scale` Строка исходной программы

Анализатор лексики



`<id,1><=><id,2><+><nm,3><*><id,4>`

Последовательность токенов

Анализатор синтаксиса

Почему для оптимизации необходимо промежуточное представление MIR? Потому что промежуточное представление HIR – это одна из форм линеаризованного представления дерева: скобочная структура или один из видов польской записи. И в том, и в другом случае трудно представить оптимизируемую программу в виде последовательности инструкций. Например, в инверсной польской записи строка исходной программы `current_pos=initial_pos+16*scale` будет иметь вид `<=><id,1><+><id,2><*><nm,3><id,4>`, причем интерпретироваться она будет с правого, а не левого конца.

id2	initial_pos	int
nm3	16	int
id4	step	int

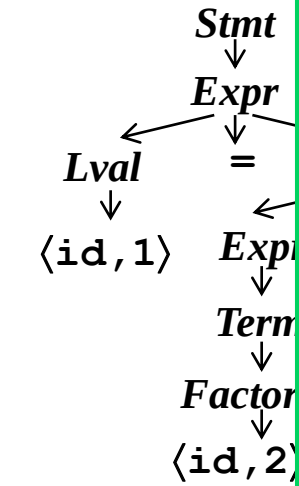
представления

`t1 ← *,nm3,id4`
`t2 ← +,id2,t1`
`id1 ← t2`

MIR

1.3 Генерация промежуточного представления MIR

`current_pos=initial_pos+16*scale` Строка исходной программы



Промежуточное представление **MIR** включает

- ♦ «четверки» (трехадресные инструкции),
- ♦ таблицу символов

Пример «четверки»: `*`, `t1`, `nm3`, `id4`
или `t1 ← *, nm3, id4`

Таблица символов

Внутрен- нее имя	Вн	
id1	current_pos	int
id2	initial_pos	int
nm3	16	int
id4	step	int

Генератор промежуточного
представления

```
t1 ← *,nm3,id4
t2 ← +,id2,t1
id1 ← t2
```

MIR

1.4. Промежуточное представление MIR

1.4.1. Определение промежуточного представления MIR

◇ Инструкции присваивания (x , y и z – имена (адреса) переменных или константы):	
x $\leftarrow$ op , y , z	выполнить бинарную операцию op над операндами y и z и поместить результат в x
x $\leftarrow$ op , y	выполнить унарную операцию op над операндом y и поместить результат в x
x $\leftarrow$ y	скопировать значение переменной y в переменную x
x [i] $\leftarrow$ y	поместить значение y в i -ю по отношению к x ячейку памяти
x $\leftarrow$ y [i]	поместить значение i -ой по отношению к y ячейки памяти в x

Не хватает указателей и адресной арифметики (т.н. *L*-выражений в терминологии языка C):

x[**i**] – простейший частный случай *L*-выражения

1.4. Промежуточное представление MIR

1.4.1. Определение промежуточного представления MIR

◇ Инструкции перехода:	
<code>goto L</code>	Безусловный переход: следующей будет выполнена инструкция с меткой <i>L</i>
<code>ifTrue x goto L</code>	Условный переход: если <i>x</i> истинно, следующей будет выполнена инструкция с меткой <i>L</i>
<code>ifFalse x goto L</code>	Условный переход: если <i>x</i> ложно, следующей будет выполнена инструкция с меткой <i>L</i>

На рисунке справа – схема трансляции оператора

```
if (expr) stmt1;
```

По аналогии можно составить схемы трансляции операторов **else**, **while**, **for**, **switch** и других операторов управления

Инструкции внутреннего представления 2, вычисляющие значение выражения expr и присваивающие его x	
ifFalse x goto L	
Инструкции внутреннего представления 2, соответствующие оператору stmt1	
L:	Инструкции внутреннего представления 2, соответствующие оператору, следующему за оператором if 17

1.4. Промежуточное представление MIR

1.4.1. Определение промежуточного представления MIR

◇ Процедуры:

param x

Передача фактического параметра вызываемой процедуре (если вызываемая процедура имеет n параметров, то инструкции ее вызова предшествует n инструкций *param*)

call p, n

Вызов процедуры p , имеющей n параметров

return y

Возврат из процедуры
 y – необязательное возвращаемое значение

Замечание. Это промежуточное представление – учебное и не содержит некоторых деталей, важных для реализации. Соответствующее промежуточное представление **MIR** системы *GCC* называется *Gimple*.

Для выполнения задания будет использоваться промежуточное представление (биткод) *IR* системы *LLVM*, которое является внутренним представлением **LIR**. Сокращенное описание, достаточное для выполнения заданий – на сайте курса, полное описание – на сайте <https://llvm.org/docs/LangRef.html>

1.4. Промежуточное представление MIR

1.4.2. Пример программы в промежуточном представлении MIR

```
{
int i, j;
int v, x;
if (n <= m) return;
/* Начало фрагмента */
i = m - 1;
j = n;
v = a[n];
while (1) {
do i = i + 1;
while (a[i] < v);
do j = j - 1;
while (a[j] > v);
if (i >= j) break;
/* Обмен a[i], a[j] */
x = a[i];
a[i] = a[j];
a[j] = x;
}
/* Обмен a[i], a[n] */
x = a[i];
a[i] = a[n];
a[n] = x;
/* Конец фрагмента */
quicksort(a,m,j); quicksort(a,i+1,n);
}
```

(1)	i ← -, m, 1	(16)	t7 ← *, 4, i
(2)	j ← n	(17)	t8 ← *, 4, j
(3)	t1 ← *, 4, n	(18)	t9 ← a[t8]
(4)	v ← a[t1]	(19)	a[t7] ← t9
(5)	L1: i ← +, i, 1	(20)	t10 ← *, 4, j
(6)	t2 ← *, 4, i	(21)	a[t10] ← x
(7)	t3 ← a[t2]	(22)	goto L1
(8)	ifTrue t3<v goto L1	(23)	L3: t11 ← *, 4, i
(9)	L2: j ← -, j, 1	(24)	x ← a[t11]
(10)	t4 ← *, 4, j	(25)	t12 ← *, 4, i
(11)	t5 ← a[t4]	(26)	t13 ← *, 4, n
(12)	ifTrue t5>v goto L2	(27)	t14 ← a[t13]
(13)	ifTrue i>=j goto L3	(28)	a[t12] ← t14
(14)	t6 ← *, 4, i	(29)	t15 ← *, 4, n
(15)	x ← a[t6]	(30)	a[t15] ← x

Слева – текст исходной Си-программы 19

1.5. Граф потока управления

1.5.1. Базовый блок (определение)

- ◇ *Базовым блоком (или линейным участком)* называется последовательность следующих одна за другой инструкций **MIR**, со следующими свойствами:
- (1) поток управления может входить в базовый блок только через его первую инструкцию, т.е. в программе нет переходов в середину базового блока;
 - (2) поток управления покидает базовый блок без останова или ветвления, за исключением, возможно, в последней инструкции базового блока.
- ◇ Пример базового блока – инструкции (14) – (22) из примера 1.4.2

```
(14) t6 ← *, 4, i
(15) x ← a[t6]
(16) t7 ← *, 4, i
(17) t8 ← *, 4, j
(18) t9 ← a[t8]
(19) a[t7] ← t9
(20) t10 ← *, 4, j
(21) a[t10] ← x
(22) goto L1
```


1.5. Граф потока управления

1.5.2. Граф потока управления (определение)



Вершинами графа потока управления являются базовые блоки, а дуги соединяют выход из вершины со входом в вершину, которая может выполняться следующей

В частности, если последней инструкцией базового блока является инструкция условного перехода, то из него будет выходить две дуги. Если первая инструкция базового блока имеет метку, то в этот базовый блок будут входить дуги из всех базовых блоков, последняя инструкция которых будет инструкцией условного или безусловного перехода на эту метку.

На первом курсе, изучая программирование, иногда строят граф потока управления программы, которую собираются составить, считая, что этот граф упрощает составление программы. Для некоторых небольших программ это действительно может помочь, но для больших программ, граф потока управления может иметь несколько сотен вершин, так что построить его, не имея текста программы, нереально.

1.5. Граф потока управления

1.5.3. Алгоритм построения графа потока управления (ГПУ)

- ◇ **Вход:** последовательность трехадресных инструкций.
- ◇ **Выход:** список базовых блоков для данной последовательности инструкций, такой что каждая инструкция принадлежит только одному базовому блоку.
- ◇ **Метод:**
 - ◇ Строится упорядоченное множество НББ
 - ◇ Каждому НББ соответствует ББ, который определяется как последовательность инструкций, содержащая само НББ и все инструкции до следующего НББ (не включая его) или до конца последовательности инструкций.
 - ◇ Строится множество дуг графа потока управления:
 - ◆ если последняя инструкция ББ не является инструкцией перехода, строится дуга, соединяющая ББ со следующим ББ;
 - ◆ если последняя инструкция ББ является инструкцией безусловного перехода, строится дуга, соединяющая ББ с ББ, НББ которого имеет соответствующую метку;
 - ◆ если последняя инструкция ББ является инструкцией условного перехода, строятся обе дуги.

1.5. Граф потока управления

1.5.4. Пример

◇ Применим алгоритм 1.5.3 для построения ГПУ программы из примера 1.4.2.

(1)	<code>i ← -, m, 1</code>	(16)	<code>t7 ← *, 4, i</code>
(2)	<code>j ← n</code>	(17)	<code>t8 ← *, 4, j</code>
(3)	<code>t1 ← *, 4, n</code>	(18)	<code>t9 ← a[t8]</code>
(4)	<code>v ← a[t1]</code>	(19)	<code>a[t7] ← t9</code>
(5)	<code>L1: i ← +, i, 1</code>	(20)	<code>t10 ← *, 4, j</code>
(6)	<code>t2 ← *, 4, i</code>	(21)	<code>a[t10] ← x</code>
(7)	<code>t3 ← a[t2]</code>	(22)	<code>goto L1</code>
(8)	<code>ifTrue t3<v goto L1</code>	(23)	<code>L3: t11 ← *, 4, i</code>
(9)	<code>L2: j ← -, j, 1</code>	(24)	<code>x ← a[t11]</code>
(10)	<code>t4 ← *, 4, j</code>	(25)	<code>t12 ← *, 4, i</code>
(11)	<code>t5 ← a[t4]</code>	(26)	<code>t13 ← *, 4, n</code>
(12)	<code>ifTrue t5>v goto L2</code>	(27)	<code>t14 ← a[t13]</code>
(13)	<code>ifTrue i>=j goto L3</code>	(28)	<code>a[t12] ← t14</code>
(14)	<code>t6 ← *, 4, j</code>	(29)	<code>t15 ← *, 4, n</code>
(15)	<code>x ← a[t6]</code>	(30)	<code>a[t15] ← x</code>

1.5. Граф потока управления

1.5.4. Пример

◇ Найдем начала базовых блоков (НББ)

(1)	<code>i ← -, m, 1</code>	(16)	<code>t7 ← *, 4, i</code>
(2)	<code>j ← n</code>	(17)	<code>t8 ← *, 4, j</code>
(3)	<code>t1 ← *, 4, n</code>	(18)	<code>t9 ← a[t8]</code>
(4)	<code>v ← a[t1]</code>	(19)	<code>a[t7] ← t9</code>
(5)	<code>t1 ← *, 4, i</code>	(20)	<code>t10 ← *, 4, i</code>
(6)	Напоминаю: НББ по определению это: ◇ первая инструкция программы ◇ помеченная инструкция программы ◇ инструкция, следующая за инструкцией перехода		
(7)			
(8)			
(9)	<code>L2: j ← -, j, 1</code>	(24)	<code>x ← a[t11]</code>
(10)	<code>t4 ← *, 4, j</code>	(25)	<code>t12 ← *, 4, i</code>
(11)	<code>t5 ← a[t4]</code>	(26)	<code>t13 ← *, 4, n</code>
(12)	<code>ifTrue t5>v goto L2</code>	(27)	<code>t14 ← a[t13]</code>
(13)	<code>ifTrue i>=j goto L3</code>	(28)	<code>a[t12] ← t14</code>
(14)	<code>t6 ← *, 4, j</code>	(29)	<code>t15 ← *, 4, n</code>
(15)	<code>x ← a[t6]</code>	(30)	<code>a[t15] ← x</code>

1.5. Граф потока управления

1.5.4. Пример

◇ Найдем начала базовых блоков (НББ)

(1)	i ← -, m, 1	НББ	(16)	t7 ← *, 4, i	
(2)	j ← n		(17)	t8 ← *, 4, j	
(3)	t1 ← *, 4, n		(18)	t9 ← a[t8]	
(4)	v ← a[t1]		(19)	a[t7] ← t9	
(5)	L1: i ← +, i, 1	НББ	(20)	t10 ← *, 4, j	
(6)	t2 ← *, 4, i		(21)	a[t10] ← x	
(7)	t3 ← a[t2]		(22)	goto L1	
(8)	ifTrue t3<v goto L1		(23)	L3: t11 ← *, 4, i	НББ
(9)	L2: j ← -, j, 1	НББ	(24)	x ←a[t11]	
(10)	t4 ← *, 4, j		(25)	t12 ← *, 4, i	
(11)	t5 ← a[t4]		(26)	t13 ← *, 4, n	
(12)	ifTrue t5>v goto L2		(27)	t14 ← a[t13]	
(13)	ifTrue i>=j goto L3	НББ	(28)	a[t12] ← t14	
(14)	t6 ← *, 4, j	НББ	(29)	t15 ← *, 4, n	
(15)	x ← a[t6]		(30)	a[t15] ← x	

1.5. Граф потока управления

1.5.4. Пример

◇ Найдем начала базовых блоков (НББ)

(1)	<code>i ← -, m, 1</code>	НББ	(16)	<code>t7 ← *, 4, i</code>
(2)	<code>j ← n</code>		(17)	<code>t8 ← *, 4, j</code>
(3)	<code>t1 ← *, 4, n</code>		(18)	<code>t9 ← a[t8]</code>
(4)	<code>v ← a[t1]</code>		(19)	<code>a[t7] ← t9</code>
(5)	<code>L1: i ← +, i, 1</code>	НББ	(20)	<code>t10 ← *, 4, j</code>

Началами базовых блоков являются инструкции с номерами:

(1), (5), (9), (13), (14), (23). Алгоритм 1.5.3 позволяет построить следующие ББ

(8)	<code>ifTrue t3<v goto L1</code>		(23)	<code>L3: t11 ← *, 4, i</code>	НББ
(9)	<code>L2: j ← -, j, 1</code>	НББ	(24)	<code>x ← a[t11]</code>	
(10)	<code>t4 ← *, 4, j</code>		(25)	<code>t12 ← *, 4, i</code>	
(11)	<code>t5 ← a[t4]</code>		(26)	<code>t13 ← *, 4, n</code>	
(12)	<code>ifTrue t5>v goto L2</code>		(27)	<code>t14 ← a[t13]</code>	
(13)	<code>ifTrue i>=j goto L3</code>	НББ	(28)	<code>a[t12] ← t14</code>	
(14)	<code>t6 ← *, 4, j</code>	НББ	(29)	<code>t15 ← *, 4, n</code>	
(15)	<code>x ← a[t6]</code>		(30)	<code>a[t15] ← x</code>	

1.5. Граф потока управления

1.5.4. Пример

Блок А

```
(1)    i ← -, m, 1
(2)    j ← n
(3)    t1 ← *, 4, n
(4)    v ← a[t1]
```

Блок В

```
(5) L1: i ← +, i, 1
(6)    t2 ← *, 4, i
(7)    t3 ← a[t2]
(8)    ifTrue t3<v gotoL1
```

Блок С

```
(9)  L2: j ← -, j, 1
(10)    t4 ← *, 4, j
(11)    t5 ← a[t4]
(12)    ifTrue t5>v goto L2
```

Блок D

```
(13) ifTrue i>=j goto L3
```

Блок Е

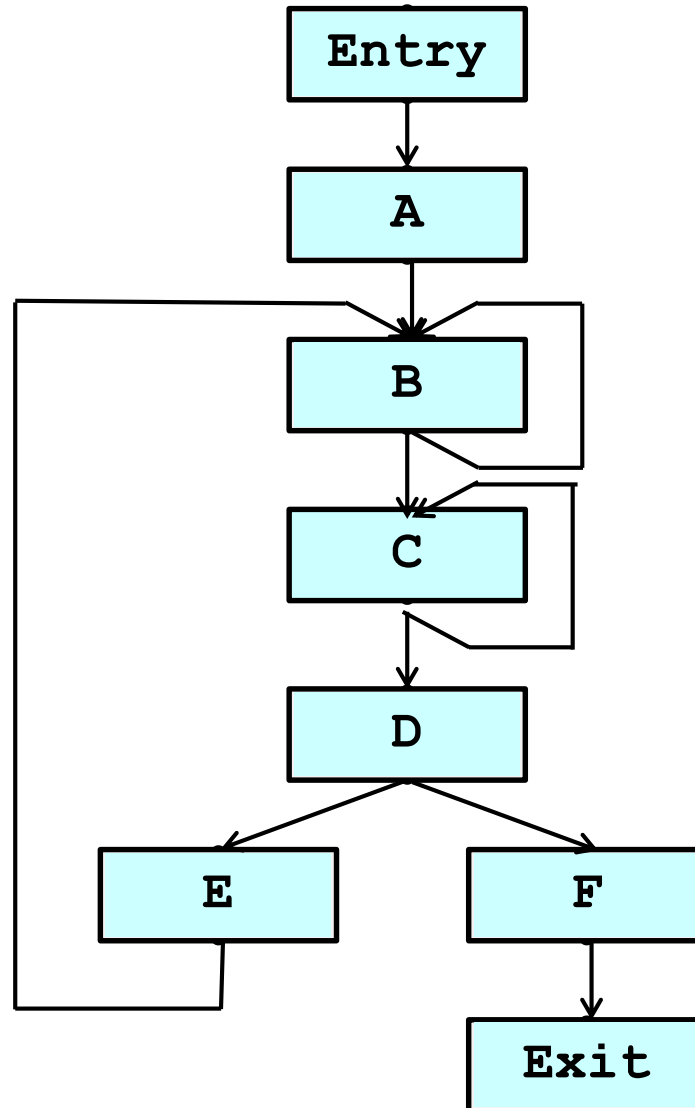
```
(14)    t6 ← *, 4, i
(15)    x ← a[t6]
(16)    t7 ← *, 4, i
(17)    t8 ← *, 4, j
(18)    t9 ← a[t8]
(19)    a[t7] ← t9
(20)    t10 ← *, 4, j
(21)    a[t10] ← x
(22)    goto L1
```

Блок F

```
(23) L3: t11 ← *, 4, i
(24)    x ← a[t11]
(25)    t12 ← *, 4, i
(26)    t13 ← *, 4, n
(27) t14 ← a[t13]
(28) a[t12] ← t14
(29) t15 ← *, 4, n
(30) a[t15] ← x
```

1.5. Граф потока управления

1.5.4. Пример. Построив дуги согласно алгоритму 1.5.3, получим следующий ГПУ. Для удобства анализа к ГПУ добавлены два дополнительных узла: *entry* (вход) и *exit* (выход).



1.6 Локальная оптимизация (оптимизация базовых блоков)

1.6.1 Постановка задачи локальной оптимизации

- ◇ Оптимизация базового блока – это выполнение следующих преобразований:
 - ◇ Удаление *общих подвыражений* (инструкций, повторно вычисляющих уже вычисленные значения).
 - ◇ Удаление *мертвого кода* (инструкций, вычисляющих значения, которые впоследствии не используются).
 - ◇ *Сворачивание констант*.
 - ◇ Изменение порядка инструкций, там, где это возможно, чтобы сократить время хранения временного значения на регистре.
- ◇ Все вышеперечисленные преобразования можно выполнить за один просмотр ББ, представив его в виде *ориентированного ациклического графа* (ОАГ).

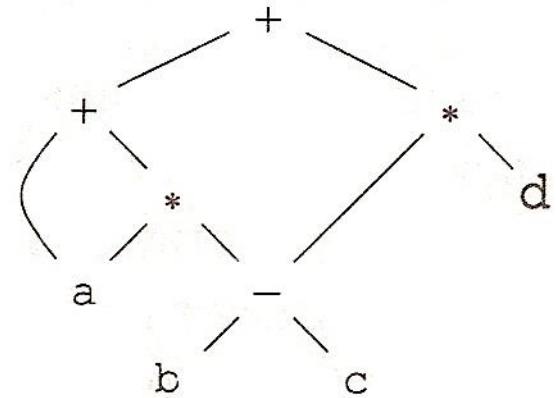
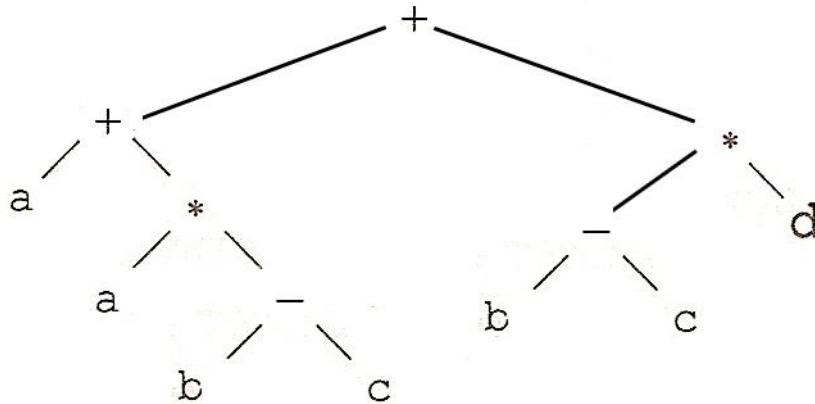
1.6 Локальная оптимизация (оптимизация базовых блоков)

1.6.2 Представление базового блока в виде ориентированного ациклического графа

◇ Простейший пример. Выражение

$$a + a * (b - c) + (b - c) * d$$

можно представить в виде фрагмента АСД (левый рисунок) либо в виде ориентированного ациклического графа (правый рисунок) – *DAG (Directed Acyclic Graph)*



◇ Основное отличие DAG от АСД в том, что в DAG каждое значение представляется только один раз: узлы, представляющие в АСД одинаковые значения, «склеиваются»

1.6 Локальная оптимизация

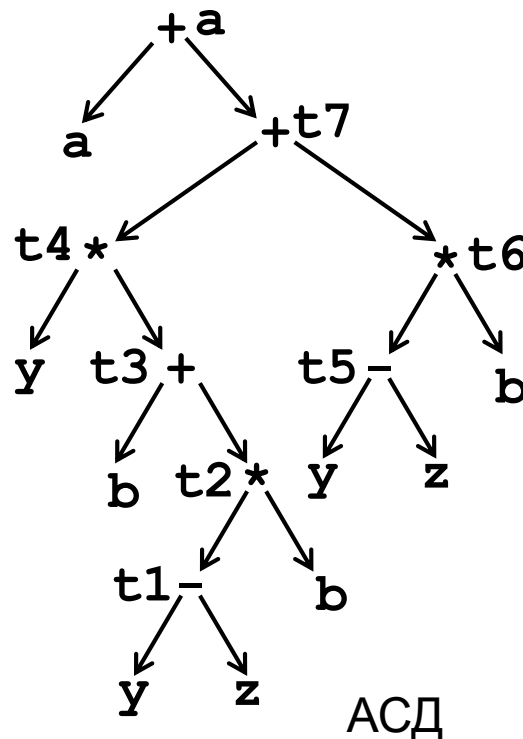
1.6.2 Представление базового блока в виде ориентированного ациклического графа

◇ Пример. Выражение в исходном коде:

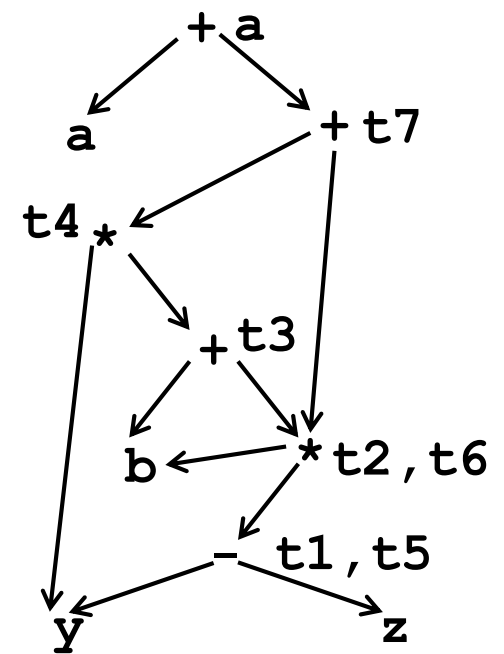
$$a = a + y * (b + (y - z) * b) + (y - z) * b$$

t1 ← -, y, z
t2 ← *, t1, b
t3 ← +, b, t2
t4 ← *, y, t3
t5 ← -, y, z
t6 ← *, t5, b
t7 ← +, t4, t6
a ← +, a, t7

Выражение в
промежуточном
представлении 2



АСД



ОАГ

1.6 Локальная оптимизация

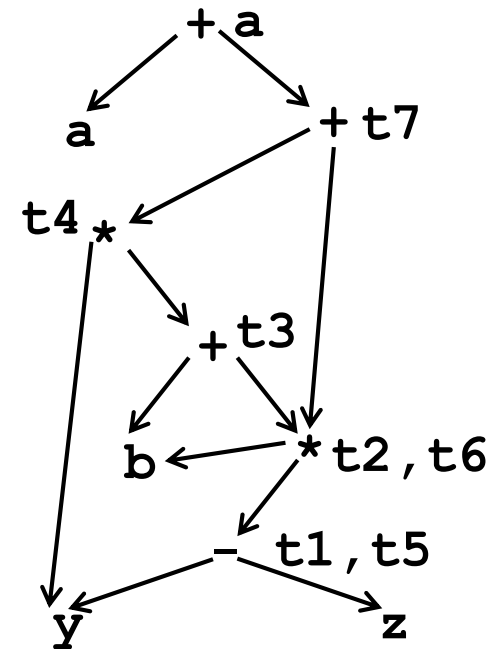
1.6.2 Представление базового блока в виде ориентированного ациклического графа

◇ Пример (окончание).

На ОАГ для выражения видно (в отличие от АСД):

- ◇ переменная **b** используется в двух вычислениях
- ◇ выражение **y-z** можно вычислить только один раз
- ◇ выражение **(y-z) * b** можно вычислить только один раз

```
t1 ← -, y, z
t2 ← *, t1, b
t3 ← +, b, t2
t4 ← *, y, t3
t5 ← -, y, z
t6 ← *, t5, b
t7 ← +, t4, t6
a ← +, a, t7
```



ОАГ

1.6 Локальная оптимизация

1.6.3 Метод нумерации значений – алгоритм построения ОАГ

Представим **ОАГ** в виде таблицы значений

- ◇ Каждая строка таблицы значений представляет узел ОАГ.
- ◇ Первое поле каждой записи содержит код операции
- ◇ Каждой правой части операции
 $\langle \text{op}, \text{left}, \text{right} \rangle$,
где **op** – код операции, а **left** и **right** – левый и правый операнды,
соответствует ее *сигнатура*

$$\langle \text{op}, \# \text{left}, \# \text{right} \rangle,$$

где **op** – код операции, а **#left** и **#right** – номера значений левого и правого операндов (у унарных операций **#right** равен 0).

- ◇ Унарные операции **id** и **nm** определяют соответственно имена переменных и константы (листовые узлы).
- ◇ Номер значения – это номер строки таблицы значений (ТЗ), в которой определяется это значение
- ◇ Для определения номера строки ТЗ обычно используют хеш-функцию, аргументом которой является сигнатура

1.6 Локальная оптимизация

1.6.3 Метод нумерации значений – алгоритм построения ОАГ

Алгоритм (на псевдокоде) построения ОАГ для базового блока B , содержащего n инструкций вида $t_i \leftarrow Op_i, l_i, r_i$.

Функция $\#val(s)$ определяет номер значения, определяемого сигнатурой

$$s = (Op, \#val(l), \#val(r)) .$$

for each " $t_i \leftarrow Op_i, l_i, r_i$ " do

$s_i = (Op_i, \#val(l_i), \#val(r_i))$

 if (ТЗ содержит $s_j == s_i$)

 then

 вернуть j в качестве номера значения $\#val(s_i)$

 else

 завести в ТЗ новую строку $ТЗ_k$

 записать сигнатуру s_i в строку $ТЗ_k$

 вернуть k в качестве номера значения $\#val(s_i)$

1.6 Локальная оптимизация

1.6.4 Представление ОАГ в виде таблицы значений. Пример.

◇ Таблица значений рассматриваемого примера имеет вид:

$t1^5 \leftarrow -, y^3, z^4$
 $t2^6 \leftarrow *, t1^5, b^2$
 $t3^7 \leftarrow +, b^2, t2^6$
 $t4^8 \leftarrow *, y^3, t3^7$
 $t5^5 \leftarrow -, y^3, z^4$
 $t6^6 \leftarrow *, t5^5, b^2$
 $t7^9 \leftarrow +, t4^8, t6^6$
 $a^{10} \leftarrow +, a^1, t7^9$

1	id	ссылка в ТС		a
2	id	ссылка в ТС		b
3	id	ссылка в ТС		y
4	id	ссылка в ТС		z
5	−	3	4	t1, t5
6	*	5	2	t2, t6
7	+	2	6	t3
8	*	3	7	t4
9	+	8	6	t7
10	+	1	9	a
# значения	КОП	# операнда	# операнда	Присоединенные переменные
	Определение значения (сигнатура)			

1.6 Локальная оптимизация

1.6.4 Представление ОАГ в виде таблицы значений. Пример.

$t_1 \leftarrow -, y, z$
 $t_2 \leftarrow *, t_1, b$
 $t_3 \leftarrow +, b, t_2$
 $t_4 \leftarrow *, y, t_3$
 $t_5 \leftarrow -, y, z$
 $t_6 \leftarrow *, t_5, b$
 $t_7 \leftarrow +, t_4, t_6$
 $a \leftarrow +, a, t_7$

(а) Блок Е
до оптимизации

$t_1^5 \leftarrow -, y^3, z^4$
 $t_2^6 \leftarrow *, t_1^5, b^2$
 $t_3^7 \leftarrow +, b^2, t_2^6$
 $t_4^8 \leftarrow *, y^3, t_3^7$
 $t_5^5 \leftarrow -, y^3, z^4$
 $t_6^6 \leftarrow *, t_5^5, b^2$
 $t_7^9 \leftarrow +, t_4^8, t_6^6$
 $a^{10} \leftarrow +, a^1, t_7^9$

(с) Блок Е
после нумерации
значений

$t_1 \leftarrow -, y, z$
 $t_2 \leftarrow *, t_1, b$
 $t_3 \leftarrow +, b, t_2$
 $t_4 \leftarrow *, y, t_3$
 $t_5 \leftarrow t_1$
 $t_6 \leftarrow t_2$
 $t_7 \leftarrow +, t_4, t_6$
 $a \leftarrow +, a, t_7$

(с) Блок Е
после оптимизации

1.6 Локальная оптимизация

1.6.5 Представление ОАГ в виде таблицы значений. Пример

◇ Заполнить таблицу значений для базового блока В:

$B = \langle P, \{x, y, z\}, \{v, w, z\} \rangle$

$P = \{$
1. $x \leftarrow 3$
2. $y \leftarrow 5$
3. $t1 \leftarrow +, x, y$
4. $t2 \leftarrow +, x, y$
5. $w \leftarrow *, t1, t2$
6. $t3 \leftarrow -, x, y$
7. $t4 \leftarrow *, w, x$
8. $v \leftarrow -, t4, z$
9. $t5 \leftarrow +, x, y$
10. $y \leftarrow +, t5, z$
11. $x \leftarrow +, x, y$
12. $v \leftarrow +, x, y$
13. $z \leftarrow +, z, y$
14. $y \leftarrow *, x, z$
15. $x \leftarrow *, t3, t4\}$

1	nm	ссылка в ТС		3
2	nm	ссылка в ТС		5
3	id	ссылка в ТС		z
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
# значе- ния	КОП	# операнда	# операнда	Переменные
	Определение значения (сигнатура)			

1.6 Локальная оптимизация

1.6.5 Представление ОАГ в виде таблицы значений. Пример

◇ Заполнить таблицу значений для базового блока В:

$B = \langle P, \{x, y, z\}, \{v, w, z\} \rangle$

$P = \{$
 1. $x^1 \leftarrow 3^1$
 2. $y^2 \leftarrow 5^2$
 3. $t1^4 \leftarrow +, x^1, y^2$
 4. $t2^4 \leftarrow +, x^1, y^2$
 5. $w^5 \leftarrow *, t1^4, t2^4$
 6. $t3^6 \leftarrow -, x^1, y^2$
 7. $t4^7 \leftarrow *, w^5, x^1$
 8. $v^8 \leftarrow -, t4^7, z^3$
 9. $t5^4 \leftarrow +, x^1, y^2$
 10. $y^9 \leftarrow +, t5^4, z^3$
 11. $x^{10} \leftarrow +, x^1, y^9$
 12. $v^{11} \leftarrow +, x^{10}, y^9$
 13. $z^{12} \leftarrow +, z^3, y^9$
 14. $y^{13} \leftarrow *, x^{10}, z^{12}$
 15. $x^{14} \leftarrow *, t3^6, t4^7\}$

1	nm	ссылка в ТС		3 x
2	nm	ссылка в ТС		5 y
3	id	ссылка в ТС		z
4	+	1	2	t1 t2 t5
5	*	4	4	w
6	−	1	2	t3
7	*	5	1	t4
8	−	7	3	v
9	+	4	3	y
10	+	1	9	x
11	+	10	9	v
12	+	3	9	z
13	*	10	12	y
14	*	6	7	x
# значе- ния	КОП	# операнда	# операнда	Переменные
	Определение значения (сигнатура)			

1.6 Локальная оптимизация

1.6.6 Удаление общих подвыражений

- ◇ Общие подвыражения обнаруживаются при построении узлов ОАГ с помощью алгоритма 1.6.3 (нумерация значений). Так в примере 1.6.5 три раза вычисляется одно и то же значение № 4 (в инструкциях № 3, 4 и 9). Удаление общих подвыражений позволит заменить инструкции 4 и 9 на $t2 \leftarrow t1$ и $t5 \leftarrow t1$.

1.6.7 Сворачивание констант

- ◇ *Сворачивание констант* заключается в вычислении констант в процессе компиляции и замене константных выражений их значениями. В примере 1.6.5 значение №4 равно $x + y$, причем $x = 3$, $y = 5$. Сворачивание константы позволит заменить присваивание $t1 \leftarrow +, x, y$ на присваивание $t1 \leftarrow 8$

1.6 Локальная оптимизация

1.6.8 Снижение стоимости вычислений

- ◇ Еще один класс алгебраических оптимизаций включает локальное *снижение стоимости вычислений*, т.е. заменяет более дорогие с операции более дешевыми (см. таблицу). Так в примере 1.6.5 инструкция № 5 могла бы быть написана как $w \leftarrow \uparrow, t1, \#2$ ($\uparrow$ обозначает операцию возведения в степень), но, как следует из таблицы, операция умножения, использованная в инструкции, является более дешевой. Если программист использовал не умножение, а возведение в степень, оптимизатор исправил бы его оплошность.

Дорогие	Дешевые
x^2	$x \times x$
$2 \times x$	$x + x$
$x/2$	$x \times 0.5$

1.6 Локальная оптимизация

1.6.9 Удаление мертвого кода

- ◇ Преобразование ОАГ, соответствующее удалению мертвого кода, состоит в удалении из ОАГ любого корня (ор), с которым не связаны живые переменные.
- ◇ *Живыми* называются переменные, значения которых, вычисленные в рассматриваемом базовом блоке, используются в других базовых блоках
- ◇ Для полноценного удаления мертвого кода необходимо уточнить определение базового блока.
- ◇ **Определение.** Базовым блоком называется тройка
$$B = \langle P, Input, Output \rangle,$$
где P – последовательность инструкций блока B ,
 $Input$ – множество переменных, определенных до входа в блок B ,
 $Output$ – множество переменных, используемых после выхода из блока B .

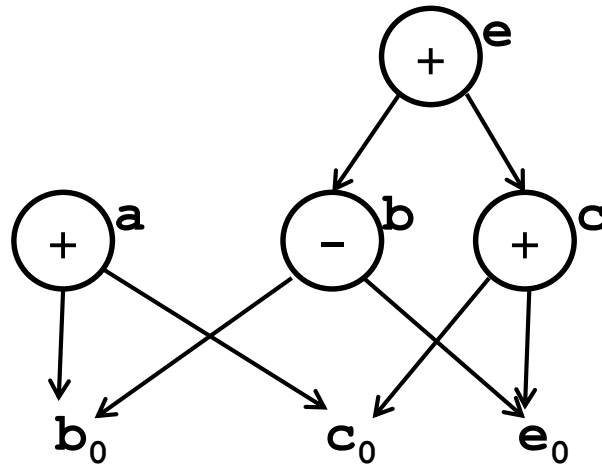
1.6 Локальная оптимизация

1.6.9 Удаление мертвого кода

◇ **Пример.** Рассмотрим базовый блок $B = \langle P, \{a, b, c, d\}, \{a, b\} \rangle$

a	←	+	,	b	,	c
b	←	-	,	b	,	d
c	←	+	,	c	,	d
e	←	+	,	b	,	c

Базовый блок B до
удаления мертвого
кода



ОАГ базового блока B

a	←	+	,	b	,	c
b	←	-	,	b	,	d

Новые значения
c и **e** можно не
вычислять, так как
c и **e** не входят в
множество *Output*

Базовый блок B после
удаления мертвого
кода

1.7 Восстановление базового блока по его ОАГ

1.7.1. Правила

- ◇ Для каждого узла с одной или несколькими связанными переменными строится трехадресная инструкция, которая вычисляет значение одной из этих переменных.
- ◇ Если у узла несколько присоединенных живых переменных, то следует добавить команды копирования, которые присвоят корректное значение каждой из этих переменных.

1.7 Восстановление базового блока по его ОАГ

1.7.2. Пример

◇ Пусть ОАГ представлен следующим массивом записей.

1	b_0	ссылка в ТС		
2	c_0	ссылка в ТС		
3	d_0	ссылка в ТС		
4	+	1	2	a
5	-	4	3	d, b
6	+	5	2	c
8	a	ссылка в ТС		
9	b	ссылка в ТС		
10	c	ссылка в ТС		
11	d	ссылка в ТС		

◇ Применяя правила из п. 1.7.1, получим:

Базовый блок

$B = \langle P, Input, Output \rangle$

$P:$

$$\begin{aligned} a &\leftarrow +, b_0, c_0 \\ d &\leftarrow -, a, d_0 \\ c &\leftarrow +, d, c_0 \\ b &\leftarrow d \end{aligned}$$

$Input = \{b_0, c_0, d_0\}$

$Output = \{a, b, c, d\}$

1.8 Локальная оптимизация. Заключительные замечания

1.8.1. Заключительные замечания

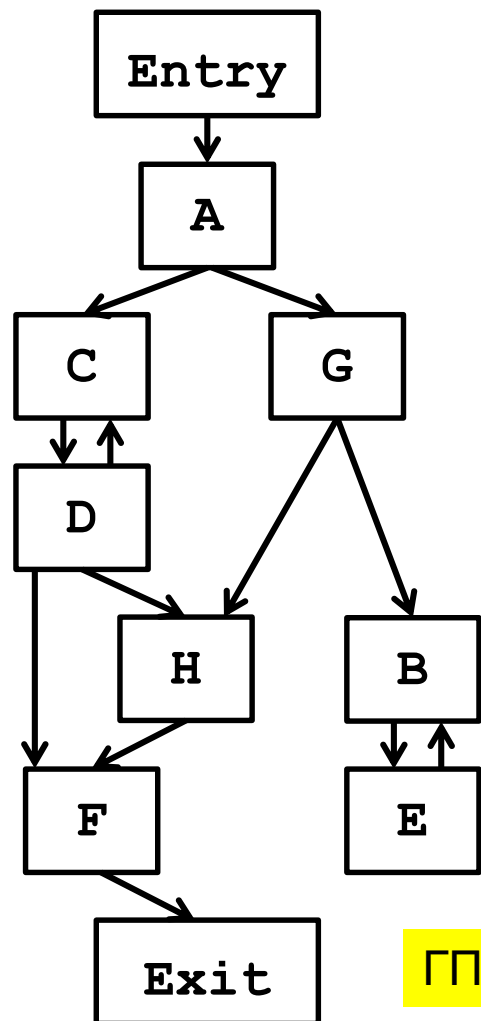
- ◇ Стремление разработать полноценную локальную оптимизацию привело к необходимости построить связи по данным оптимизируемого базового блока с другими базовыми блоками, т.е. к необходимости глобального анализа оптимизируемой процедуры (функции, метода).
- ◇ Необходимо научиться строить базовые блоки больших размеров, чтобы повысить эффективность локальной оптимизации.
- ◇ Ссылаться на базовые блоки по их идентификаторам неудобно. Необходимо научиться нумеровать базовые блоки, чтобы их можно было называть B_1 , B_2 и т.д. Решение этой задачи связано с обходом ГПУ.
- ◇ Далее будут рассмотрены методы решения перечисленных задач и некоторых других задач, которые будут возникать по ходу дела.

2. Построение множеств Input и Output

2.1 Нумерация вершин ГПУ

2.1.1 Остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию»



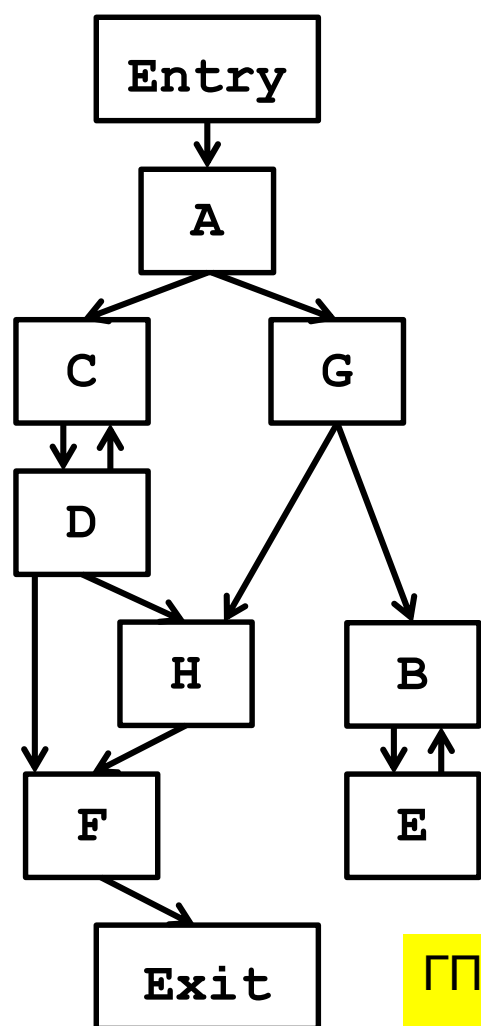
Остовное дерево графа содержит **все** вершины графа и часть его дуг.

Обратная нумерация используется для того, чтобы, например, вершина **A** имела номер 1, а не 8.

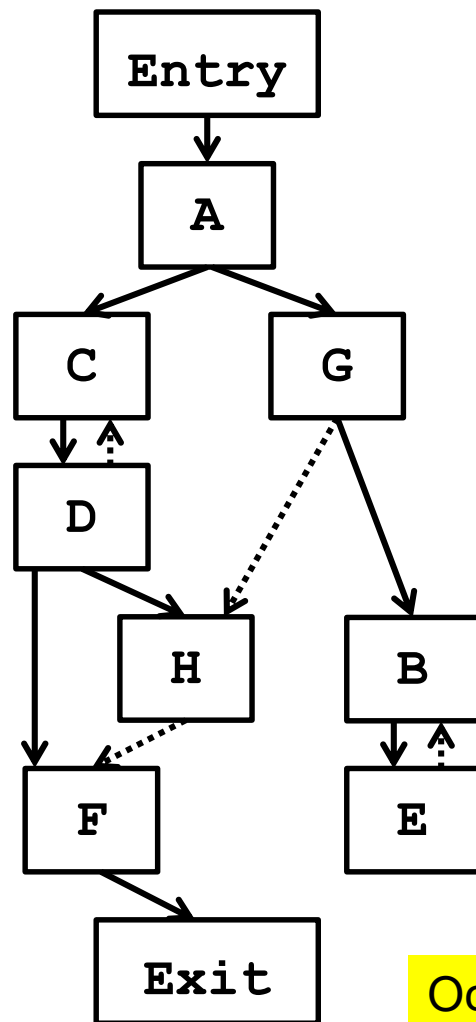
2.1 Нумерация вершин ГПУ

2.1.2 Глубинное остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию»



ГПУ



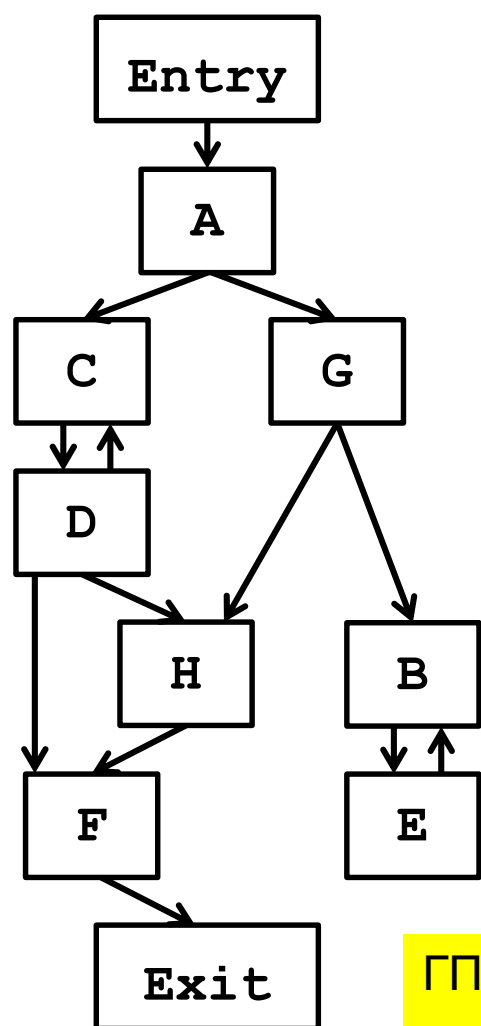
Остовное дерево

«Лишние» дуги ГПУ, не вошедшие в ОД, на рисунке изображены пунктиром

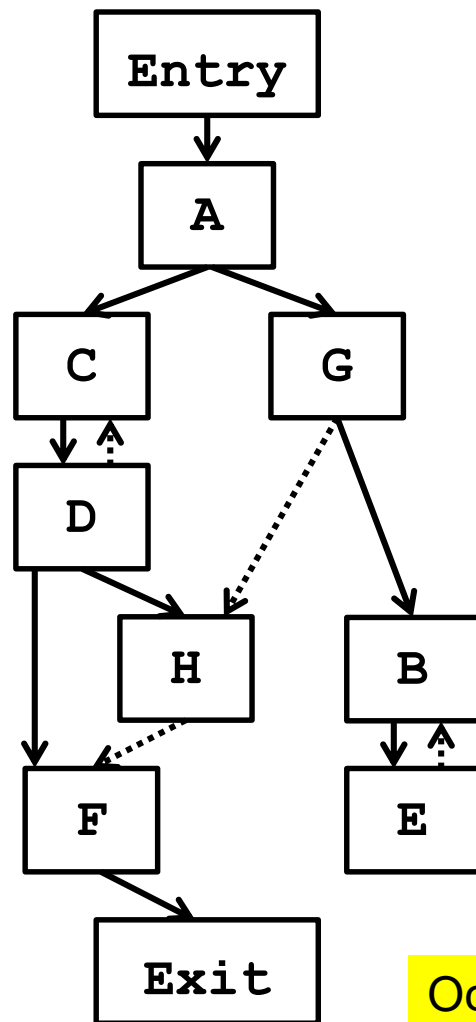
2.1 Нумерация вершин ГПУ

2.1.2 Глубинное остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию»



ГПУ



Остовное дерево

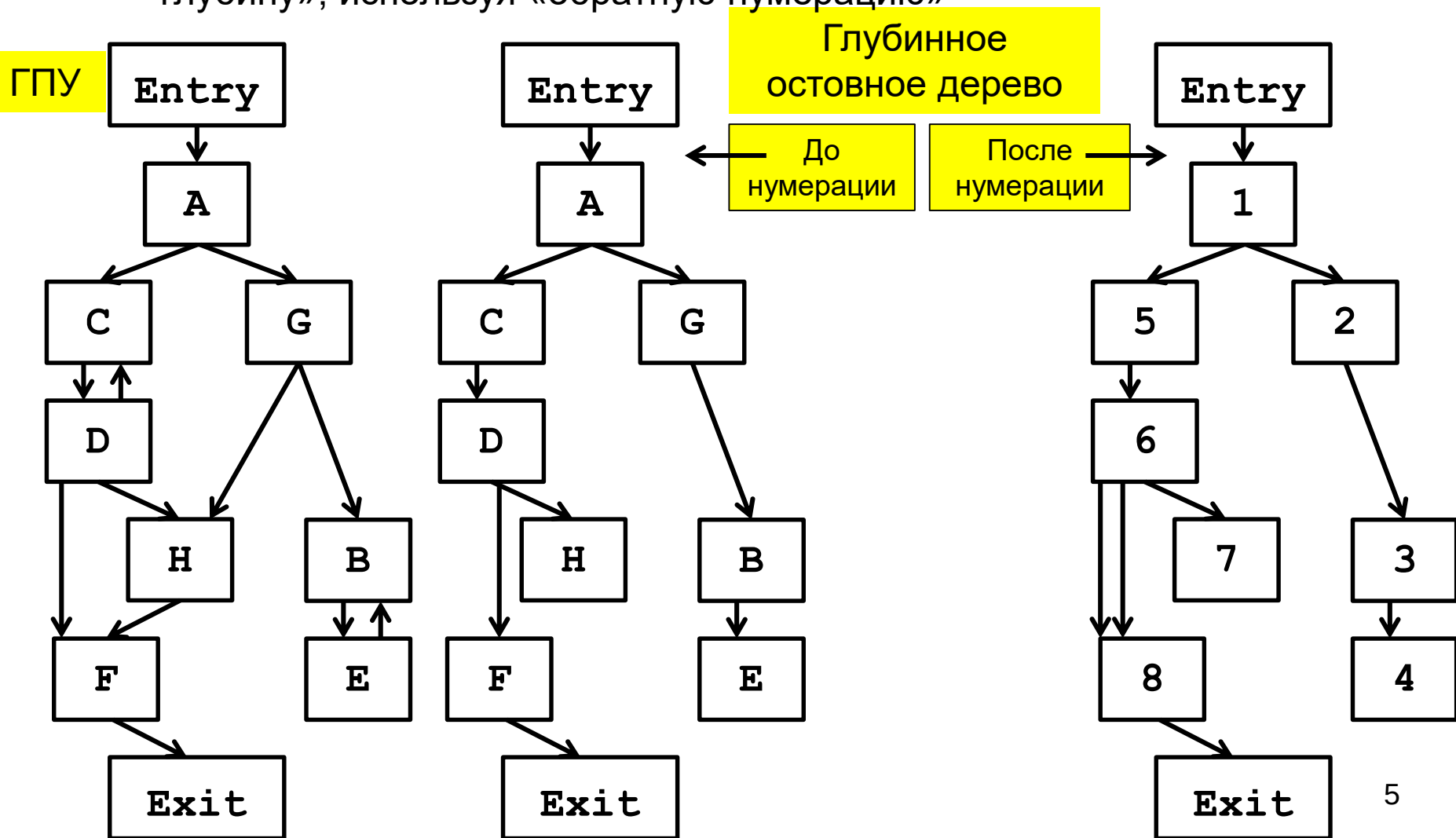
«Лишние» дуги ГПУ, не вошедшие в ОД, на рисунке изображены пунктиром

Такое остовное дерево будем называть глубинным остовным деревом, чтобы помнить, как пронумерованы его вершины

2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

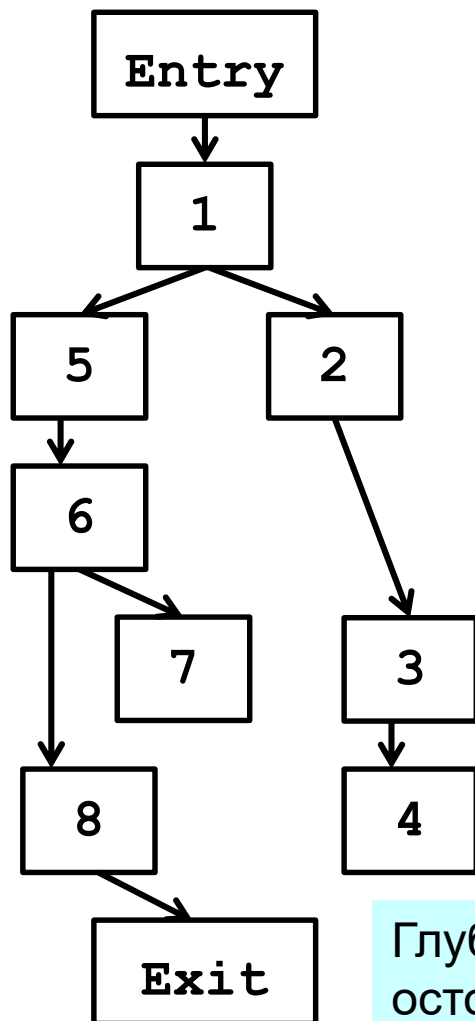
- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем его слева направо «сначала в глубину», используя «обратную нумерацию»



2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем его слева направо «сначала в глубину», используя «обратную нумерацию»



Глубинное
остовное дерево

В случае обратной нумерации вершин графа, содержащего n вершин i -ой вершине присваивается номер $n - i$

На остовном дереве идентификаторы вершин заменяем их номерами

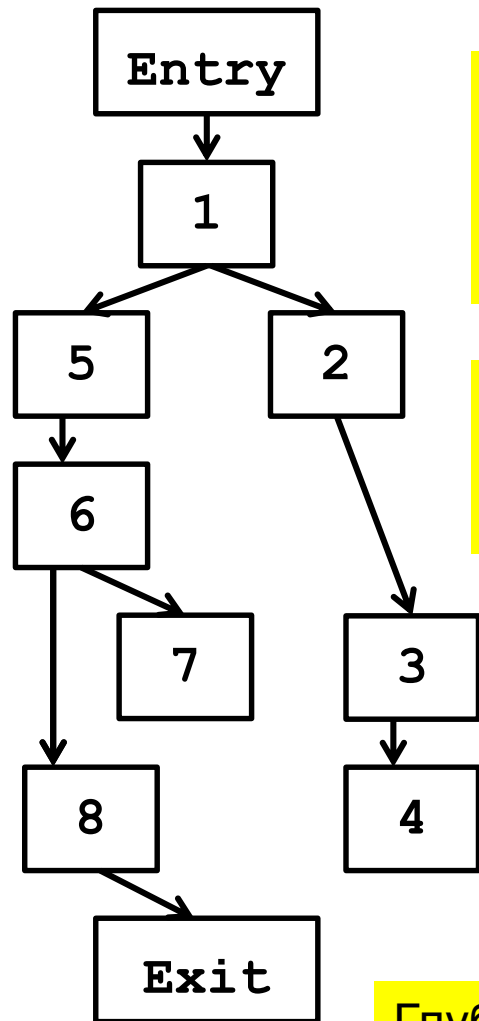
Остовное дерево с корнем в **Entry** и такой нумерацией вершин называется *глубинным остовным деревом* – *DFST*).

DFST – *Depth First Spanning Tree*

2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

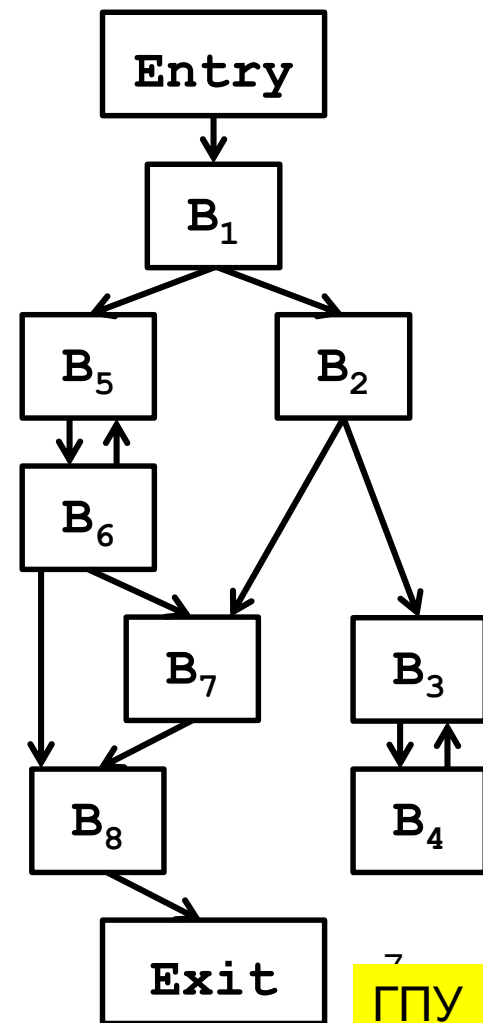
- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем его слева направо «сначала в глубину», используя «обратную нумерацию»



Присвоив номера вершин остовного дерева соответствующим базовым блокам, получим ГПУ с пронумерованными вершинами.

Нумерация вершин ГПУ определяет порядок их посещения во время обходов ГПУ

Глубинное остовное дерево

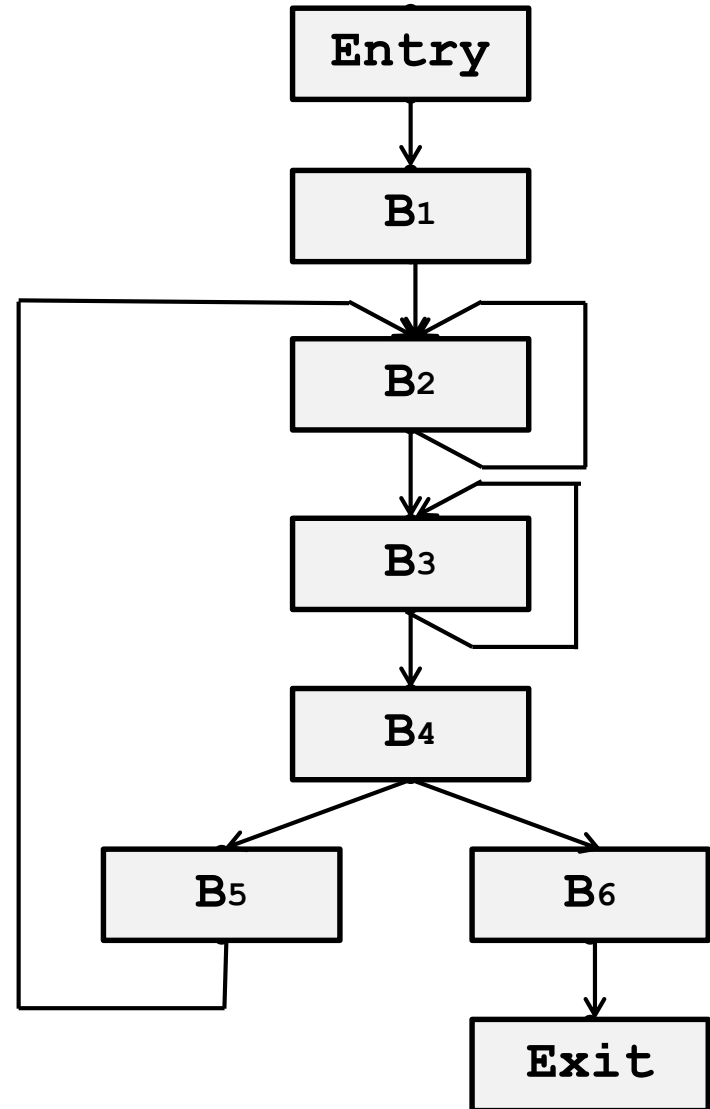
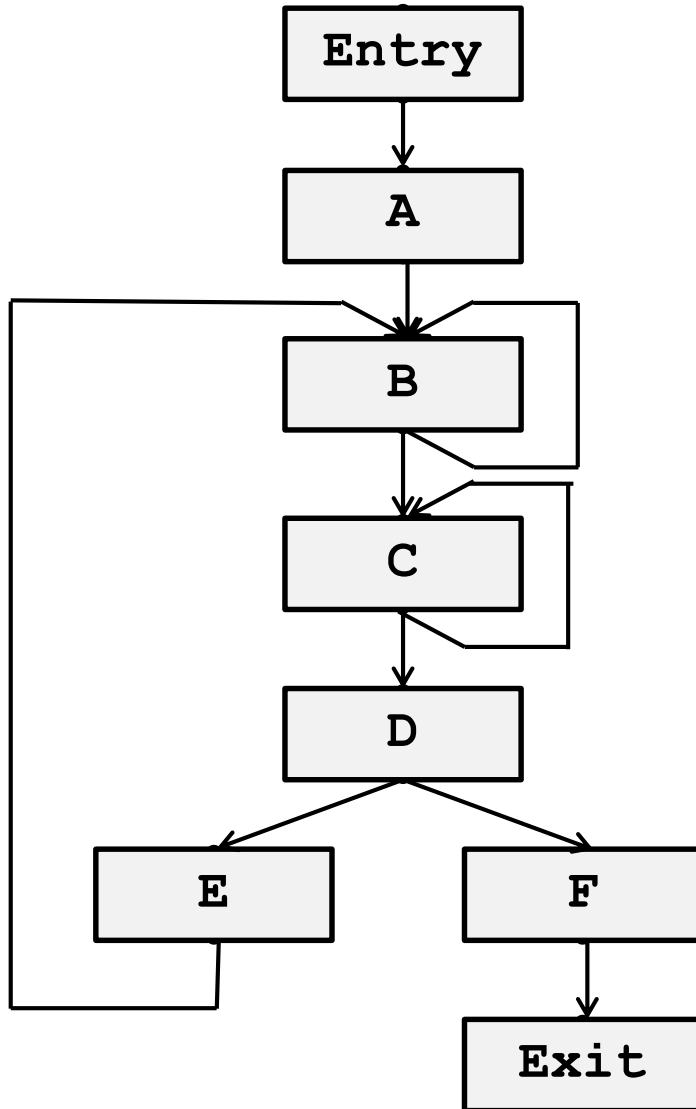


ГПУ

2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

◇ После нумерации вершин ГПУ из примера 1.5.5 примет вид



2.1 Нумерация вершин ГПУ

2.1.2 Алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ

Алгоритм

- ◇ **Вход:** ГПУ $G = \langle N, E \rangle$ с корнем $Entry \in N$
- ◇ **Выход:** глубинное остовное дерево графа G ($T_{DFS}(G)$) и нумерация узлов графа G , соответствующая упорядочению в глубину.
- ◇ **Метод:** все узлы $n \in N$ помечаются как nv (*not visited*)
вызывается рекурсивная процедура **DFST (n0)**
когда процедура **DFST** завершится, будут построены:
 - ◇ массив номеров узлов dfn
 - ◇ множество T ребер глубинного остовного дерева $T_{DFS}(G)$

2.1 Нумерация вершин ГПУ

2.1.2 Алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ

Рекурсивный алгоритм построения *DFST*

◇ Функция **main()** :

```
main() {  
    T =  $\emptyset$ ;  
    for all n  $\in$  N n.vst = nv;  
    c = |N|;  
    DFST(n0);  
}
```

Каждая вершина ГПУ представлена структурой

```
struct n {number, vst},
```

где *number* – номер вершины, а *vst* имеет 2 значения:

v (вершина была посещена) и

nv (вершина не была посещена)

2.1 Нумерация вершин ГПУ

2.1.2 Алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ

◇ Функция **DFST()**:

```
void DFST(n) {  
    Отмечаем n как v;  
    for all s ∈ Succ(n)  
        if (s.vst == nv) {  
            T ∪= {n→s};  
            DFST(s);  
        }  
    dfn[c] = n - c;  
    c--;  
}
```

◇ **Замечание.** Для каждой вершины $n \in N$ нетрудно построить множество $Succ(n)$, содержащее все вершины $s \in N$, в которые входят дуги, выходящие из вершины n

2.2 Анализ потока данных

2.2.1 Поток данных

- ◇ *Состояние программы* – множество значений всех переменных программы, включая переменные в кадрах стека времени выполнения, находящихся ниже текущей вершины стека
- ◇ *Точки программы* $(\dots, p_j, p_{j+1}, p_{j+2}, \dots)$ расположены между ее инструкциями $(\dots, I_j, I_{j+1}, \dots)$

$\dots \bullet \overset{p_j}{\bullet} \underset{I_j}{\bullet} \bullet \overset{p_{j+1}}{\bullet} \underset{I_{j+1}}{\bullet} \bullet \overset{p_{j+2}}{\bullet} \dots$

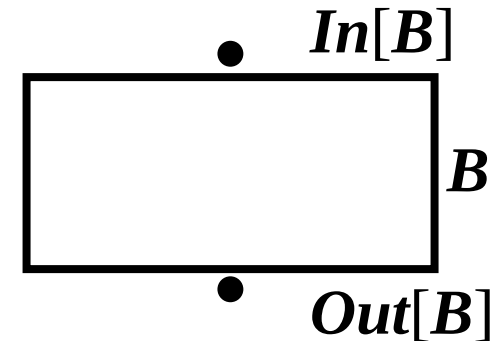
- ◇ Инструкция программы I_j описывается парой состояний:
 - ◇ состоянием в *точке программы* p_j *перед* инструкцией I_j
 - ◇ состоянием в *точке программы* p_{j+1} *после* инструкции I_j .

2.2 Анализ потока данных

2.2.1 Поток данных

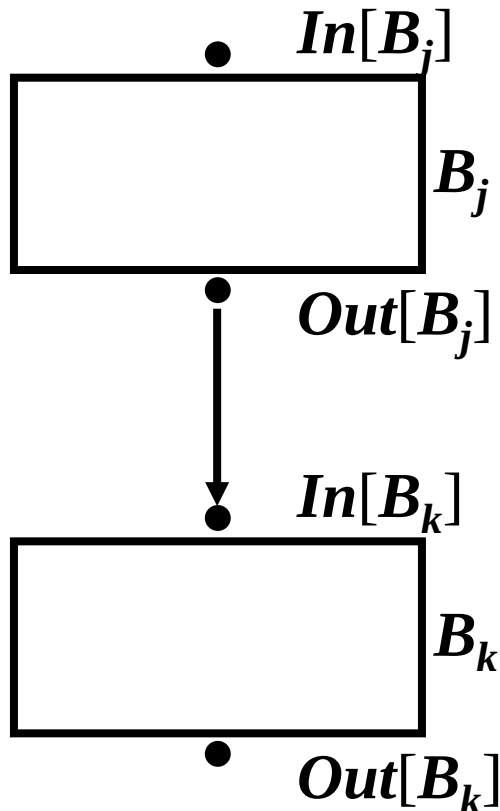
Базовый блок B описывается парой состояний:

- ◇ состоянием $In[B]$ в точке входа в B (перед первой инструкцией),
- ◇ состоянием $Out[B]$ в точке выхода из B (после последней инструкции)



С дугой от блока B_j к блоку B_k связаны две точки программы :

- ◇ точка выхода из блока B_j (ей соответствует состояние $Out[B_j]$)
- ◇ точка входа в блок B_k (ей соответствует состояние $In[B_k]$)



При рассмотрении потока данных между базовыми блоками нельзя отождествлять точку выхода из базового блока и точку входа в следующий за ним базовый блок, так как последняя может следовать за точками выхода из нескольких базовых блоков (есть же **goto**)

2.2 Анализ потока данных

2.2.3 Передаточные функции инструкций

- ◇ Соотношение f_{I_j} между значениями данных до и после инструкции I_j называется *передаточной функцией* инструкции I_j .
- ◇ Передаточные функции работают **в прямом и обратном** направлениях:
 - ◇ **В задаче прямого обхода:** $Out[I_j] = f_{I_j}(In[I_j])$
 - ◇ **В задаче обратного обхода:** $In[I_j] = f_{I_j}^b(Out[I_j])$
- ◇ Если I_j и I_{j+1} – *последовательные* инструкции блока B , то
 - ◇ **В задаче прямого обхода** $In[I_{j+1}] = Out[I_j]$
 - ◇ **В задаче обратного обхода** $Out[I_{j-1}] = In[I_j]$

2.2 Анализ потока данных

2.2.3 Передаточные функции инструкций

- ◇ Соотношение f_{I_j} между значениями данных до и после инструкции I_j называется *передаточной функцией* инструкции I_j .
- ◇ Передаточные функции работают **в прямом и обратном** направлениях:
 - ◇ **В задаче прямого обхода:**
 - ◇ **В задаче обратного обхода:** $Out[I_j] = f_{I_j}(In[I_j])$
 $In[I_j] = f_{I_j}^b(Out[I_j])$
- ◇ Если I_j и I_{j+1} являются инструкциями системы P , то
 - ◇ **В задаче прямого обхода** f и f^b – две разные функции
 - ◇ **В задаче обратного обхода** $In[I_{j+1}] = Out[I_j]$
 $Out[I_{j-1}] = In[I_j]$

2.2 Анализ потока данных

2.2.3 Передаточные функции базовых блоков

- ◇ Рассмотрим базовый блок

$$B = \langle P, Input, Output \rangle,$$

где $P = I_1, \dots, I_n$ (в указанном порядке)

- ◇ По определению

$$In[B] = In[I_1], Out[B] = Out[I_n].$$

- ◇ Передаточная функция f_B блока B по определению равна композиции передаточных функций его инструкций $I_1, \dots, I_n$

$$f_B(x) = f_{I_n}(f_{I_{n-1}}(\dots f_{I_1}(x)\dots)) = (f_{I_1} \circ f_{I_2} \circ \dots \circ f_{I_n})(x)$$

или

$$f_B = f_{I_1} \circ f_{I_2} \circ \dots \circ f_{I_n}$$

2.2 Анализ потока данных

2.2.4 Передаточные функции базовых блоков



При прямом обходе:

Соотношение между потоком данных при выходе из блока B и потоком данных при входе в него имеет вид

$$Out[B] = f_B(In[B])$$



При обратном обходе:

Соотношение между потоком данных при входе в блок B и потоком данных при выходе из него имеет вид

$$In[B] = f_B^b(Out[B])$$

2.3 Достигающие определения

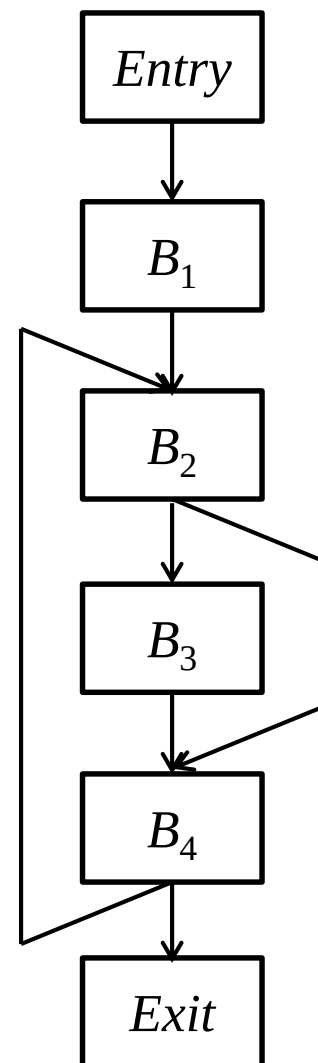
2.3.1 Терминология

- ◇ *Определением переменной x* называется инструкция, которая присваивает значение переменной x .
 - ◇ *Использованием переменной x* является инструкция, одним из операндов которой является переменная x .
 - ◇ Каждое определение переменной x *убивает* все другие ее определения.
 - ◇ Определение d *достигает* точки p , если существует путь от точки, непосредственно следующей за d , к точке p , такой, что вдоль этого пути d остается живым.
- ◇ **Замечание.** Во время анализа достигающих определений рассматриваются не переменные, а их определения, причем каждая переменная может иметь несколько определений

2.3 Достигающие определения

2.3.2 Пример

B_1 $i \leftarrow -, m, 1$ $j \leftarrow n$ $a \leftarrow u1$	B_2 $i \leftarrow +, i, 1$ $j \leftarrow -, j, 1$
B_3 $a \leftarrow u2$	B_4 $i \leftarrow u3$



- ◇ Начало блока B_2 **достигается** определениями:
 - ◇ $(i, B_1), (j, B_1), (a, B_1),$
 - ◇ (j, B_2) (других определений j на пути от (j, B_2) до начала блока B_2 нет)
 - ◇ (a, B_3)
 - ◇ (i, B_4)
- ◇ Начало блока B_2 **не достигается** определением:
 (i, B_2) , так как его убивает определение (i, B_4)
- ◇ Определение (j, B_1) **не достигает** блоков B_3 и B_4 , так как его убивает определение (j, B_2)

2.3 Достигающие определения

2.3.3 Передаточные функции для достигающих определений

◇ Рассмотрим инструкцию I

$$d: \mathbf{u} = \mathbf{v} + \mathbf{w}$$

расположенную между точками p_1 и p_2 программы.

- ◇ Пусть X – множество определений, достигающих точки p_1
 gen_I – множество определений, порождаемых
инструкцией I
 $kill_I$ – множество определений, убиваемых инструкцией I
 u – множество определений, достигающих точки p_2
- ◇ $gen_I = \{d\}$
- ◇ для определения $kill_I$ нужно иметь **все другие
определения $\mathbf{u}$** , т.е. несколько базовых блоков,
а иногда и всю процедуру.

2.3 Достигающие определения

2.3.3 Передаточные функции для достигающих определений

- ◇ Рассмотрим инструкцию I

$$d: \mathbf{u} = \mathbf{v} + \mathbf{w}$$

расположенную между точками p_1 и p_2 программы.

- ◇ По определению передаточной функции

$$y = f_I(x)$$

- ◇ Инструкция I сначала убивает все предыдущие определения $\mathbf{u}$, а потом порождает d – новое определение $\mathbf{u}$.
Следовательно

$$y = gen_I \cup (x - kill_I)$$

- ◇ Следовательно, передаточная функция f_I инструкции I может быть записана в виде:

$$f_I(x) = gen_I \cup (x - kill_I)$$

2.3 Достигающие определения

2.3.4. Передаточные функции вида *gen-kill*

◇ **Определение.**

Передаточные функции, определяемые соотношением

$$f(x) = gen \cup (x - kill)$$

будем называть передаточными функциями вида *gen-kill*.

◇ **Утверждение 1.**

Композиция двух функций вида *gen-kill* является функцией вида *gen-kill*.

$$\begin{aligned}(f_1 \circ f_2)(x) &= f_2(f_1(x)) = \\ &= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2) = \\ &= gen_2 \cup (gen_1 - kill_2) \cup (x - kill_1 - kill_2) \\ (f_1 \circ f_2)(x) &= gen_{f_1 \circ f_2} \cup (x - kill_{f_1 \circ f_2}),\end{aligned}$$

где

$$\begin{aligned}gen_{f_1 \circ f_2} &= gen_2 \cup (gen_1 - kill_2) \\ kill_{f_1 \circ f_2} &= kill_1 \cup kill_2\end{aligned}$$

2.3 Достигающие определения

2.3.4. Передаточные функции вида *gen-kill*

◇ **Утверждение 2.**

Пусть базовый блок B содержит n инструкций, каждая из которых имеет передаточную функцию $f_i(x) = gen_i \cup (x - kill_i)$
 $i = 1, 2, \dots, n$. Тогда передаточная функция для базового блока B может быть записана как

$$f_B(x) = gen_B \cup (x - kill_B) \quad ,$$

где

$$kill_B = kill_1 \cup kill_2 \cup \dots \cup kill_n$$

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \dots \\ \cup (gen_1 - kill_2 - kill_3 - \dots - kill_n)$$

2.3 Достигающие определения

2.3.5. Передаточные функции вида *gen-kill*

- ◇ Если какая-либо переменная определяется в блоке B несколько раз, то в gen_B войдет только ее последнее определение, т.е.
только последнее определение переменной будет действительно вне блока.

2.3 Достигающие определения

2.3.6. Система уравнений

- ◇ Таким образом, для КАЖДОГО базового блока B_i можно выписать уравнение

$$Out[B_i] = f_B(In[B_i])$$

или в случае анализа достигающих определений

$$Out[B_i] = gen_B \cup (In[B_i] - kill_B)$$

- ◇ Если ГПУ содержит n базовых блоков, получится n уравнений относительно $2 \cdot n$ неизвестных $In[B_i]$ и $Out[B_i]$, $i = 1, 2, \dots, n$.
- ◇ Еще n уравнений получится с помощью сбора вкладов путей.

2.3 Достигающие определения

2.3.6 Сбор вкладов путей

- ◇ Определение достигает точки программы, тогда и только тогда, когда существует по крайней мере один путь, вдоль которого эта точка может быть достигнута.
- Этот путь должен пройти через какую-нибудь вершину из $Pred(B)$, причем если путь, проходящий через вершину $P \in Pred(B)$, не проходит ни через одну вершину, содержащую определение какой-либо переменной из B , то $Out(P) = \emptyset$. Следовательно

$$In[B] = \bigcup_{P \in Pred(B)} Out[P]$$

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

◇ Получается система уравнений

$$Out_{RD}[B_i] = gen_{B_i} \cup (In_{RD}[B_i] - kill_{B_i})$$

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} Out_{RD}[P]$$

($i = 1, 2, \dots, n$).

(RD - Reaching definitions)

◇ $In_{RD}[B]$ – множество переменных, определенных на входе в блок B

◇ $Out_{RD}[B]$ – множество переменных, определенных на выходе из блока B

В дальнейшем индекс RD будет опускаться

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

◇ Полученную систему уравнений

$$Out_{RD}[B_i] = gen_{B_i} \cup (In_{RD}[B_i] - kill_{B_i})$$

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} Out_{RD}[P]$$

можно упростить, произведя очевидную подстановку, в результате чего система уравнений примет вид:

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} (gen_P \cup (In_{RD}[P] - kill_P))$$

$(i = 1, 2, \dots, n).$

или, если вспомнить, что было обещано опускать индекс RD ,

$$In[B_i] = \bigcup_{P \in Pred(B_i)} (gen_P \cup (In[P] - kill_P))$$

$(i = 1, 2, \dots, n).$

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

- ◇ Систему уравнений

$$In[B_i] = \bigcup_{P \in Pred(B_i)} (gen_P \cup (In[P] - kill_P))$$

$$(i = 1, 2, \dots, n)$$

будем решать методом итераций.

- ◇ При этом для одного или более блоков B_i множество $Pred(B_i)$ может содержать вершину $Entry$. В этих случаях будет использоваться «граничное условие» $In[Entry] = \emptyset$
- ◇ В качестве начальных итераций $In[B_i]$ возьмем пустые множества:
 $(In[B_i])^0 = \emptyset$

Объяснения будут позже (в следующий понедельник).

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

◇ Алгоритм «Достигающие определения»

- ◇ **Вход:** ГПУ (N, E) , в котором для каждого базового блока $B_i \in N$ вычислены множества $kill_{B_i}$ и gen_{B_i}
- ◇ **Выход:** множества $In[B_i]$, $(i = 1, 2, \dots, n)$ достигающих определений **НА ВХОДЕ** в каждый базовый блок B_i графа потока управления

- ◇ **Метод:** Используется **метод итераций** с начальной итерацией

$$(In[B_i])^0 = \emptyset$$

На всех итерациях r $(In[Entry])^r = \emptyset$ (граничное условие)

Итерации продолжаются до тех пор, пока все множества $(In[B_i])^r$

(r – номер итерации) не перестанут изменяться.

- ◇ **Замечание.** Некоторые из множеств $(In[B_i])^r$ могут перестать изменяться гораздо раньше последней итерации.

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

$In[Entry] = \emptyset;$

$change = true;$

for (каждый базовый блок B , отличный от $Entry$)

$In[B] = \emptyset;$

/* основной цикл*/

while ($change$) do {

$change = false;$

for (каждый базовый блок B , отличный от $Entry$) {

/* вычисление новых значений $In[B]$ и переменной $change$ */

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

if ($InNew[B] \neq In[B]$) {

$In[B] = InNew[B];$

$change = true;$

}

}

}

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

```
In[Entry] =  $\emptyset$ ;  
change = true;  
for (каждый базовый блок B, отличный от Entry)  
    In[B] =  $\emptyset$ ;  
/* основной цикл*/  
    while (change) do {
```

Алгоритм 2.3.7 не учитывает **замечания**: для некоторых значений i окончательный результат может быть получен намного раньше, чем будет выполнена последняя итерация. А это означает, что если на самом деле так получится, соответствующие значения будут много раз напрасно вычисляться, сравниваться с предыдущей итерацией и отвергаться – много лишней работы.

Получится, что алгоритм оптимизации сам не оптимален

```
    }  
}  
}
```

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

```
In[Entry] =  $\emptyset$ ;  
change = true;  
for (каждый базовый блок B, отличный от Entry)  
    In[B] =  $\emptyset$ ;  
/* основной цикл*/  
    while (change) do {  
        change = false;
```

Поэтому имеет смысл модифицировать алгоритм 2.3.7, введя понятие **рабочего множества**. Рабочее множество **WorkList**; представляет собой очередь, в которую помещаются только те базовые блоки, которые требуют дальнейшей обработки.

```
        for ( $P \in \text{Pred}(B)$ )  
            if (InNew[B]  $\neq$  In[B]) {  
                In[B] = InNew[B];  
                change = true;  
            }  
        }  
    }  
}
```

2.3 Достигающие определения

2.3.8 Модифицированный итеративный алгоритм для вычисления достигающих определений

In[Entry] = $\emptyset$;

WorkList = $\emptyset$;

for (каждый базовый блок В, отличный от Entry) {

 поместить В в WorkList;

 In[B] = $\emptyset$; /* Каждому In[B] присваивается значение его нулевой итерации */

};

do { /* основной цикл*/

 Выбрать из очереди WorkList очередной блок В

 Вычислить InNew[B], используя уравнение

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

/*При вычислении InNew[B] множества In[P],
где $P \in Pred(B)$, могут иметь значения либо текущей, либо следующей итерации

if (InNew[B] $\neq$ In[B]) {

 In[B] = InNew[B];

 Поместить In[B] в конец очереди WorkList

}

} while |WorkList|>0;

2.3 Достигающие определения

2.3.8 Модифицированный итеративный алгоритм для вычисления достигающих определений

```
In[Entry] =  $\emptyset$ ;  
WorkList =  $\emptyset$ ;  
for (каждый базовый блок В, отличный от Entry) {  
    поместить В в WorkList;  
    In[B] =  $\emptyset$ ; /* Каждому In[B] присваивается значение его  
                  нулевой итерации */
```

Ценность алгоритма в том, что каждый базовый блок рассматривается не на каждой итерации, а столько раз, сколько он попадает в WorkList

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

```
/*При вычислении InNew[B] множества In[P],  
   где P ∈ Pred(B), могут иметь значения либо текущей,  
   либо следующей итерации
```

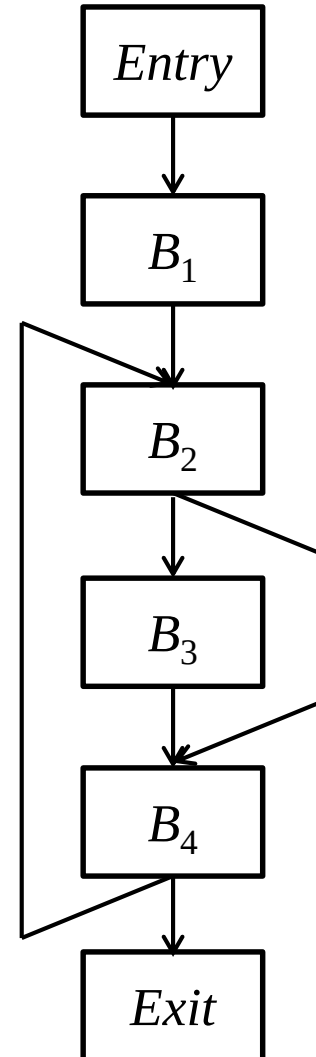
```
if (InNew[B] ≠ In[B]) {  
    In[B] = InNew[B];  
    Поместить In[B] в конец очереди WorkList  
}
```

```
} while |WorkList| > 0;
```

2.3 Достигающие определения

2.3.8 Пример

B_1 $i \leftarrow -, m, 1$ $j \leftarrow n$ $a \leftarrow u1$	B_2 $i \leftarrow +, i, 1$ $j \leftarrow -, j, 1$
B_3 $a \leftarrow u2$	B_4 $i \leftarrow u3$



2.3 Достигающие определения

2.3.8 Пример

В простой программе справа 7 определений. Требуется определить, какие определения достигают входов в ее 5 базовых блоков **B1**, **B2**, **B3**, **B4** и **Exit**

B_1 $d_1 : i \leftarrow -, m, 1$ $d_2 : j \leftarrow n$ $d_3 : a \leftarrow u1$	B_2 $d_4 : i \leftarrow +, i, 1$ $d_5 : j \leftarrow -, j, 1$
B_3 $d_6 : a \leftarrow u2$	B_4 $d_7 : i \leftarrow u3$

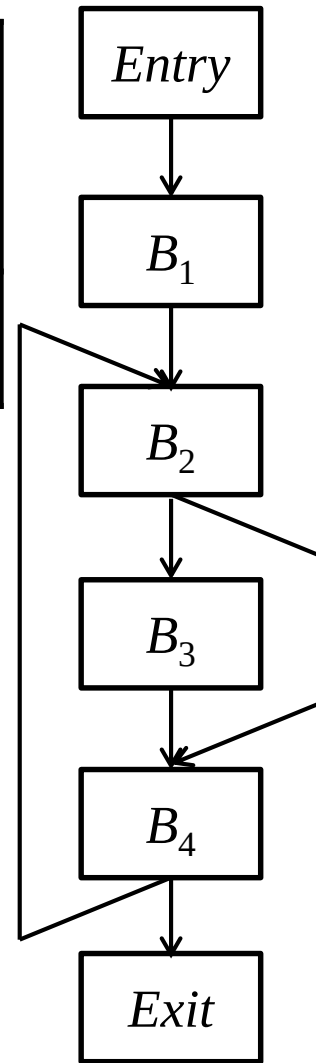
$d_1 = (i, B_1)$ – определение переменной i в блоке B_1

$d_4 = (i, B_2)$ – определение переменной i в блоке B_2

$d_7 = (i, B_4)$ – определение переменной i в блоке B_2

Вычислим множества gen и $kill$ для каждого базового блока

Для блока B_1 $gen_{B_1} = \{d_1, d_2, d_3\}$ $kill_{B_1} = \{d_4, d_5, d_6, d_7\}$



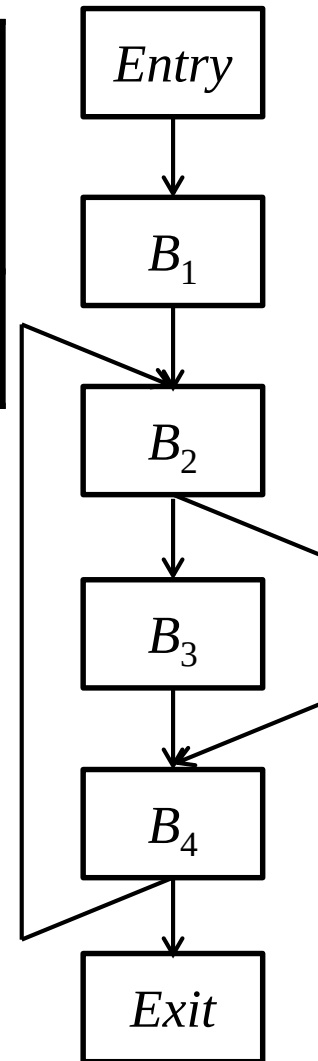
2.3 Достигающие определения

2.3.8 Пример

B_1 $d_1: i \leftarrow -, m, 1$ $d_2: j \leftarrow n$ $d_3: a \leftarrow u1$	B_2 $d_4: i \leftarrow +, i, 1$ $d_5: j \leftarrow -, j, 1$
B_3 $d_6: a \leftarrow u2$	B_4 $d_7: i \leftarrow u3$

Результаты вычисления множеств *gen* и *kill* для базовых блоков сведены в таблицу

B	gen_B	$kill_B$
B_1	$\{(i, B_1), (j, B_1), (a, B_1)\}$ $= (1110000)$	$\{(i, B_2), (j, B_2), (a, B_3), (i, B_4)\}$ $= (0001111)$
B_2	$\{(i, B_2), (j, B_2)\}$ $= (0001100)$	$\{(i, B_1), (j, B_1), (i, B_4)\}$ $= (1100001)$
B_3	$\{(a, B_3)\} = (0000010)$	$\{(a, B_1)\} = (0010000)$
B_4	$\{(i, B_4)\} = (0000001)$	$\{(i, B_1), (i, B_2)\} = (1001000)$
$Exit$	$\emptyset$	$\emptyset$



2.3 Достигающие определения

2.3.8 Пример

B_1 $d_1 : i \leftarrow -, m, 1$ $d_2 : j \leftarrow n$ $d_3 : a \leftarrow u1$	B_2 $d_4 : i \leftarrow +, i, 1$ $d_5 : j \leftarrow -, j, 1$
B_3 $d_6 : a \leftarrow u2$	B_4 $d_7 : i \leftarrow u3$

Множества удобно представлять битовыми векторами, длина которых равна мощности базового множества. В рассматриваемом примере длина векторов равна 7.

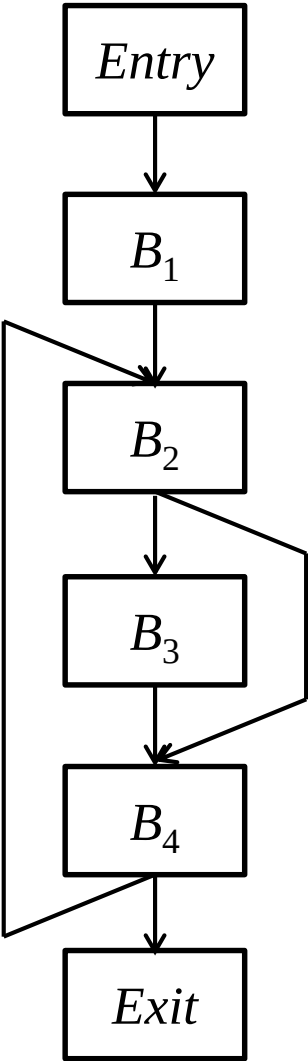
$$gen_{B_1} = (1110000) \qquad kill_{B_1} = (0001111)$$

d_1 d_2 d_3 d_4 d_5 d_6 d_7

(i, B_1) (j, B_1) (a, B_1) (i, B_2) (j, B_2) (a, B_3) (i, B_4)

Определения примера 2.3.8

B	gen_B	$kill_B$
B_1	(1110000)	(0001111)
B_2	(0001100)	(1100001)
B_3	(0000010)	(0010000)
B_4	(0000001)	(1001000)



Множества *gen* и *kill* базовых блоков примера 2.3.8

2.3 Достигающие определения

2.3.8 Пример

Нулевая итерация:

$$(In[B_i])^0 = \emptyset = (00000000), i = 1, 2, 3, 4.$$

$$(In[Exit])^0 = \emptyset = (00000000)$$

Первая итерация:

Вычисляем $(In[B_i])^1$ ($i = 1, 2, 3, 4$) и $(In[Exit])^1$ по формуле

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

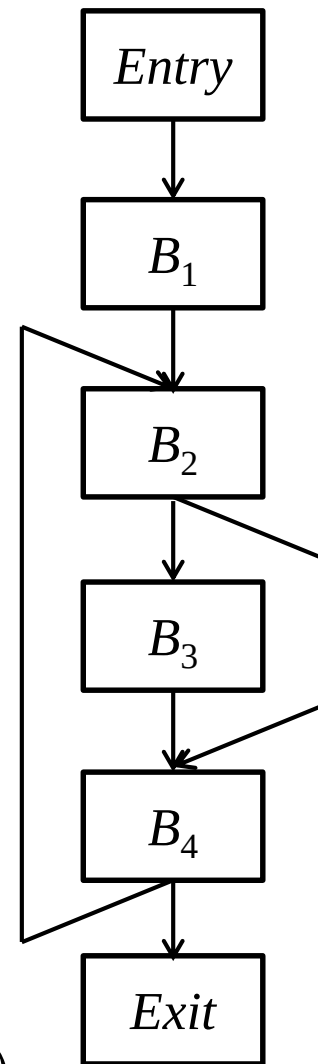
используя значения $(In[B_i])$, известные к моменту вычисления

Вычисление $(In[B_1])^1$. $(In[B_1])^1 = \emptyset$ $(In[B_1])$ не попадает в WorkList и больше не будет вычисляться

Вычисление $(In[B_2])^1$.

$$\begin{aligned} (In[B_2])^1 &= (gen_{B_1} \cup ((In[B_1])^1 - kill_{B_1})) \cup (gen_{B_4} \cup ((In[B_4])^0 - kill_{B_4})) = \\ &= ((1110000) \cup (\emptyset - (0001111))) \cup ((0000001) \cup (\emptyset - (1001000))) = \\ &= (1110000) \cup (0000001) = (1110001) \end{aligned}$$

B	$Pred(B)$
B_1	$Entry$
B_2	$\{B_1, B_4\}$
B_3	B_2
B_4	$\{B_2, B_3\}$
$Exit$	B_4



2.3 Достигающие определения

2.3.8 Пример

B	gen_B	$kill_B$	$Pred(B)$
B_1	(1110000)	(0001111)	$Entry$
B_2	(0001100)	(1100001)	$\{B_1, B_4\}$
B_3	(0000010)	(0010000)	B_2
B_4	(0000001)	(1001000)	$\{B_2, B_3\}$

Первая итерация:

Вычисление $(In[B_3])^1$. $(In[B_3])^1 = gen_{B_2} \cup ((In[B_2])^1 - kill_{B_2}) =$
 $= (0001100) \cup ((1110001) - (1100001)) =$
 $= (0011100)$

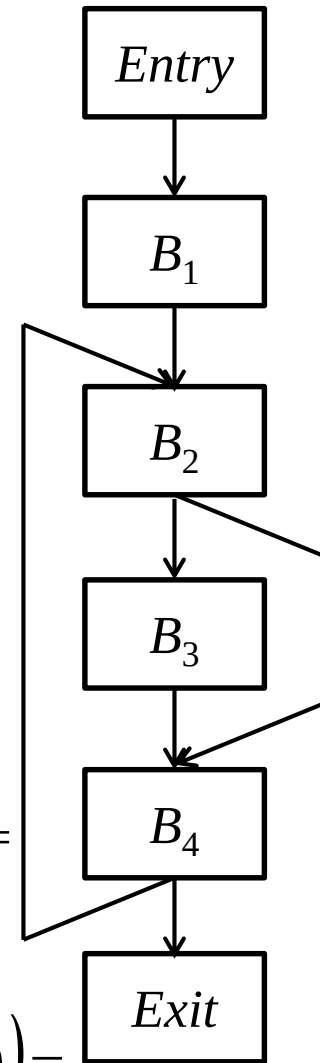
Вычисление $(In[B_4])^1$.

$$(In[B_4])^1 = (gen_{B_2} \cup ((In[B_2])^1 - kill_{B_2})) \cup (gen_{B_3} \cup ((In[B_3])^1 - kill_{B_3})) =$$

$$= ((0001100) \cup ((1110001) - (1100001))) \cup$$

$$\cup ((0000010) \cup ((0011100) - (0010000))) =$$

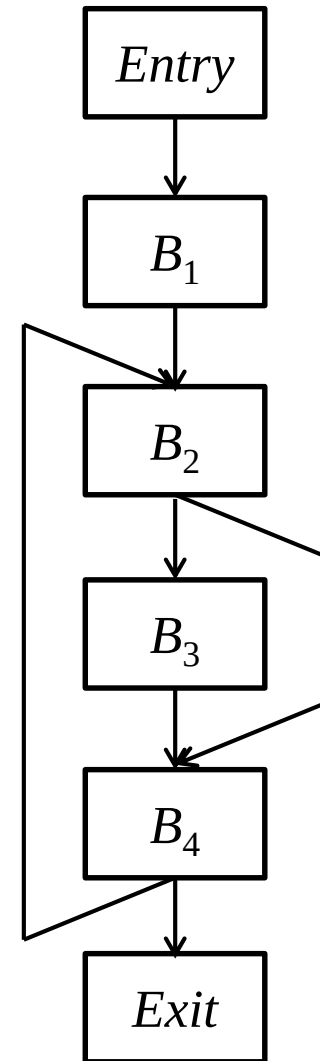
$$= (0001110)$$



2.3 Достигающие определения

2.3.8 Пример

B	gen_B	$kill_B$	$Pred(B)$
B_1	(1110000)	(0001111)	$Entry$
B_2	(0001100)	(1100001)	$\{B_1, B_4\}$
B_3	(0000010)	(0010000)	B_2
B_4	(0000001)	(1001000)	$\{B_2, B_3\}$



Первая итерация:

Вычисление $(In[Exit])^1$.

$$\begin{aligned}
 (In[Exit])^1 &= gen_{B_4} \cup ((In[B_4])^1 - kill_{B_4}) = \\
 &= (0000001) \cup ((0001110) - (1001000)) = \\
 &= (0010111)
 \end{aligned}$$

2.3 Достигающие определения

2.3.8 Пример

B	$(In[B])^1$	$(In[B])^2$	$(In[B])^3$
B_1	0000000	0000000	0000000
B_2	1110001	1110111	1110111
B_3	0011100	0011110	0011110
B_4	0011111	0011110	0011110
$Exit$	0010111	0010111	0010111

Вторая и третья итерации выполняются аналогично (их результаты внесены в таблицу). На этом процесс завершается, так как, как видно из таблицы, все $(In[B])^3$ совпадают с $(In[B])^2$, т.е. *WorkList* пуст.

2.3.9 Множества *Input* для базовых блоков

- ◇ Множество $Input[B]$ для базового блока B – это множество $In_{RD}[B]$, которое строится при исследовании достигающих определений

2.4 Живые переменные

2.4.1 Множества *Output* для базовых блоков

- ◇ Множества *Output* для базовых блоков строятся как результат анализа, позволяющего выявить *живые переменные*, т.е. переменные, используемые в базовых блоках, в которые управление попадает после выхода из исследуемого базового блока.
- ◇ Анализ похож на предыдущий, но ГПУ просматривается не с начала, а с конца: от *Exit* к *Entry*.

2.4.2. Определение

- ◇ Цель анализа – для определения переменной X в точке p программы выяснить, будет ли указанное значение X использоваться вдоль какого-нибудь пути, начинающегося в точке p .
 - ◇ Если да – переменная X *жива* (активна) в точке p ,
 - ◇ если нет – переменная X *мертва* (неактивна) в точке p .

2.4 Живые переменные

2.4.3 Уравнения потока данных

◇ $In_{LV}[B]$ – множество переменных, **живых** на входе в блок B

LV – Live Variables

$Out_{LV}[B]$ – множество переменных, **живых** на выходе из блока B .

◇ В чем проблема?

Пусть в блоке B используется переменная v .

Возможны 2 случая:

- 1) используется определение v в одном из блоков $B' \in Pred^*(B)$;
- 2) используется определение v в самом блоке B .

В первом случае говорят, что v жива на выходе из B'

Во втором случае говорят, что v мертва на выходе из B'

2.4 Живые переменные

2.4.3 Уравнения потока данных

◇ $In_{LV}[B]$ – множество переменных, **живых** на входе в блок B

LV – Live Variables

$Out_{LV}[B]$ – множество переменных, **живых** на выходе из блока B .

◇ def_B – множество переменных, определяемых в блоке B до их использования в этом блоке

(любая переменная из def_B **мертва** на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred_B$)

◇ use_B – множество переменных, используемых в блоке B до их определения в этом блоке

(любая переменная из use_B **жива** на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred_B$)

◇ **Замечание.** В анализе живых переменных рассматриваются не определения переменных, а сами переменные.

2.4 Живые переменные

2.4.3 Уравнения потока данных

Рассмотрим блок B

$m \leftarrow 1$
$k \leftarrow m + k$
$a \leftarrow a + b + m$

Речь идет о выходе из блоков из $Pred(B)$

Для этих блоков $m \in def$

Остальные переменные $\in use$

2.4 Живые переменные

2.4.3 Уравнения потока данных

B_1 $i \leftarrow -, m, 1$ $j \leftarrow n$ $a \leftarrow u1$	B_2 $i \leftarrow +, i, 1$ $j \leftarrow -, j, 1$
B_3 $a \leftarrow u2$	B_4 $i \leftarrow u3$

◇ Пример

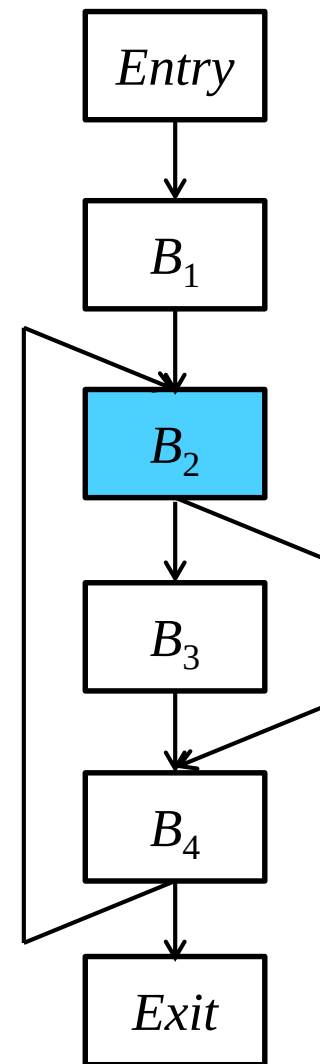
- (1) в блоке B_2 переменные i и j используются до их переопределения, следовательно,

$$use_{B_2} = \{i, j\} = (\mathbf{11000000})$$

- (2) в блоке B_2 определяются новые значения переменных i и j , так что

$$def_{B_2} = \{i, j\} = (\mathbf{11000000})$$

В программе 8 переменных: $i, j, a, m, n, u1, u2, u3$



2.4 Живые переменные

2.4.3 Уравнения потока данных

- ◇ Уравнения, связывающие *def* и *use* с неизвестными *In* и *Out*, определяются следующим образом:

$$In[B] = use_B \cup (Out[B] - def_B)$$

$$Out[B] = \bigcup_{S \in Succ(B)} In[S]$$

- ◇ К ним добавляется граничное условие

$$Out[Exit] = \emptyset$$

2.4 Живые переменные

2.4.3 Уравнения потока данных

- ◇ Уравнения, связывающие *def* и *use* с неизвестными *In* и *Out*, определяются следующим образом:

$$In[B] = use_B \cup (Out[B] - def_B)$$

$$Out[B] = \bigcup_{S \in Succ(B)} In[S]$$

- ◇ К ним добавляется граничное условие

$$In[Exit] = \emptyset$$

def_B – множество переменных, определяемых в блоке B до их использования в этом блоке (**Исключаем заведомо мертвые переменные**)

use_B – множество переменных, используемых в блоке B до их определения в этом блоке (**Добавляем новые живые переменные**)

2.4 Живые переменные

2.4.3 Уравнения потока данных

- ◇ Если значение In , определяемое первым уравнением подставить во второе уравнение, множества In будут исключены из системы уравнений и получится система уравнений, содержащая в качестве неизвестных только множества Out :

$$Out[B] = \bigcup_{S \in Succ(B)} (use_S \cup (Out[S] - def_S))$$

- ◇ К ним добавляется граничное условие

$$Out[Exit] = \emptyset$$

2.4 Живые переменные

2.4.4 Итеративный алгоритм анализа живых переменных

◇ Алгоритм «Живые переменные»

- ◇ **Вход:** ГПУ, в котором для каждого блока B вычислены множества def и use
- ◇ **Выход:** множества переменных, живых на выходе ($Out[B]$) каждого базового блока B .
- ◇ **Метод:** выполнить следующую программу:

2.4 Живые переменные

2.4.4 Итеративный алгоритм анализа живых переменных

◇ Алгоритм «Живые переменные»

◇ Метод: выполнить следующую программу:

```
{Out[Exit] =  $\emptyset$ ;  
WorkList =  $\emptyset$ ;  
for(каждый базовый блок B, отличный от Exit){  
    Out[B] =  $\emptyset$ ;  
    WorkList  $\cup$ = {B}  
}  
/*основной цикл*/  
do { Извлечь блок B из WorkList (исключив B);  
    OutNew[B]=  $\bigcup_{S \in \text{Succ}(B)} (\text{use}_S \cup (\text{Out}[S] - \text{def}_S))$   
    if (OutNew[B]  $\neq$  Out[B]){  
        Out[B] = OutNew[B] ;  
        WorkList  $\cup$ = {B}  
    }  
} while |WorkList|>0;  
}
```

2.4 Живые переменные

2.4.5 Множества *Output* для базовых блоков

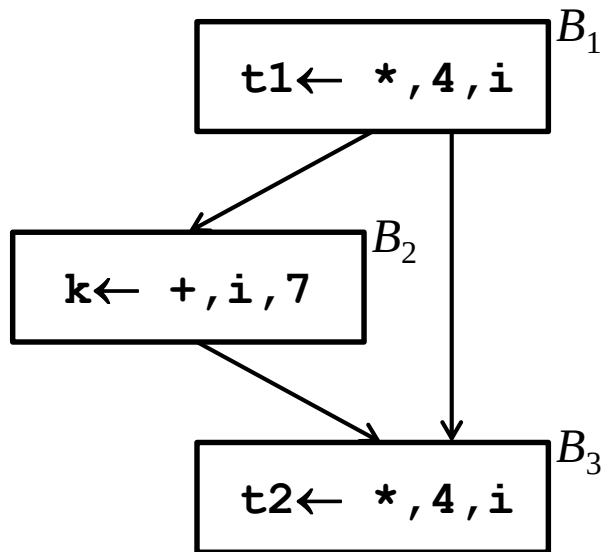
- ◇ Множество $Output[B]$ для базового блока B – это множество $Out_{LV}[B]$, которое строится при исследовании живых переменных

3. Анализ потока данных

3.1 Доступные выражения

3.1.1 Определение

- ◇ Выражение e *доступно* в точке p , если e вычисляется на любом пути от $Entry$ до p причем переменные, входящие в состав e не переопределяются между последним таким вычислением и точкой p .
- ◇ **Пример.** Пусть точка p – вход в блок B_3 .



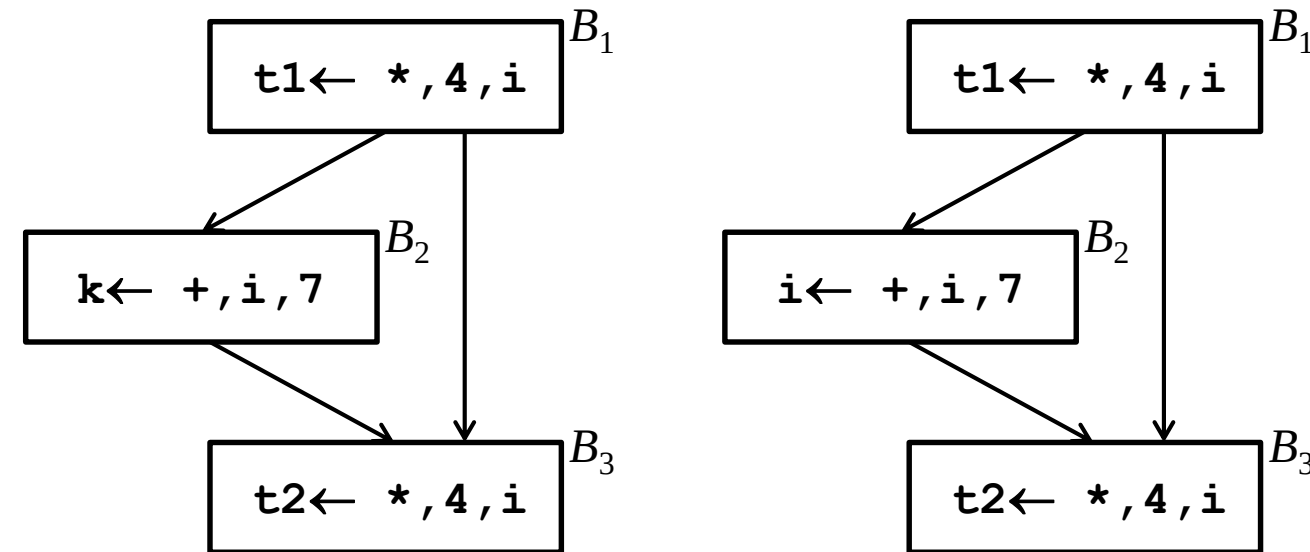
На рисунке присваивание *не убивает* выражение $*, 4, i$. Поэтому $t2 \leftarrow *, 4, i$ можно заменить на $t2 \leftarrow t1$

3.1 Доступные выражения

3.1.1 Определение

◇ Выражение e *доступно* в точке p , если e вычисляется на любом пути от $Entry$ до p причем переменные, входящие в состав e не переопределяются между последним таким вычислением и точкой p .

◇ **Пример.** Пусть точка p – вход в блок B_3 .



На рисунке присваивание *не убивает* выражение $\ast, 4, i$. Поэтому $t2 \leftarrow \ast, 4, i$ можно заменить на $t2 \leftarrow t1$

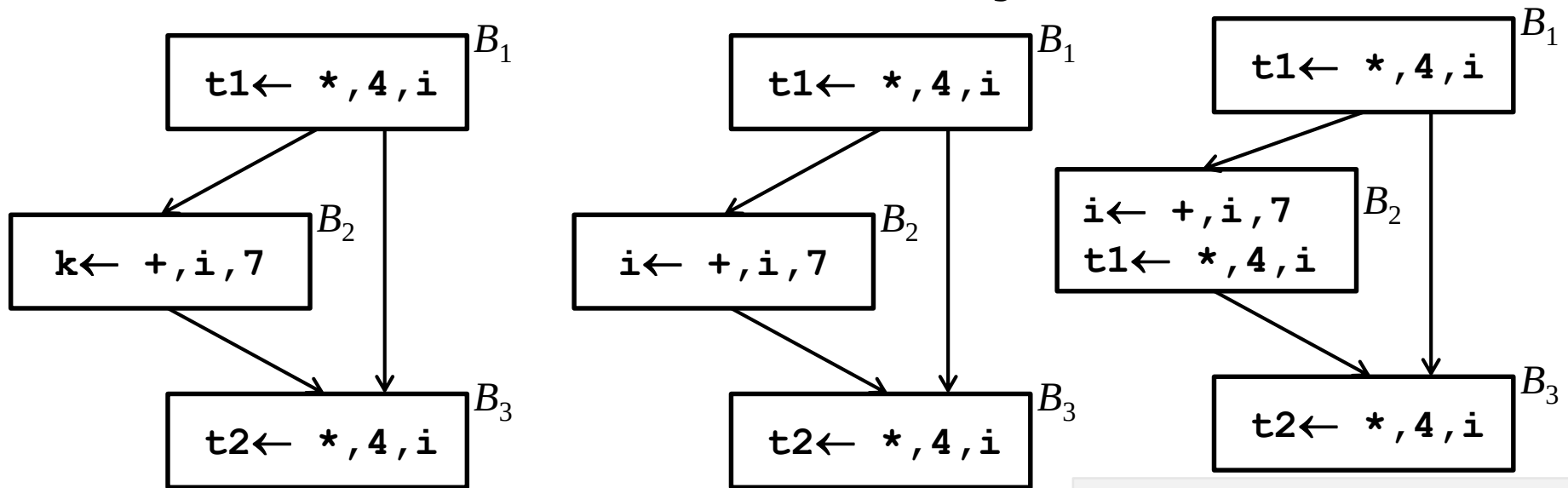
Присваивание *убивает* выражение $\ast, 4, i$. Замена $t2 \leftarrow \ast, 4, i$ на $t2 \leftarrow t1$ будет некорректной.

3.1 Доступные выражения

3.1.1 Определение

◇ Выражение e *доступно* в точке p , если e вычисляется на любом пути от $Entry$ до p причем переменные, входящие в состав e не переопределяются между последним таким вычислением и точкой p .

◇ **Пример.** Пусть точка p – вход в блок B_3 .



Присваивание *не убивает* выражение $*, 4, i$.
Замена $t2 \leftarrow *, 4, i$ на $t2 \leftarrow t1$ корректна.

Присваивание *убивает* выражение $*, 4, i$.
Замена $t2 \leftarrow *, 4, i$ на $t2 \leftarrow t1$ некорректна.

Перевычисление в B_2 выражения $*, 4, i$ «воскрешает» «убитое» выражение и обеспечивает **корректность** замены $t2 \leftarrow *, 4, i$ на $t2 \leftarrow t1$

3.1 Доступные выражения

3.1.1 Определение

- ◇ Для каждого базового блока B определим
 - ◇ множество e_kill_B выражений, *убиваемых* в блоке B
(пример – присваивание $i \leftarrow +, i, 7$ в блоке B_2 *убивает* выражение $\ast, 4, i$)
 - ◇ множество e_gen_B выражений, *порождаемых* в блоке B
(пример – присваивание $t1 \leftarrow \ast, 4, i$ в блоке B_2 *порождает* выражение $\ast, 4, i$).

3.1 Доступные выражения

3.1.2 Уравнения потока данных

◇ Для того, чтобы найти доступные выражения, можно использовать метод, напоминающий метод вычисления достигающих определений.

◇ U – множество всех выражений программы.

◇ $In[B]$ – множество выражений из U , доступных на входе в B ,

◇ Граничное условие:

$$In[Entry] = \emptyset$$

◇ Система уравнений:

$$Out[B] = e\_gen_B \cup (In[B] - e\_kill_B)$$

$$In[B] = \bigcap_{P \in Pred(B)} Out[P]$$

$$In[B_i] = \bigcap_{P \in Pred(B_i)} (e\_gen_P \cup (In[P] - e\_kill_P))$$

3.1 Доступные выражения

3.1.3 Итеративный алгоритм вычисления доступных выражений

- ♦ **Вход:** граф потока, в котором для каждого блока B вычислены множества e_gen_B и e_kill_B
- ♦ **Выход:** множества выражений, доступных на входе ($In[B]$) каждого базового блока B графа потока. .
- ♦ **Метод:** выполнить следующую программу

$In[Entry] = \emptyset;$

for (каждый базовый блок B , отличный от $Entry$) $In[B] = U;$

while (внесены изменения в Out)

for (каждый базовый блок B , отличный от $Entry$) {

$$In[B] = \bigcap_{P \in Pred(B)} (e\_gen_B \cup (In[B] - e\_kill_B))$$

}

3.1 Доступные выражения

3.1.3 Итеративный алгоритм вычисления доступных выражений

♦ **Метод:** выполнить следующую программу

```
In[Entry] =  $\emptyset$ ;  
WorkList =  $\emptyset$ ;  
for (каждый базовый блок В, отличный от Entry) {  
    поместить В в WorkList;  
    In[B] = U; /* Каждому In[B] присваивается значение  
                его нулевой итерации */  
};  
do {                               /* основной цикл*/  
    Выбрать из очереди WorkList очередной блок В  
    Вычислить InNew[B], используя уравнение  

$$InNew[B] = \bigcap_{P \in Pred(B)} (e\_gen_B \cup (In[B] - e\_kill_B))$$
  
    if (InNew[B]  $\neq$  In[B]) {  
        In[B] = InNew[B];  
        Поместить In[B] в конец очереди WorkList  
    }  
} while |WorkList|>0;
```

3.2 Полурешетки

3.2.1 Анализ потока данных

- ◇ При анализе потока данных рассматриваются множества переменных для описания состояния и такие операции как *объединение* ($\cup$) и *пересечение* ($\cap$) множеств.

3.2 Полурешетки

3.2.1 Анализ потока данных

- ◇ При анализе потока данных рассматриваются множества переменных для описания состояния и такие операции как *объединение* ($\cup$) и *пересечение* ($\cap$) множеств.
- ◇ Свойства операций $\cup$ и $\cap$:

$$A \cup A = A$$

$$A \cup B = B \cup A$$

$$A \cup (B \cup C) = (A \cup B) \cup C$$

$$\text{Если } A \cup B = A, \text{ то } B \subseteq A$$

$$A \cap A = A$$

$$A \cap B = B \cap A$$

$$A \cap (B \cap C) = (A \cap B) \cap C$$

$$\text{Если } A \cap B = A, \text{ то } B \supseteq A$$

(идемпотентность)

(коммутативность)

(ассоциативность)

отношение частичного
порядка, связанное с
операцией

3.2 Полурешетки

3.2.1 Анализ потока данных

◇ При анализе потока данных рассматриваются множества переменных для описания состояния и такие операции как *объединение* ($\cup$) и *пересечение* ($\cap$) множеств.

◇ Свойства операций $\cup$ и $\cap$:

$A \cup A = A$	$A \cap A = A$	(идемпотентность)
$A \cup B = B \cup A$	$A \cap B = B \cap A$	(коммутативность)
$A \cup (B \cap C) = (A \cup B) \cap C$	$A \cap (B \cup C) = (A \cap B) \cup C$	(ассоциативность)
Если $A \cup B = A$, то $B \subseteq A$	Если $A \cap B = A$, то $B \supseteq A$	отношение частичного порядка, связанное с операцией

◇ Если множество U содержит все элементы, то любое множество $A \subseteq U$ и $A \cup U = U \cup A = U$

◇ Для пересечения ($\cap$) роль U играет $\emptyset$: любое множество $A \supseteq \emptyset$ и $A \cap \emptyset = \emptyset \cap A = \emptyset$

3.2 Полурешетки

3.2.2 Определение полурешетки

- ◇ *Полурешетка* – это абстрактная алгебраическая структура, над элементами которой определена абстрактная операция $\wedge$ (мы будем называть ее «**сбор**»), обладающая свойствами операций $\cup$ и $\cap$.
- ◇ **Определение.** *Полурешетка* представляет собой множество L , на котором определена бинарная операция «сбор» $\wedge$, такая, что для всех x, y и $z \in L$:

$$x \wedge x = x \quad (\text{идемпотентность})$$

$$x \wedge y = y \wedge x \quad (\text{коммутативность})$$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z \quad (\text{ассоциативность})$$

Полурешетка имеет *верхний элемент* (или *верх*) $T \in L$ такой, что для всех $x \in L$ выполняется $T \wedge x = x$

3.2 Полурешетки

3.2.3 Полурешеточное отношение частичного порядка $\leq$

- ◇ Для всех пар $x, y \in L$ определим отношение $\leq$:
 $x \leq y$ тогда и только тогда, когда $x \wedge y = x$
- ◇ Отношение $\leq$ является **отношением частичного порядка**.
 - (1) *Рефлексивность* $\leq$ следует из идемпотентности $\wedge$:
$$x \leq x \Leftrightarrow x \wedge x = x$$
 - (2) *Антисимметричность* $\leq$ следует из коммутативности $\wedge$:
пусть $x \leq y$ и $y \leq x$; тогда $x = x \wedge y = y \wedge x = y$
 - (3) *Транзитивность* $\leq$ следует из ассоциативности $\wedge$:
пусть $x \leq y$ и $y \leq z$; тогда по определению $\leq$
 $x \wedge y = x$ и $y \wedge z = y$;
$$(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) = (x \wedge y) = x,$$
$$(x \wedge z) = x \Leftrightarrow x \leq z.$$

3.2 Полурешетки

3.2.4 Наибольшая нижняя граница

◇ **Определение.** Пусть $\langle L, \leq \rangle$ – частично упорядоченное множество (в частности, полурешетка).

Наибольшей нижней границей $\inf(x, y)$ элементов x и $y \in L$ называется элемент $g \in L$, такой, что $g \leq x$; $g \leq y$; и если $z \in L$, такой, что $z \leq x$ и $z \leq y$, то $z \leq g$.

◇ **Утверждение.** Если x и $y \in L$, где $\langle L, \wedge \rangle$ – полурешетка, то $\inf(x, y) = x \wedge y$.

Доказательство. Пусть $g = x \wedge y$. Тогда

$$\begin{aligned} g \wedge x &= ((x \wedge y) \wedge x) = (x \wedge (y \wedge x)) = (x \wedge (x \wedge y)) = \\ &= ((x \wedge x) \wedge y) = (x \wedge y) = g, \end{aligned}$$

откуда следует $g \leq x$.

Точно таким же образом доказывается, что $g \leq y$.

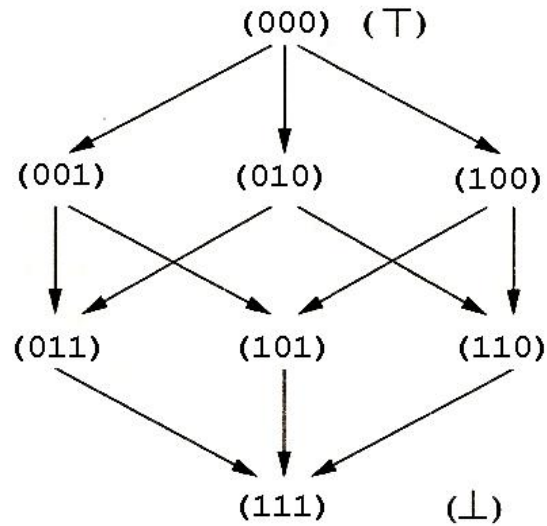
Пусть $z \in L$, $z \leq x$ и $z \leq y$. Докажем, что $z \leq g$. Имеем:

$$\begin{aligned} (z \wedge g) &= (z \wedge (x \wedge y)) = ((z \wedge x) \wedge y); \quad z \leq x \Rightarrow (z \wedge x) = z \\ &\Rightarrow (z \wedge g) = (z \wedge y) = z. \Rightarrow (z \wedge g) = z \Rightarrow z \leq g. \end{aligned}$$

3.2 Полурешетки

3.2.5. Диаграммы полурешеток

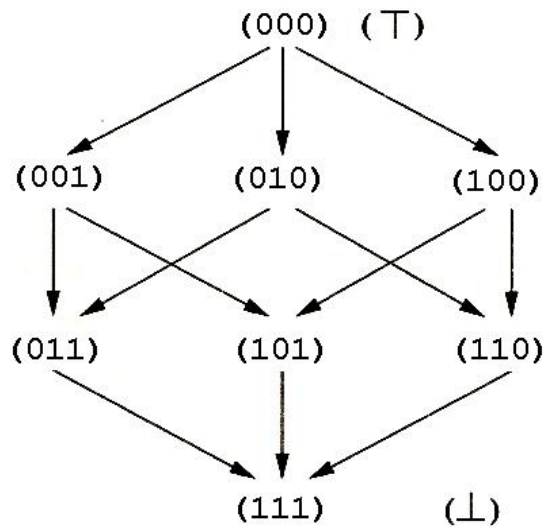
- ◇ *Диаграмма полурешетки* $\langle L, \wedge \rangle$ представляет собой граф, узлами которого являются элементы L , а ребра направлены от x к y , если $y \leq x$.
- ◇ **Пример.** На рисунке – диаграмма полурешетки $\langle U, \cup \rangle$, $|U| = 8$: элемент множества U представляется битовым 3-вектором.



3.2 Полурешетки

3.2.5. Диаграммы полурешеток

- ◇ *Диаграмма полурешетки* $\langle L, \wedge \rangle$ представляет собой граф, узлами которого являются элементы L , а ребра направлены от x к y , если $y \leq x$.
- ◇ **Пример.** На рисунке – диаграмма полурешетки $\langle U, \cup \rangle$, $|U| = 8$: элемент множества U представляется битовым 3-вектором.



- ◇ **Замечание.** Определение элемента $\perp$ внизу диаграммы: для любого $x \in L$: $\perp \wedge x = \perp$. Этот элемент называется «низом», так как по определению отношения $\leq$ для любого $x \in L$ $\perp \leq x$. Полурешетка с операцией $\wedge$ может и не содержать $\perp$.

3.2 Полурешетки

3.2.6 Наименьшая верхняя граница

- ◇ В частично упорядоченном множестве $\langle L, \leq \rangle$ можно по аналогии с $\inf(x, y)$ определить наименьшую верхнюю границу $\sup(x, y)$ как такой $b \in L$, что $x \leq b$, $y \leq b$, и если для $z \in L$ $x \leq z$ и $y \leq z$, то $b \leq z$.
- ◇ В полурешетке $\langle L, \wedge \rangle$ $\inf(x, y)$ существует для любой пары элементов x и $y \in L$, а $\sup(x, y)$ может не быть.
- ◇ В частично упорядоченном множестве $\langle L, \leq \rangle$, в котором для любых x и $y \in L$ существует $\sup(x, y)$, можно определить еще одну бинарную операцию $\vee$ (объединение) : $x \vee y = \sup(x, y)$.
- ◇ Множество, в котором определены обе операции – $\wedge$ (сбор) и $\vee$ (объединение), – называется *решеткой*.

3.3 Структура потока данных

3.3.1 Определение

◇ **Определение.** Структурой потока данных называется четверка

$$\langle D, F, L, \wedge \rangle ,$$

где

- ◇ D – направление анализа (*Forward* или *Backward*),
- ◇ F – семейство передаточных функций,
- ◇ L – поток данных (множество элементов полурешетки),
- ◇ $\wedge$ - реализация операции сбора.

◇ **Примеры.**

- 1 Структура потока данных для анализа **достигающих определений**: $\langle \textit{Forward}, \mathcal{GK}, \textit{Def}, \cup \rangle$,

где $\mathcal{GK}$ – семейство передаточных функций вида *gen-kill*,
 $\textit{Def}$ – множество определений переменных.

3.3 Структура потока данных

3.3.1 Определение

◇ Примеры.

- 2 Структура потока данных для анализа **живых переменных**:

$\langle \textit{Backward}, \mathcal{LV}, \textit{Var}, \cup \rangle$,

где $\mathcal{LV}$ – семейство передаточных функций вида,

$$f(x) = use \cup (x - def)$$

Def – множество определений переменных.

- 3 Структура потока данных для анализа **доступных выражений**:

$\langle \textit{Forward}, \mathcal{AE}, \textit{Expr}, \cap \rangle$,

где $\mathcal{AE}$ – семейство передаточных функций вида

$$f(x) = e\_gen \cup (x - e\_kill),$$

$Expr$ – множество выражений программы.

3.3 Структура потока данных

3.3.2 Замкнутость

- ◇ **Определение.** Семейство передаточных функций F называется *замкнутым*, если:
 - ◇ F содержит тождественную функцию $I: \forall x \in L: I(x) = x$.
 - ◇ F замкнуто относительно композиции:
$$\forall f, g \in F \Rightarrow h(x) = g(f(x)) \in F.$$
- ◇ **Утверждение 1.** Семейство передаточных функций F , используемое при анализе достигающих определений (передаточные функции вида *gen-kill*), является замкнутым.

3.3 Структура потока данных

3.3.2 Замкнутость

◇ **Утверждение 1.** Семейство передаточных функций F , используемое при анализе достигающих определений (передаточные функции вида *gen-kill*), является замкнутым.

1) Замкнутость относительно композиции уже установлена.

2) Тожественная функция $I(x) = x$ является функцией вида *gen-kill* с $gen = kill = \emptyset$.

3.3 Структура потока данных

3.3.2 Замкнутость

- ◇ **Утверждение 1.** Семейство передаточных функций F , используемое при анализе достигающих определений (передаточные функции вида *gen-kill*), является замкнутым.
 - 1) Замкнутость относительно композиции уже установлена.
 - 2) Тожественная функция $I(x) = x$ является функцией вида *gen-kill* с $gen = kill = \emptyset$.
- ◇ **Утверждение 2.** Семейство передаточных функций, используемое при анализе живых переменных является замкнутым.
- ◇ **Утверждение 3.** Семейство передаточных функций, используемое при анализе доступных выражений является замкнутым.

3.3 Структура потока данных

3.3.3 Монотонные структуры

- ◇ **Определение 1.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если $\forall x, y \in L, \forall f \in F (x \leq y) \Rightarrow f(x) \leq f(y)$.
- ◇ **Определение 2.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если $\forall x, y \in L, \forall f \in F f(x \wedge y) \leq f(x) \wedge f(y)$.

3.3 Структура потока данных

3.3.3 Монотонные структуры

- ◇ **Определение 1.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если $\forall x, y \in L, \forall f \in F (x \leq y) \Rightarrow f(x) \leq f(y)$.
- ◇ **Определение 2.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если $\forall x, y \in L, \forall f \in F f(x \wedge y) \leq f(x) \wedge f(y)$.
- ◇ **Утверждение.** Определения 3.3.3.1 и 3.3.3.2 эквивалентны.

3.3 Структура потока данных

3.3.3 Монотонные структуры

- ◇ **Определение 1.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если $\forall x, y \in L, \forall f \in F (x \leq y) \Rightarrow f(x) \leq f(y)$.
- ◇ **Определение 2.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если $\forall x, y \in L, \forall f \in F f(x \wedge y) \leq f(x) \wedge f(y)$.
- ◇ **Утверждение.** Определения 3.2.3.1 и 3.2.3.2 эквивалентны.
 - ◇ Из определения 1 следует определение 2:
$$x \wedge y = \inf(x, y) \Rightarrow x \wedge y \leq x \text{ и } x \wedge y \leq y \Rightarrow (1) \Rightarrow$$
$$f(x \wedge y) \leq f(x) \text{ и } f(x \wedge y) \leq f(y) \Rightarrow$$
$$f(x \wedge y) = \inf(f(x), f(y)) = f(x) \wedge f(y)$$
 - ◇ Из определения 2 следует определение 1:
$$x \leq y \Rightarrow x \wedge y = x \Rightarrow_{(\text{опр2})} f(x) \leq f(x) \wedge f(y),$$
$$f(x) \wedge f(y) = \inf(f(x), f(y)) \leq f(y); \Rightarrow f(x) \leq f(y)$$

3.3 Структура потока данных

3.3.4 Дистрибутивные структуры

◇ **Определение.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *дистрибутивной*, если

$$\forall x, y \in L, \forall f \in F: f(x \wedge y) = f(x) \wedge f(y).$$

3.3 Структура потока данных

3.3.4 Дистрибутивные структуры

◇ **Определение.** Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *дистрибутивной*, если

$$\forall x, y \in L, \forall f \in F: f(x \wedge y) = f(x) \wedge f(y).$$

◇ **Утверждение.** Если структура потока данных $\langle D, F, L, \wedge \rangle$ дистрибутивна, то она монотонна.

$$a = b \Rightarrow_{\text{идемпотентность}} a \wedge b = a \Rightarrow a \leq b$$

$$f(x \wedge y) = f(x) \wedge f(y) \Rightarrow f(x \wedge y) \leq f(x) \wedge f(y)$$

◇ Обратное утверждение неверно.

В качестве доказательства можно привести пример монотонной структуры потока данных, которая не дистрибутивна.

3.3 Структура потока данных

3.3.5. Дистрибутивность структуры достигающих определений

◇ **Утверждение.** Структура достигающих определений $RD = \langle Forward, Gen-kill, \emptyset, \cup \rangle$ дистрибутивна

◇ $y, z \in RD, f(x) = G \cup (x - K) \in Gen-kill.$

Докажем, что

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

(1) $y, z \in G$: равенство выполняется, так как G входит
и в левую, и в правую его части.

(2) $y, z \notin G$: G можно исключить из равенства:

$$(y \cup z) - K = (y - K) \cup (z - K)$$

Это равенство проверяется при помощи диаграмм Венна.

3.3 Структура потока данных

3.3.5. Дистрибутивность структуры достигающих определений

◇ **Утверждение.** Структура достигающих определений $RD = \langle Forward, Gen\text{-}kill, \emptyset, \cup \rangle$ дистрибутивна

◇ $y, z \in RD, f(x) = G \cup (x - K) \in Gen\text{-}kill.$

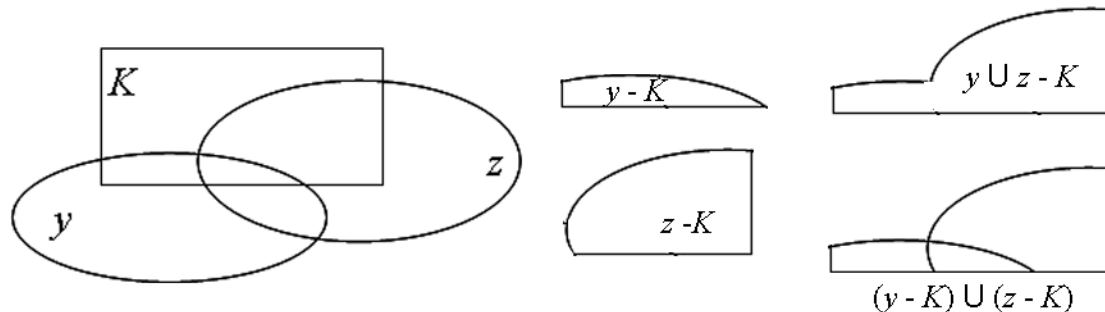
Докажем, что

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

(2) $y, z \notin G$: G можно исключить из равенства:

$$(y \cup z) - K = (y - K) \cup (z - K)$$

Это равенство проверяется при помощи диаграмм Венна.



3.3 Структура потока данных

3.3.6. Дистрибутивность структур живых переменных и доступных выражений

- ◇ **Следствие 1.** Структура живых переменных
 $LV = \langle Backward, Def-use, \emptyset, \cup \rangle$
дистрибутивна
- ◇ $f(x) \in Def-use$ алгебраически подобна функции класса *Gen-kill*.
- ◇ **Следствие 2.** Структура доступных выражений
 $AE = \langle Forward, Gen-kill, U, \cap \rangle$
дистрибутивна

3.4 Обобщенный итеративный алгоритм

3.4.1 Описание алгоритма

- ◇ **Алгоритм.** Итеративное решение задачи анализа потока данных
 - ◇ **Вход:** граф потока управления,
структура потока данных $\langle D, F, L, \wedge \rangle$,
передаточная функция $f_B \in F$
константа из $l \in L$ для граничного условия
 - ◇ **Выход:** значения из L для $In[B]$ и $Out[B]$ для каждого блока B в графе потока.
 - ◇ **Метод:** если $D = Forward$ выполнить программу (3.4.2);
если $D = Backward$ выполнить программу (3.4.3).

3.4 Обобщенный итеративный алгоритм

3.4.2 Решение задачи потока данных ($D = Forward$)

$Out[Entry] = l;$

for (*each* $B \neq Entry$) $Out[B] = \mathbf{T};$

while (внесены изменения в $In[B]$)

for (*each* $B \neq Entry$) {

$$In[B] = \bigwedge_{P \in Pred(B)} f_P(In[P])$$

}

3.4 Обобщенный итеративный алгоритм

3.4.3 Решение задачи потока данных ($D = Backward$)

$Out[Exit] = l;$

for (*each* $B \neq Exit$) $Out[B] = \top;$

while (внесены изменения в $Out[B]$)

for (*each* $B \neq Exit$) {

$$Out[B] = \bigwedge_{S \in Succ(B)} f_S(Out[S])$$

 }

3.5 Свойства итеративного алгоритма

3.5.1 Сходимость к решению

◇ **Утверждение.** Если обобщенный итеративный алгоритм сходится, то полученный результат является решением системы уравнений потока данных.

◇ $D = Forward:$

Если после очередной итерации цикла **while**

хотя бы для одного блока B уравнения

$$In[B] = \bigwedge_{P \in Pred(B)} f_P(In[P])$$

не удовлетворяется, то для этого B

$$InNew[B] \neq In[B].$$

Следовательно, в множество $In[B]$ будет внесено

изменение, блок B снова будет включен в $WorkList$

и итерации будут продолжены.

3.5 Свойства итеративного алгоритма

3.5.1 Сходимость к решению

◇ **Утверждение.** Если обобщенный итеративный алгоритм сходится, то получающийся результат является решением уравнений потоков данных.

◇ $D = \textit{Backward}$:

Аналогичные рассуждения, только вместо множеств $In[B]$ рассматриваются множества $Out[B]$.

◇ Дальнейшие рассмотрения будут только для случая $D = \textit{Forward}$

3.5 Свойства итеративного алгоритма

3.5.2 Максимальная фиксированная точка

◇ **Определение.** *Максимальная фиксированная точка* системы уравнений

$$In[B] = \bigwedge_{P \in Pred(B)} f_P(In[P])$$

представляет собой решение $\{In[B_i]^{max}\}$ этой системы, обладающее тем свойством, что для любого другого решения $\{In[B_i]\}$ выполняются условия

$$In[B_i] \leq In[B_i]^{max}$$

где $\leq$ – полурешеточное отношение частичного порядка.

3.5 Свойства итеративного алгоритма

3.5.3 Монотонность итераций

◇ **Утверждение 1.** Пусть $In[B_i]^j$ значения $In[B_i]$ после j -ой итерации.

Если структура потока данных монотонна, то для всех i

$$In[B_i]^{j+1} \leq In[B_i]^j$$

◇ Индукция по j .

3.5 Свойства итеративного алгоритма

3.5.3 Монотонность итераций

- ◇ **Утверждение 1.** Пусть $In[B]^j$ – значения $In[B]$ после j -ой итерации. Если структура потока данных монотонна, то

$$In[B]^{j+1} \leq In[B]^j$$

- ◇ Индукция по j .

- ◇ **Основание:**

$$In[B]^1 \leq In[B]^0$$

так как для $\forall B \neq Entry: In[B]^0 = \top$

3.5 Свойства итеративного алгоритма

3.5.3 Монотонность итераций

- ◇ **Утверждение 1.** Пусть $In[B]^j$ – значения $In[B]$ после j -ой итерации. Если структура потока данных монотонна, то

$$In[B]^{j+1} \leq In[B]^j$$

- ◇ Индукция по j .

- ◇ **Шаг:** Пусть для некоторого k $In[B]^k \leq In[B]^{k-1}$. Тогда $In[B]^{k+1} \leq In[B]^k$, так как

$$\begin{aligned} In[B]^{k+1} &= \bigwedge_{P \in Pred(B)} f_P \left(In[P]^k \right) \leq \\ &\leq \bigwedge_{P \in Pred(B)} f_P \left(In[P]^{k-1} \right) = In[B]^k \end{aligned}$$

(передаточная функция и операция сбор монотонны).

3.5 Свойства итеративного алгоритма

3.5.3 Монотонность итераций

◇ **Утверждение 1.** Пусть $In[B]^j$ – значение $In[B]$ после j -ой итерации.
Если структура потока данных монотонна, то $In[B]^{j+1} \leq In[B]^j$

◇ Индукция по j .

◇ **Шаг:** Пусть $In[B]^k \leq In[B]^{k-1}$ и $Out[B]^k \leq Out[B]^{k-1}$

$$In[B]^{k+1} = \bigwedge_{P \in Pred(B)} Out[P]^k \leq \bigwedge_{P \in Pred(B)} Out[P]^{k-1} = In[B]^k$$

так как операция сбора монотонна.

3.5 Свойства итеративного алгоритма

3.5.3 Монотонность итераций

◇ **Утверждение 2.** Пусть $In[B]^j$ – значение $In[B]$ после j -ой итерации. Если структура потока данных монотонна, то $In[B]^{j+1} \leq In[B]^j$

◇ **Следствие.** Для любого j

$$In[B] \leq In[B]^j$$

3.5 Свойства итеративного алгоритма

3.5.3 Монотонность итераций

◇ **Утверждение 3.** Если структура потока данных монотонна, то решение системы уравнений (1), найденное с помощью итеративного алгоритма, является максимальной фиксированной точкой этой системы.

◇ Требуется доказать, что для всех j

$$In[B_j] \leq In[B_j]^{IA},$$

где $\{In[B_j]^{IA}\}$ – решение системы (1), найденное с помощью итеративного алгоритма, $\{In[B_j]\}$ – любое другое решение этой системы.

Аналогично доказательству предыдущего утверждения.

3.5 Свойства итеративного алгоритма

3.5.4. Сходимость итеративного алгоритма

- ◇ **Определение 1.** *Восходящей цепочкой* в частично упорядоченном множестве $(L, \leq)$ называется последовательность его элементов, в которой

$$x_1 < x_2 < \dots < x_n.$$

- ◇ **Определение 2.** *Высотой* полурешетки называется наибольшее количество отношений $<$ в восходящих цепочках.
- ◇ **Утверждение.** Если полурешетка структуры монотонна и имеет конечную высоту, то итеративный алгоритм гарантированно сходится после количества итераций, не превышающего произведения высоты полурешетки на количество базовых блоков.

3.5 Свойства итеративного алгоритма

3.5.5 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

◇ Итеративный алгоритм

- (1) посещает базовые блоки не в порядке их выполнения, а в порядке обхода ГПУ (на каждой итерации каждый узел посещается только один раз)
- (2) в каждой точке сбора применяет операцию сбора к значениям потока данных, полученным к этому моменту
- (3) иногда в пределах итерации базовый блок B посещается до посещения его предшественников (прямой обход)

3.5 Свойства итеративного алгоритма

3.5.5 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

- ◇ (4) для процесса итерации *необходимо граничное условие*, так как к блоку *Entry* передаточная функция неприменима.
- (5) в качестве «нулевой итерации» все $Out[B]$ инициализируются значением T , которое, по определению, «не меньше» всех значений потока, и, следовательно, того значения, которое оно заменяет; при этом монотонность передаточных функций обеспечивает получение результата, «не меньшего», чем искомое решение:

3.5 Смысл решения уравнений потока данных

Без потери общности будем считать, что рассматривается прямая задача (обход графа потока от *Entry* к *Exit*).

Обратная задача (обход графа потока от *Exit* к *Entry*) рассматривается аналогично.

3.5.1 Идеальное решение

◇ Пусть f_{B_k} – передаточная функция блока B_k в графе потока.

Рассмотрим путь $P = \text{Entry} \rightarrow B_1 \rightarrow B_2 \rightarrow \dots \rightarrow B_{k-1} \rightarrow B_k$

(Путь P может содержать циклы: базовые блоки могут встречаться в нем по несколько раз).

По определению *передаточная функция пути* P :

$$f_P = f_{B_1} \circ f_{B_2} \circ \dots \circ f_{B_{k-1}}$$

(f_{B_k} не входит в композицию, так как путь достигает начала блока B_k , но не его конца). Значение потока данных, создаваемое этим путем, представляет собой $f_P(l_{\text{Entry}})$, где l_{Entry} – граничное условие.

3.5 Смысл решения уравнений потока данных

3.5.1 Идеальное решение

- ◇ Пусть $\mathcal{P} = \{P_1, P_2, \dots\}$ – множество *всех выполнимых* путей от $Entry$ до B_k

Путь P является *выполнимым* только тогда, когда известно выполнение программы (начальные данные), которое следует в точности по этому пути.

- ◇ Идеальным решением системы уравнений потока данных для $In[B_k]$ будет:

$$Ideal[B_k] = \bigwedge_{P \in \mathcal{P}} f_P(l_{Entry}).$$

- ◇ Решение $Ideal[B_k]$ названо идеальным, так как
 - (1) оно наиболее точное и
 - (2) вычислить его почти никогда не удастся, так как поиск всех возможных путей выполнения – задача неразрешимая. Следовательно, требуется поиск приближенного решения.⁴⁷

3.5 Смысл решения уравнений потока данных

3.5.2 Свойства решений уравнений потока данных для монотонных и дистрибутивных структур

- ◇ Добавление еще одного пути в сбор $\bigwedge_{p \in P}$ делает решение «меньше» в смысле частичного порядка полурешетки $\leq$
- ◇ Если просмотрены все выполнимые пути и ни одного лишнего, получается идеальное решение *Ideal*.
- ◇ Решение $Sol_1: Ideal \leq Sol_1$, получается, когда *просмотрены не все выполнимые пути*. Использовать решение Sol_1 опасно, так как для непросмотренных путей преобразования программы могут оказаться неверными (нарушается консервативность).

3.5 Смысл решения уравнений потока данных

3.5.2 Свойства решений уравнений потока данных для монотонных и дистрибутивных структур

◇ Решение Sol_2 : $Sol_2 \leq Ideal$ обладает следующими свойствами:

- (1) Sol_2 консервативно: оно содержит все выполнимые пути
- (2) Sol_2 неточно: в нем не отсеяны «лишние» пути, т.е. либо не существующие в графе потока, либо существующие, но такие, по которым программа никогда не проследует.

В результате:

- (1) Sol_2 может запрещать некоторые из преобразований, разрешенных решением $Ideal$.
- (2) все преобразования, которые разрешает Sol_2 , корректны

◇ В абстракции потока данных предполагается, что каждый путь в графе потока может быть пройден.

3.5 Смысл решения уравнений потока данных

3.5.3 Решение сбором по всем путям

- ◇ *Решение сбором по всем путям (MOP-решение)*
от *Entry* до входа в B_k определяется соотношением:

$$MOP[B_k] = \bigwedge_{P \in Q} f_P(l_{Entry}),$$

где $Q = \{P_1, P_2, \dots\}$ – множество *всех* путей от *Entry* до входа в блок B_k

- ◇ Пути, рассматриваемые в *MOP*-решении, – это надмножество всех выполнимых путей: *MOP*-решение собирает значения потоков данных как для всех выполнимых путей, так и для путей, которые не могут быть выполнены. Следовательно, для всех B_k выполняется соотношение

$$MOP[B_k] \leq Ideal[B_k].$$

3.5 Смысл решения уравнений потока данных

3.5.3 Решение сбором по всем путям

- ◇ **Замечание.** Для прямой задачи $MOP[B_k]$ дает значения для $In[B_k]$. Для обратной задачи $MOP[B_k]$ дает значения для $Out[B_k]$. Если рассматривается обратная задача, MOP -решение определяется соотношением :

$$MOP[B_k] = \bigwedge_{P \in Q} f_P(l_{Exit}),$$

где $Q = \{P_1, P_2, \dots\}$ – множество *всех* путей от выхода из B_k до $Exit$.

- ◇ Если в анализируемой программе есть циклы, количество всех путей, рассматриваемых при построении MOP -решения, становится бесконечным. Поэтому построить MOP -решение, как правило, не удастся (невозможно учесть бесконечное множество путей).

3.5 Смысл решения уравнений потока данных

3.5.4 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

◇ Итеративный алгоритм

- (1) посещает базовые блоки не в порядке их выполнения (как при вычислении *МОР*-решения), а в порядке обхода графа потока (на каждой итерации каждый узел посещается только один раз)
- (2) в каждой точке слияния алгоритм применяет операцию сбора к значениям потока данных, полученным к этому моменту
- (3) иногда в пределах итерации базовый блок *B* посещается до посещения его предшественников (прямой обход)

3.5 Смысл решения уравнений потока данных

3.5.4 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

- ◇ Следовательно:
 - ◇ *необходимо граничное условие*, так как к блоку *Entry* передаточная функция не применяется.
 - ◇ необходима инициализация потока данных для выходов из базовых блоков: $Out[B]$ инициализируется значением T , которое, как известно, «не меньше» всех значений потока, и, следовательно, того значения, которое оно заменяет.
 - ◇ при использовании T в качестве входных данных монотонность передаточных функций обеспечивает получение результата, «не меньшего», чем искомое решение:

$$MFP \leq MOP$$

3.5 Смысл решения уравнений потока данных

3.5.4 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

◇ Пример (см. рисунок)

Требуется вычислить значение $In[B_4]$.

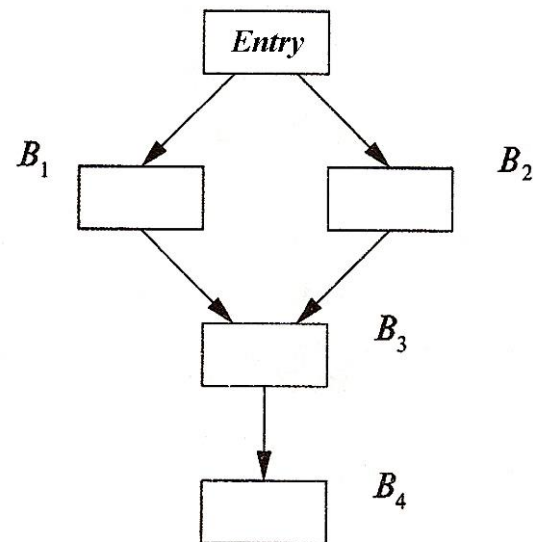
(1) MOP-решение:

$$\begin{aligned} MOP[B_4] &= \\ &= ((f_{B_1} \circ f_{B_3}) \wedge (f_{B_2} \circ f_{B_3}))(l_{Entry}) \end{aligned}$$

(2) Итеративный алгоритм (узлы посещаются
в порядке B_1, B_2, B_3, B_4)

$$In[B_4] = f_{B_3}(l_{Entry}) \wedge f_{B_2}(l_{Entry})$$

Операция сбора $\wedge$ в итеративном алгоритме
применяется раньше, чем при вычислении
MOP-решения



3.5 Смысл решения уравнений потока данных

3.5.4 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

◇ Если структура потока данных дистрибутивна, то

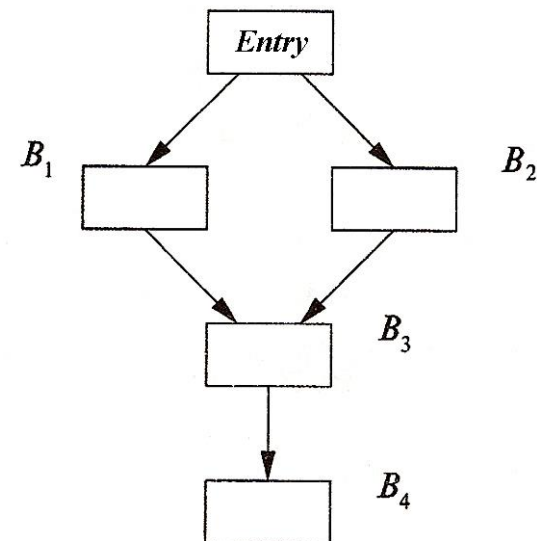
$$In[B_4] = MOP[B_4]$$

если же структура потока данных монотонна,
но не дистрибутивна, то

$$In[B_4] \leq MOP[B_4]$$

◇ Следовательно, в обоих случаях приближенное решение, представленное максимальной фиксированной точкой (*MFP*-решение), удовлетворяет условию:

$$MFP[B] \leq MOP[B]$$



3.5 Смысл решения уравнений потока данных

3.5.4 Решение, получаемое итеративным алгоритмом (максимальная фиксированная точка)

- ◇ Если структура потока данных дистрибутивна, то для всех B
$$MFP[B] = MOP[B].$$
- ◇ В разделах 3.2.5 и 3.2.6 была установлена дистрибутивность структур достигающих определений (RD), живых переменных (LV) и доступных выражений (AE).

Следовательно, в указанных случаях итеративный алгоритм позволяет получить MOP -решение.

3.5 Смысл решения уравнений потока данных

3.5.5. Консервативность *MFP*-решения

- ◇ **Утверждение.** *MFP*-решение, получаемое итеративным алгоритмом, всегда консервативно

Доказательство

Индукция по i : значения, полученные итеративным алгоритмом после i итераций, не превосходят результата операции сбора по всем путям длины i .

Итеративный алгоритм завершается только тогда, когда он получает такой же ответ, как и при неограниченном количестве итераций.

Следовательно, $MFP \leq MOP$.

Но уже установлено: $MOP \leq Ideal$.

Следовательно (транзитивность $\leq$), $MFP \leq Ideal$

и *MFP*-решение консервативно.

4. Доминаторы и постдоминаторы

4.1 Доминаторы

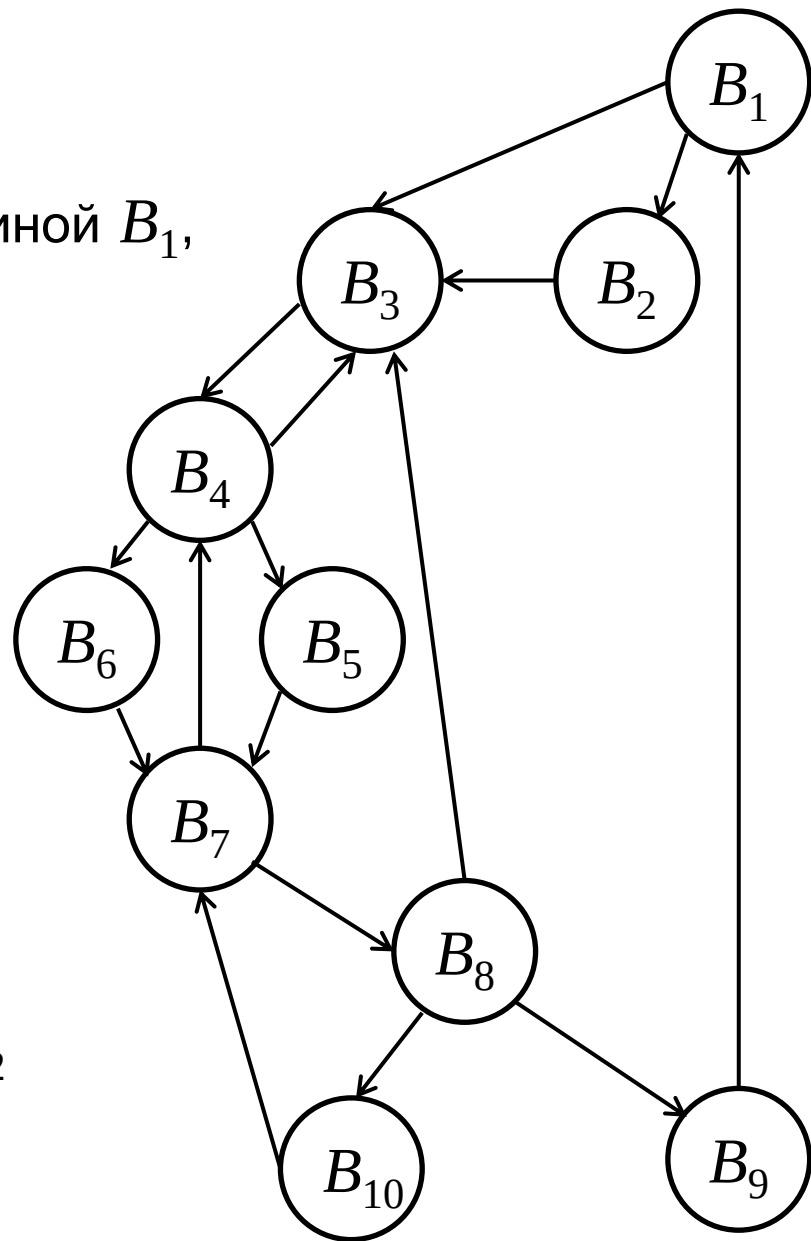
4.1.1 Определение

- ◇ В ГПУ вершина d является *доминатором* вершины n (этот факт записывается как $d \text{ dom } n$ или $d = \text{Dom}(n)$), если любой путь от вершины Entry до вершины n проходит через вершину d .
- ◇ **Замечание.** Из определения 4.2.1 следует, что каждая вершина n является доминатором самой себя, так как путь от Entry до n проходит через n .

4.1 Доминаторы

4.1.2 Примеры доминаторов

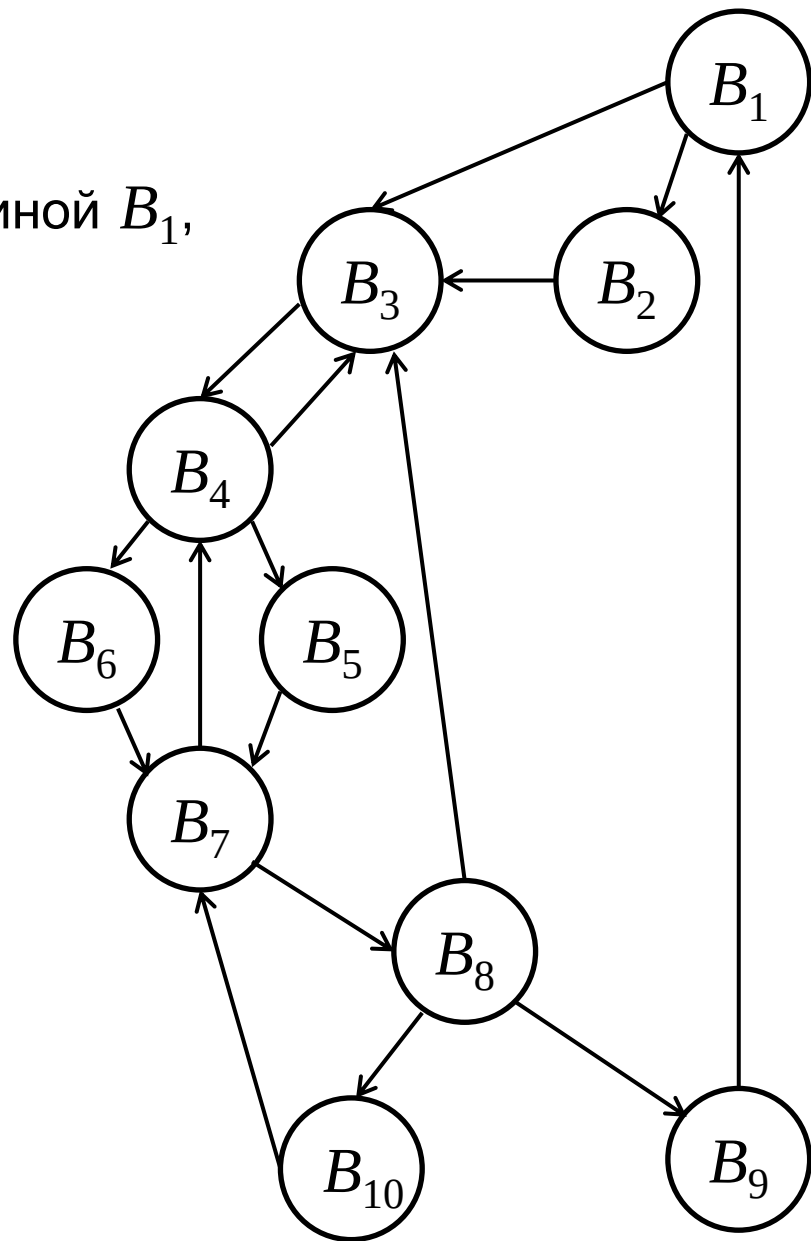
- ◇ Рассмотрим ГПУ с входной вершиной B_1 , показанный на рисунке.
- ◇ B_1 является доминатором всех узлов, включая себя самого.
- ◇ B_2 является доминатором только себя самого.
- ◇ B_3 является доминатором всех вершин, кроме B_1 и B_2 .
- ◇ B_4 является доминатором всех вершин, кроме B_1 , B_2 и B_3 .
- ◇ B_5 и B_6 являются доминаторами только себя.



4.1 Доминаторы

4.1.2 Примеры доминаторов

- ◇ Рассмотрим ГПУ с входной вершиной B_1 , показанный на рисунке.
- ◇ B_7 является доминатором вершин B_7 , B_8 , B_9 и B_{10} .
- ◇ B_8 является доминатором вершин B_8 , B_9 и B_{10} .
- ◇ B_9 и B_{10} являются доминаторами только себя.



4.1 Доминаторы

4.1.3 Свойства отношения dom

- ◇ 1. **Утверждение.** Отношение dom рефлексивно, антисимметрично и транзитивно, т.е. является отношением частичного порядка.
- (1) *Рефлексивность: $a dom a$.*
- (2) *Антисимметричность: если $a dom b$ и $b dom a$,
то $a = b$.*
- (3) *Транзитивность: если $a dom b$ и $b dom c$, то $a dom c$.*
- ◇ 2. **Утверждение.** Для любой вершины n ГПУ каждый ациклический путь от $Entry$ до n проходит через все доминаторы n , причем на всех таких путях доминаторы проходятся в *одном и том же порядке*

4.1 Доминаторы

4.1.3 Свойства отношения *dom*

- ◇ 3. **Определение.** Вершина i ГПУ является *непосредственным доминатором* вершины n ($i = \text{Idom}(n)$), если
 - (1) $i = \text{Dom}(n)$
 - (2) не существует вершины m , $m \neq i$, $m \neq n$, такой что $i = \text{Dom}(m)$ и $m = \text{Dom}(n)$.
- ◇ 4. **Утверждение.** У каждой вершины n за исключением *Entry* существует единственный непосредственный доминатор. Доказательство тривиально.
- ◇ 5. **Определение.** Вершина s ГПУ является *строгим доминатором* вершины n , ($s = \text{Sdom}(n)$), если $s = \text{Dom } n$ и $s \neq n$.

4.1 Доминаторы

4.1.3 Свойства отношения *dom*

- ◇ 6 **Утверждение.** Пусть n – вершина ГПУ,
 $Pred(n) = \{p_1, p_2, \dots, p_k\}$ и $d \neq n$.
Тогда $d = Dom(n)$ тогда и только тогда, когда $\forall i \ d = Dom(p_i)$.
- ◇ 7 **Утверждение.** Множество **строгих** доминаторов вершины n является пересечением множеств доминаторов всех ее предшественников.

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

- ◇ **Задача поиска всех доминаторов** вершин ГПУ формулируется как задача анализа потока данных в прямом направлении.

Значением потока данных на входе в блок B является множество вершин (базовых блоков), являющихся строгими доминаторами блока B .

Операцией сбора является операция пересечения множеств.

Передаточная функция f_B добавляет вершину B к рассматриваемому множеству вершин.

Граничное условие: единственным доминатором вершины $Entry$ является она сама.

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

◇ Алгоритм

Вход: граф потока $G = \langle N, E \rangle$ с входным узлом $Entry$.

Выход: для каждой вершины $n \in N$ множество $D(n)$ ее доминаторов.

Метод: найти решение следующей задачи потока данных (вершины n соответствуют базовым блокам)

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

Область определения	Множество подмножеств базовых блоков
Направление обхода	<i>Forward</i>
Передаточная функция	$f_B(x) = x \cup \{B\}$
Граничное условие	$Out [Entry] = Entry$
Операция сбора ($\wedge$)	$\cap$
Система уравнений	$In[B] = \bigcap_{P \in Pred(B)} (In[P] \cup \{P\})$
Начальное приближение	$In[B] = N$ для всех B

Замечание. В соответствии с утверждением 7 $In[B]$ – множество строгих доминаторов блока B .

Множество $D(B)$ всех доминаторов блока B связано с $In[B]$ соотношением: $D(B) = \{B\} \cup In[B]$.

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

Таким образом, систему уравнений для вычисления $D(B)$ можно записать в виде

$$\begin{aligned} D(B) &= \{B\} \cup In[B] = \{B\} \cup \left(\bigcap_{P \in Pred(B)} (In[P] \cup \{P\}) \right) = \\ &= \{B\} \cup \left(\bigcap_{P \in Pred(B)} D(P) \right) \end{aligned}$$

или

$$D(B) = \{B\} \cup \left(\bigcap_{P \in Pred(B)} D(P) \right)$$

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

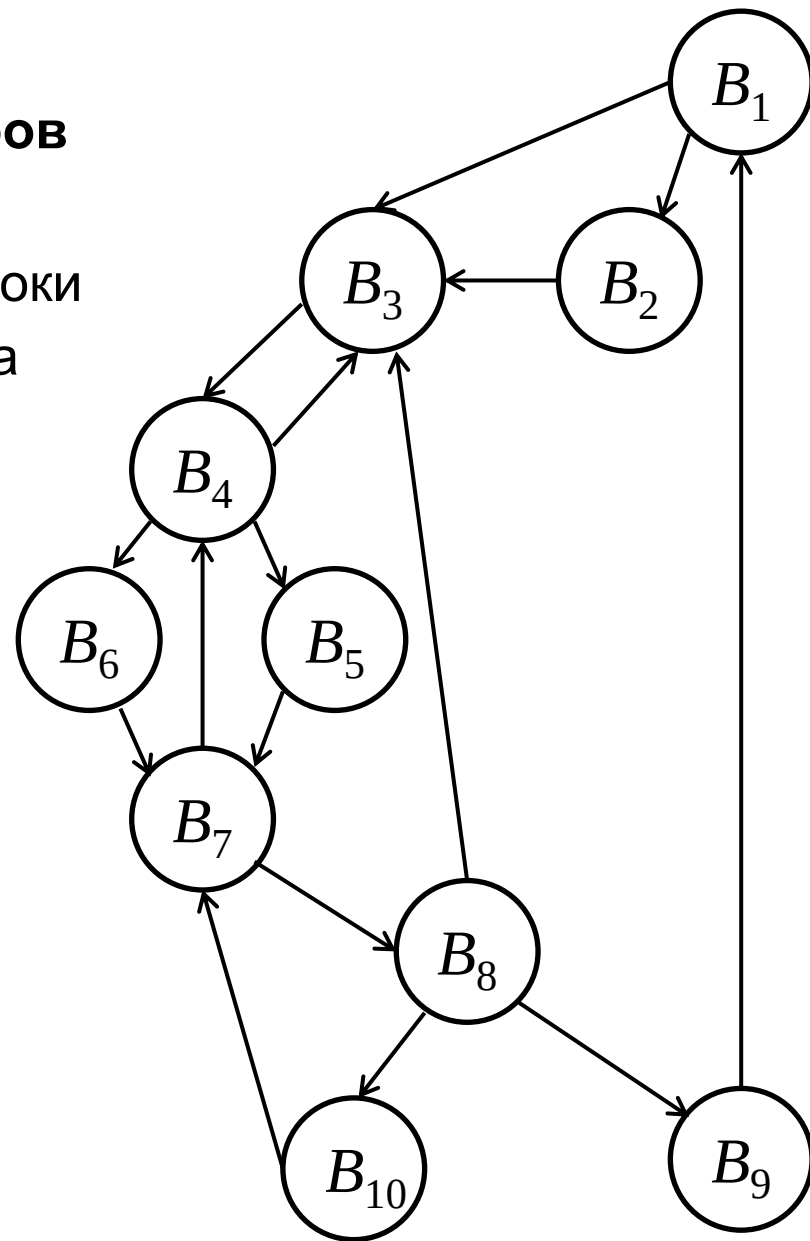
- ◇ **Пример.** Применим алгоритм к ГПУ на рисунке в предположении, что блоки посещаются в порядке их номеров, а *Entry* это блок B_1 .

- ◇ **Первая итерация**

Граничное условие: $D(B_1) = \{B_1\}$

$Pred(B_2) = \{B_1\}$

$D(B_2) = \{B_2\} \cup D(B_1) = \{B_1, B_2\}$



4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

- ◇ **Пример.** Применим алгоритм к ГПУ на рисунке в предположении, что блоки посещаются в порядке их номеров, а *Entry* это блок B_1 .

- ◇ **Первая итерация**

Граничное условие: $D(B_1) = \{B_1\}$

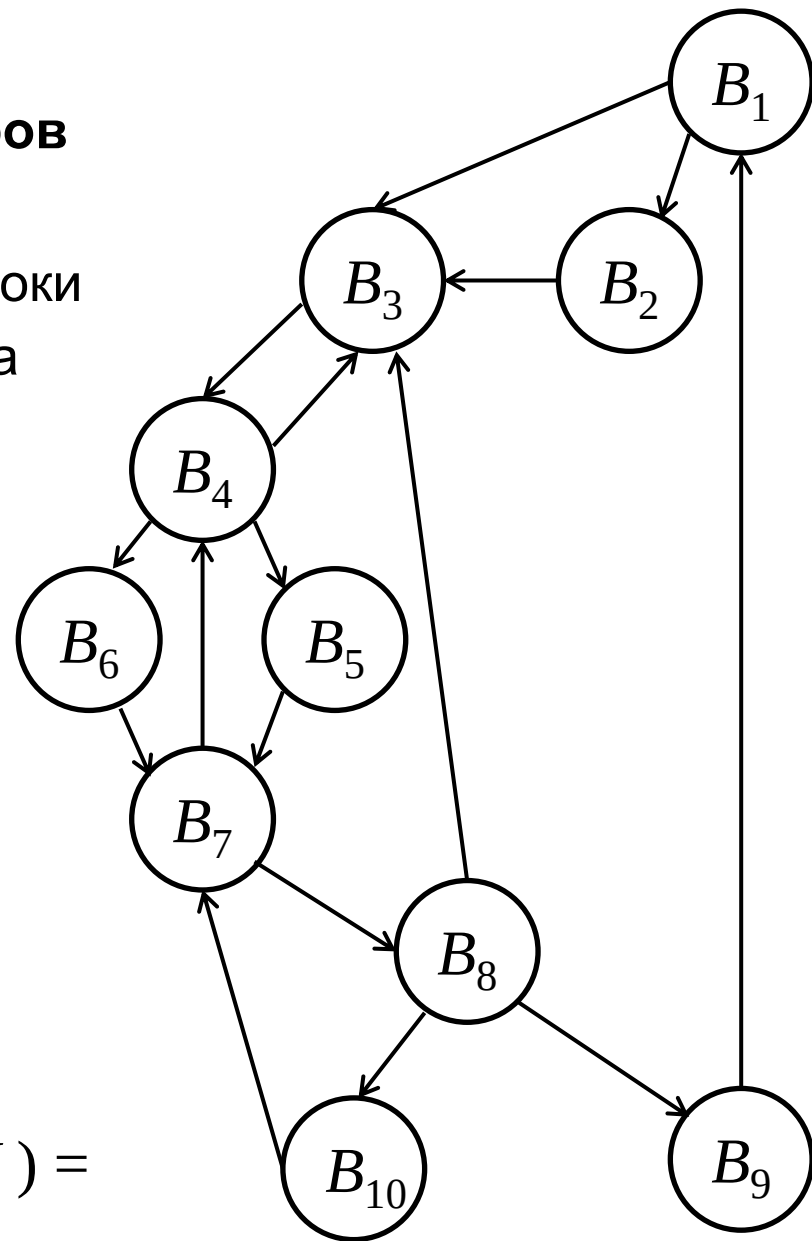
$Pred(B_2) = \{B_1\}$

$D(B_2) = \{B_2\} \cup D(B_1) = \{B_1, B_2\}$

$Pred(B_3) = \{B_1, B_2, B_4, B_8\}$

$D(B_3) = \{B_3\} \cup (D(B_1) \cap D(B_2) \cap D(B_4) \cap D(B_8)) = \{B_3\} \cup (\{B_1\} \cap \{B_1, B_2\} \cap N \cap N) = \{B_1, B_3\}$

$N = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}\}$



4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

◇ Первая итерация (окончание)

$$Pred(B_4) = \{B_3, B_7\}$$

$$\begin{aligned} D(B_4) &= \{B_4\} \cup (D(B_3) \cap D(B_7)) = \{B_4\} \cup (\{B_1, B_3\} \cap N) = \\ &= \{B_1, B_3, B_4\} \end{aligned}$$

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

◇ Первая итерация (окончание)

$$Pred(B_4) = \{B_3, B_7\}$$

$$\begin{aligned} D(B_4) &= \{B_4\} \cup (D(B_3) \cap D(B_7)) = \{B_4\} \cup (\{B_1, B_3\} \cap N) = \\ &= \{B_1, B_3, B_4\} \end{aligned}$$

$$Pred(B_5) = Pred(B_6) = \{B_4\}$$

$$D(B_5) = \{B_5\} \cup D(B_4) = \{B_5\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_5\}$$

$$D(B_6) = \{B_6\} \cup D(B_4) = \{B_6\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_6\}$$

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

◇ Первая итерация (окончание)

$$Pred(B_4) = \{B_3, B_7\}$$

$$D(B_4) = \{B_4\} \cup (D(B_3) \cap D(B_7)) = \{B_4\} \cup (\{B_1, B_3\} \cap N) = \\ = \{B_1, B_3, B_4\}$$

$$Pred(B_5) = Pred(B_6) = \{B_4\}$$

$$D(B_5) = \{B_5\} \cup D(B_4) = \{B_5\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_5\}$$

$$D(B_6) = \{B_6\} \cup D(B_4) = \{B_6\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_6\}$$

$$Pred(B_7) = \{B_5, B_6, B_{10}\}$$

$$D(B_7) = \{B_7\} \cup (D(B_5) \cap D(B_6)) \cap D(B_{10})) = \{B_7\} \cup (\{B_1, B_3, B_4, B_5\} \cap \\ = \{B_1, B_3, B_4, B_6\} \cap N) = \{B_1, B_3, B_4, B_7\}$$

$$Pred(B_8) = \{B_7\}$$

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

◇ Первая итерация (окончание)

$$Pred(B_4) = \{B_3, B_7\}$$

$$D(B_4) = \{B_4\} \cup (D(B_3) \cap D(B_7)) = \{B_4\} \cup (\{B_1, B_3\} \cap N) = \\ = \{B_1, B_3, B_4\}$$

$$Pred(B_5) = Pred(B_6) = \{B_4\}$$

$$D(B_5) = \{B_5\} \cup D(B_4) = \{B_5\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_5\}$$

$$D(B_6) = \{B_6\} \cup D(B_4) = \{B_6\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_6\}$$

$$Pred(B_7) = \{B_5, B_6, B_{10}\}$$

$$D(B_7) = \{B_7\} \cup (D(B_5) \cap D(B_6)) \cap D(B_{10})) = \{B_7\} \cup (\{B_1, B_3, B_4, B_5\} \cap \\ = \{B_1, B_3, B_4, B_6\} \cap N) = \{B_1, B_3, B_4, B_7\}$$

$$Pred(B_8) = \{B_7\}$$

$$D(B_8) = \{B_8\} \cup D(B_7) = \{B_8\} \cup \{B_1, B_3, B_4, B_7\} = \{B_1, B_3, B_4, B_7, B_8\}$$

$$Pred(B_9) = Pred(B_{10}) = \{B_8\}$$

$$D(B_9) = \{B_9\} \cup D(B_8) = \{B_9\} \cup \{B_1, B_3, B_4, B_7, B_8\} = \{B_1, B_3, B_4, B_7, B_8, B_9\}$$

$$D(B_{10}) = \{B_{10}\} \cup D(B_8) = \{B_{10}\} \cup \{B_1, B_3, B_4, B_7, B_8\} = \\ = \{B_1, B_3, B_4, B_7, B_8, B_{10}\}$$

4.1 Доминаторы

4.1.4 Алгоритм вычисления доминаторов

◇ Первая итерация (окончание)

$$Pred(B_4) = \{B_3, B_7\}$$

$$D(B_4) = \{B_4\} \cup (D(B_3) \cap D(B_7)) = \{B_4\} \cup (\{B_1, B_3\} \cap N) = \\ = \{B_1, B_3, B_4\}$$

$$Pred(B_5) = Pred(B_6) = \{B_4\}$$

$$D(B_5) = \{B_5\} \cup D(B_4) = \{B_5\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_5\}$$

$$D(B_6) = \{B_6\} \cup D(B_4) = \{B_6\} \cup \{B_1, B_3, B_4\} = \{B_1, B_3, B_4, B_6\}$$

$$Pred(B_7) = \{B_5, B_6\}$$

$$D(B_7) = \{B_7\} \cup (D(B_5) \cap D(B_6)) = \{B_7\} \cup (\{B_1, B_3, B_4, B_5\} \cap \{B_1, B_3, B_4, B_6\}) = \\ = \{B_1, B_3, B_4, B_6\} \cap N = \{B_1, B_3, B_4, B_7\}$$

$$Pred(B_8) = \{B_7\}$$

$$D(B_8) = \{B_8\} \cup D(B_7) = \{B_8\} \cup \{B_1, B_3, B_4, B_7\} = \{B_1, B_3, B_4, B_7, B_8\}$$

$$Pred(B_9) = Pred(B_{10}) = \{B_8\}$$

$$D(B_9) = \{B_9\} \cup D(B_8) = \{B_9\} \cup \{B_1, B_3, B_4, B_7, B_8\} = \{B_1, B_3, B_4, B_7, B_8, B_9\}$$

$$D(B_{10}) = \{B_{10}\} \cup D(B_8) = \{B_{10}\} \cup \{B_1, B_3, B_4, B_7, B_8\} = \\ = \{B_1, B_3, B_4, B_7, B_8, B_{10}\}$$

**Полученные значения
 $D(B_1) - D(B_{10})$ на второй
итерации не изменяются**

4.1 Доминаторы

4.1.5 Непосредственные доминаторы и дерево доминаторов

n	$D(n)$	$Idom(n)$
B_1	—	—
B_2	B_1	B_1
B_3	B_1	B_1
B_4	B_1, B_3	B_3
B_5	B_1, B_3, B_4	B_4
B_6	B_1, B_3, B_4	B_4
B_7	B_1, B_3, B_4	B_4
B_8	B_1, B_3, B_4, B_7	B_7
B_9	B_1, B_3, B_4, B_7, B_8	B_8
B_{10}	B_1, B_3, B_4, B_7, B_8	B_8

В таблице приведены списки строгих доминаторов каждой вершины ГПУ из рассмотренного примера.

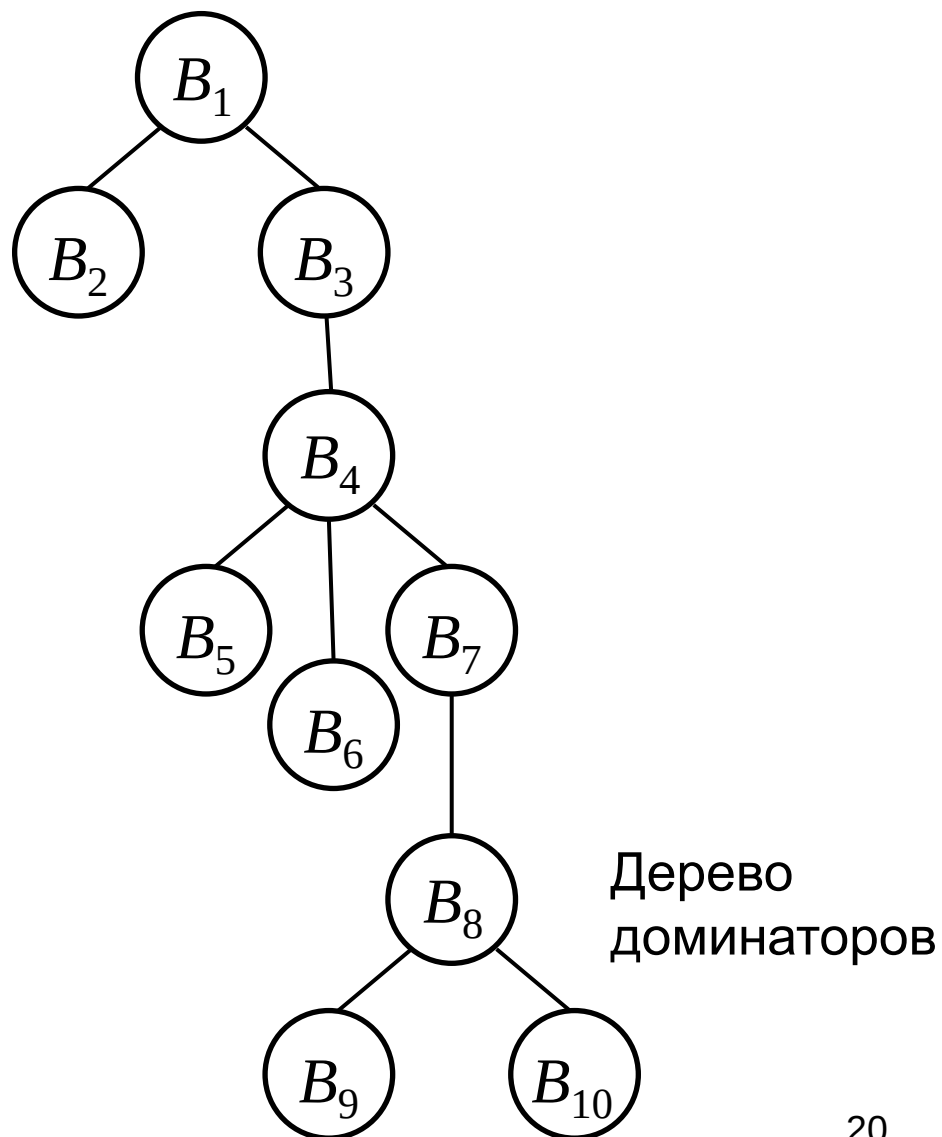
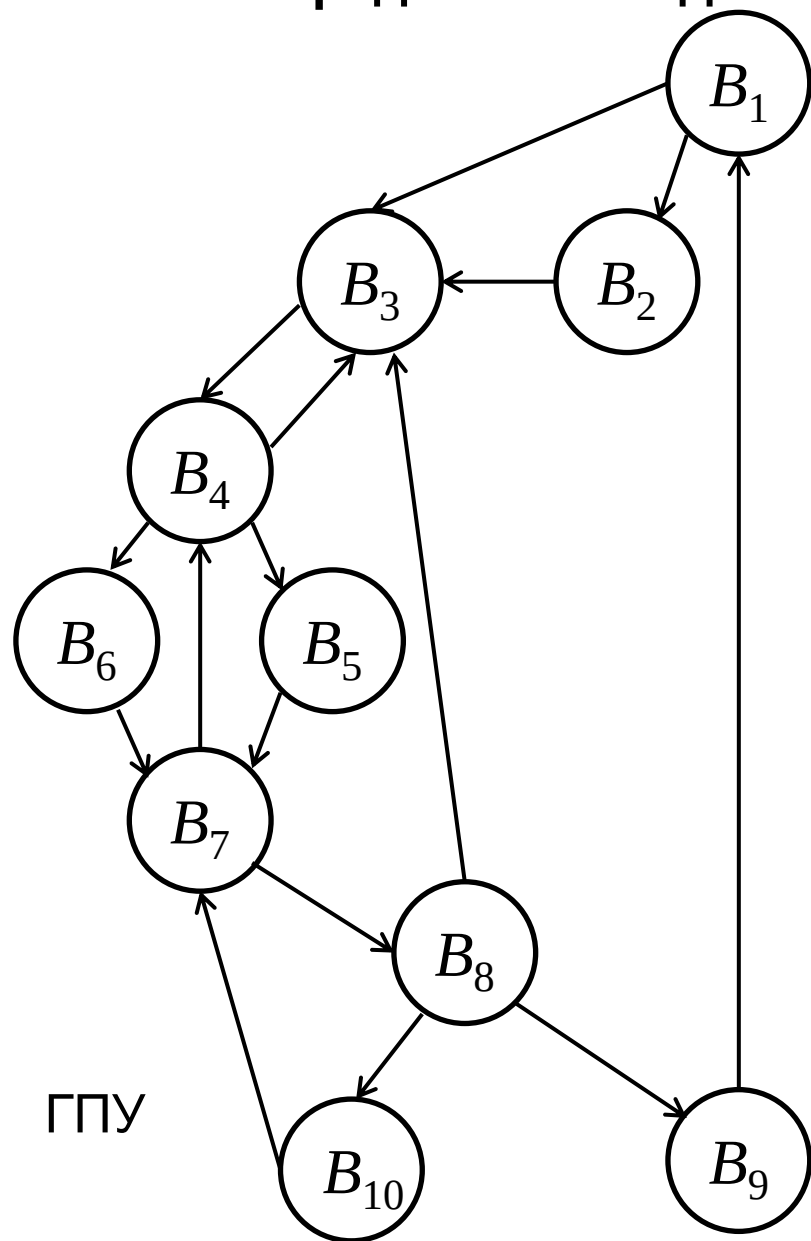
Непосредственный доминатор в каждом списке последний.

Соединив дугами для каждого $n \in N$

$Idom(n)$ с n , получим дерево доминаторов. Оно изображено на следующем слайде

4.1 Доминаторы

4.1.5 Непосредственные доминаторы и дерево доминаторов



4.2 Граница доминирования

4.2.1. Определение границы доминирования (*Dominance Frontier*)

◇ **Определение.** Рассмотрим узлы ГПУ p , n и m .

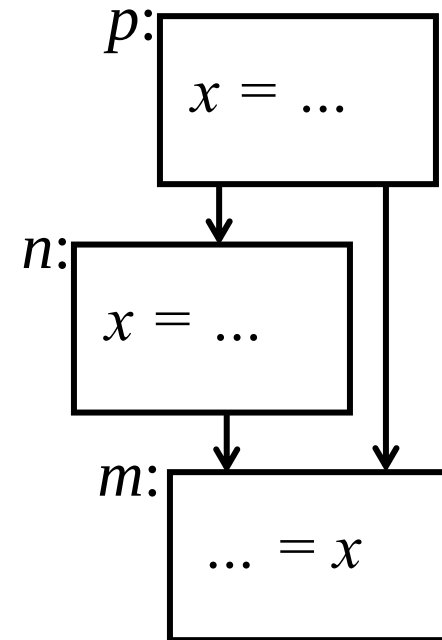
Если выполняются условия:

- (1) $n, p \in \text{Pred}(m) \ \&$
 $p \in \text{Dom}(n) \ \&$
 $p \in \text{Dom}(m)$
- (2) $n \neq m \rightarrow n \notin \text{Dom}(m)$

(n может быть только **нестрогим**
доминатором m).

то m входит в состав *границы доминирования* n .

Обозначение: $m \in DF(n)$.

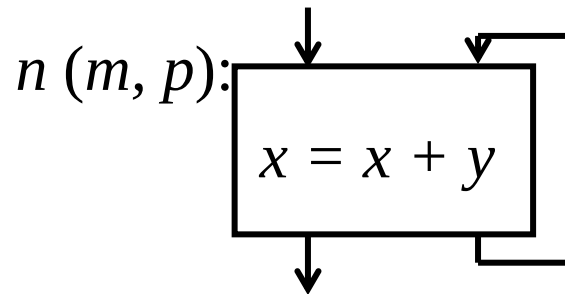


◇ **Неформально:** $DF(n)$ содержит все **первые** узлы, которые достижимы из n , на любом пути графа потока, проходящем через n , но над которыми n не доминирует.

4.2 Граница доминирования

4.2.1. Определение границы доминирования

- ◇ **Замечание.** Условие (2) Определения 4.2.1 позволяет определить границу доминирования в случае цикла, тело которого состоит из единственного базового блока: в этом случае n , m и p совпадают и n является своей собственной границей доминирования.



$$n = DF(n)$$

4.2 Граница доминирования

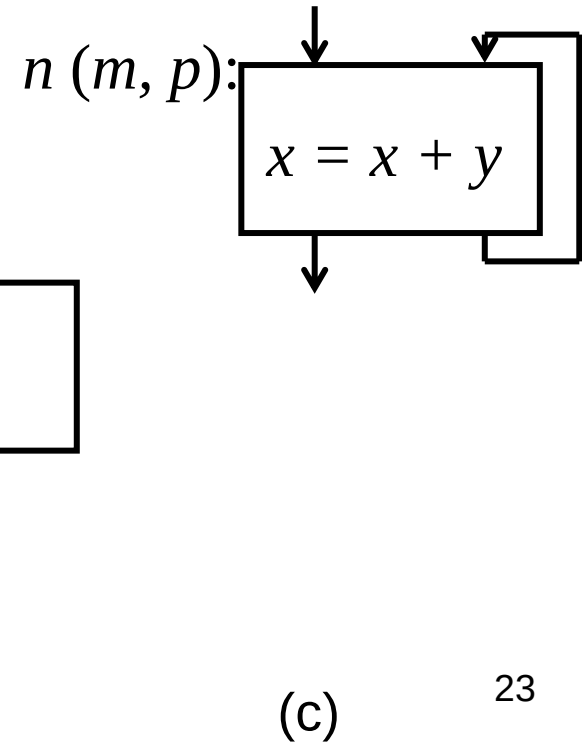
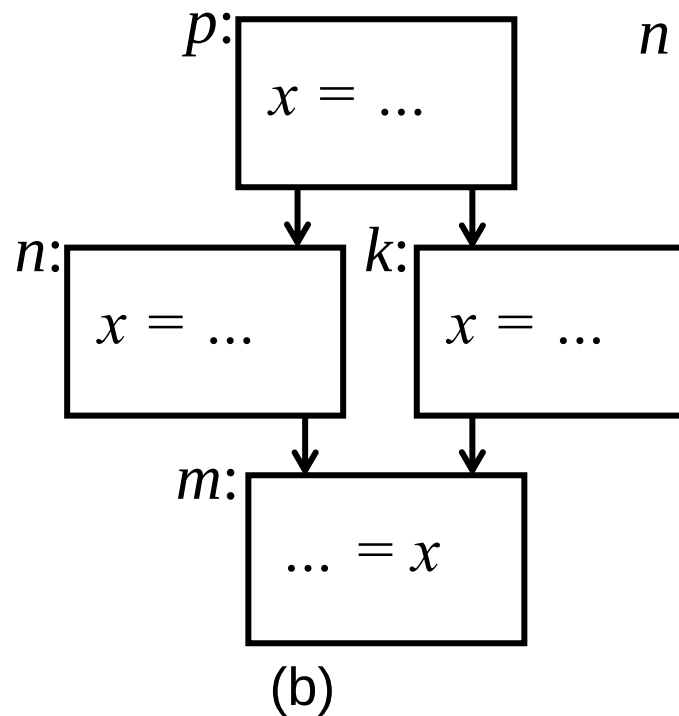
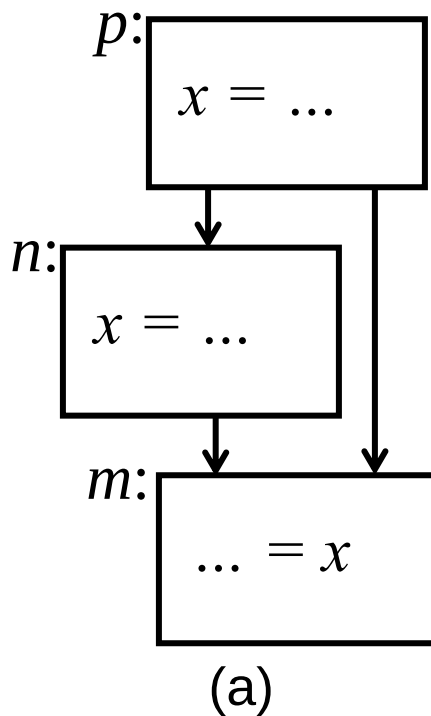
4.2.1. Определение границы доминирования

◇ На рисунках показаны три варианта границ доминирования:

(a) $m = DF(n)$

(b) $m = DF(n) \ \& \ m = DF(k)$

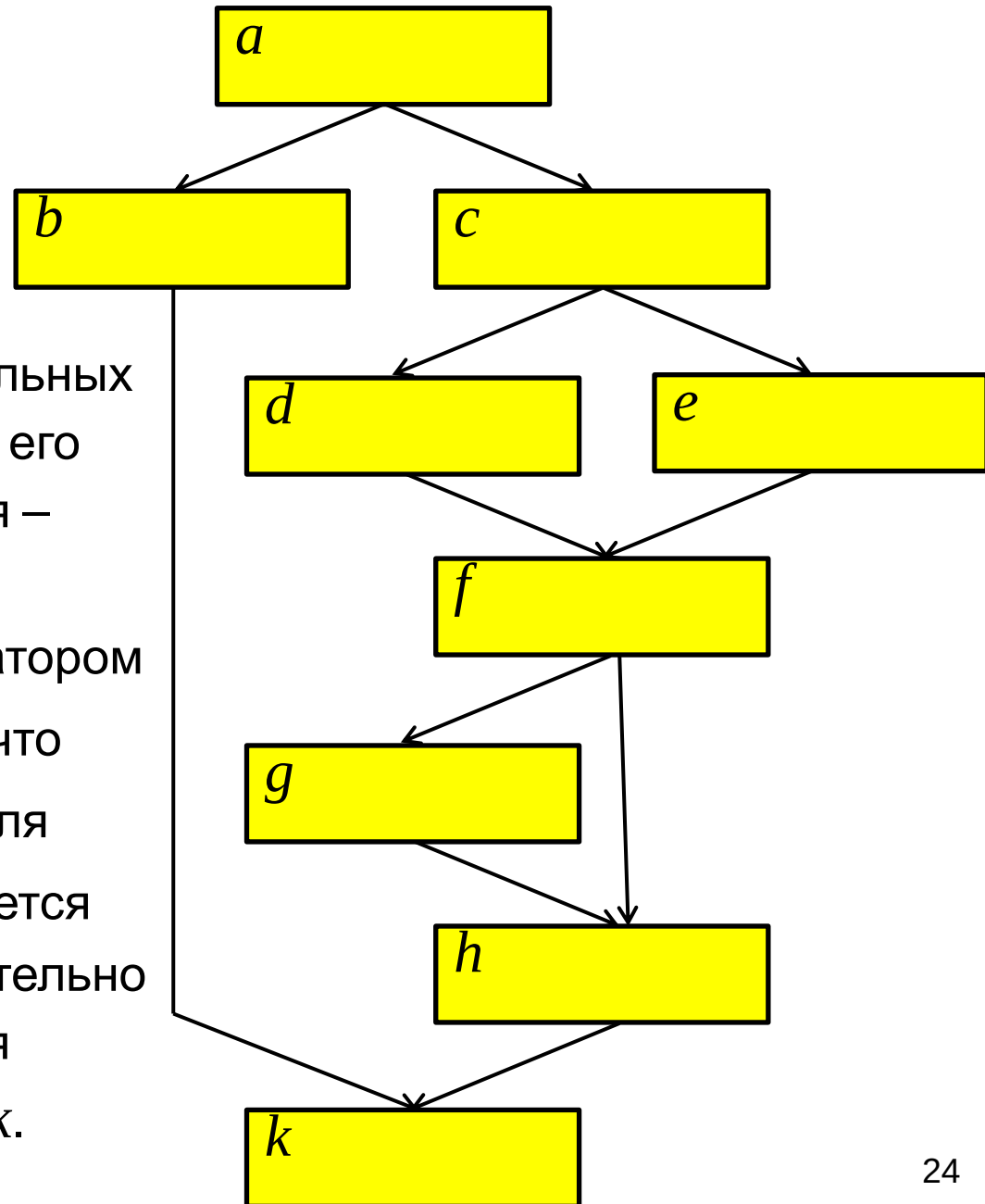
(c) $n = DF(n)$



4.2 Граница доминирования

4.2.1. Определение границы доминирования

- ◇ На рисунке справа:
 - ◇ Блок a является доминатором всех остальных блоков; следовательно, его граница доминирования – пустое множество
 - ◇ Блок c является доминатором блоков d, e, f, g, h , так что блок k – первый блок, для которого блок c не является доминатором ; следовательно граница доминирования блока c содержит блок k .

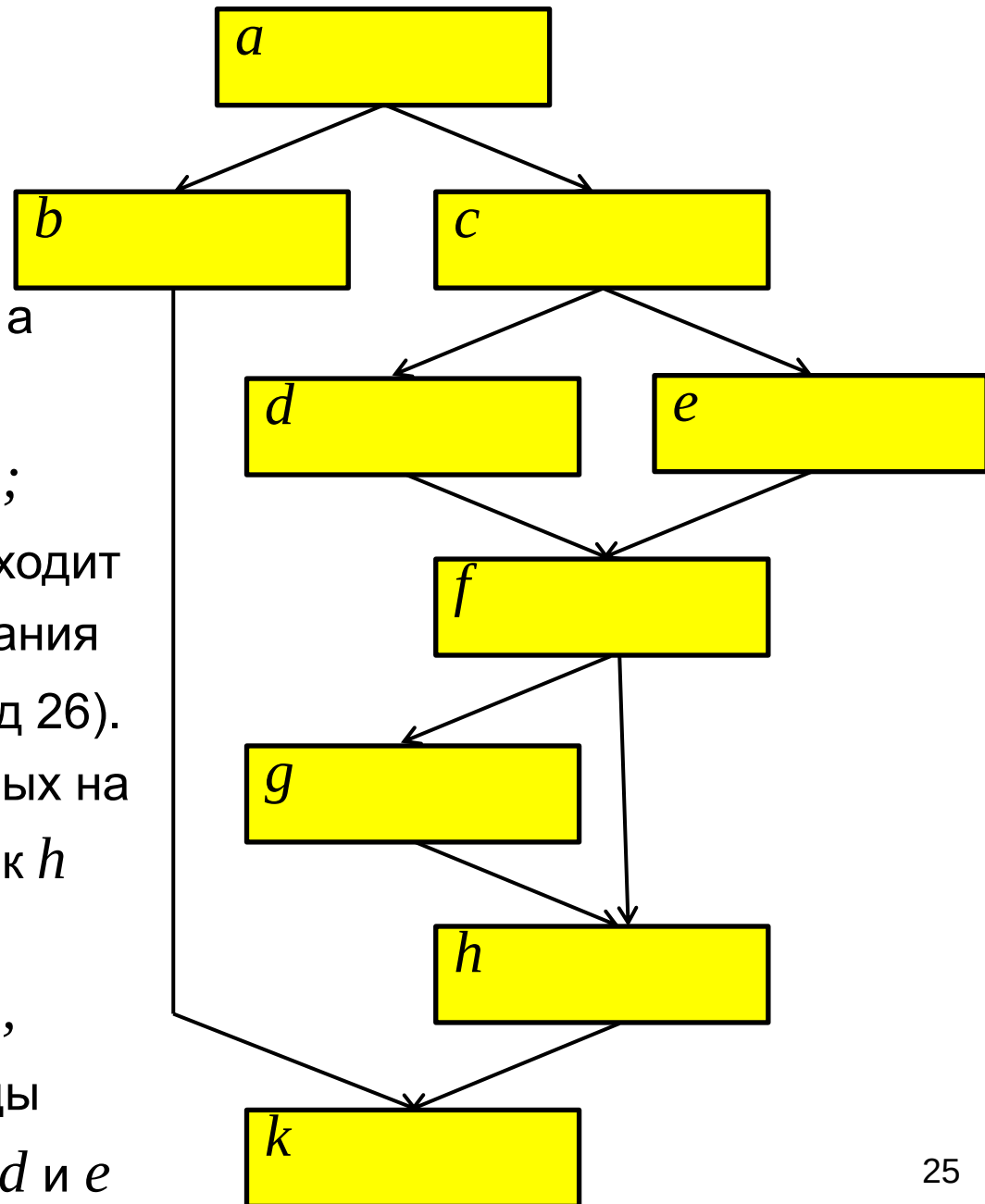


4.2 Граница доминирования

4.2.1. Определение границы доминирования

◇ На рисунке справа:

- ◇ Блоки c и f являются доминаторами блока h , а блок k входит в границу доминирования блока h ; следовательно блок k входит и в границы доминирования блоков c и f . (см. слайд 26).
- ◇ Из рисунков, приведенных на слайде 22 ясно, что блок h входит в границу доминирования блока g , а блок f входит в границы доминирования блоков d и e



4.2 Граница доминирования

4.2.1. Определение границы доминирования

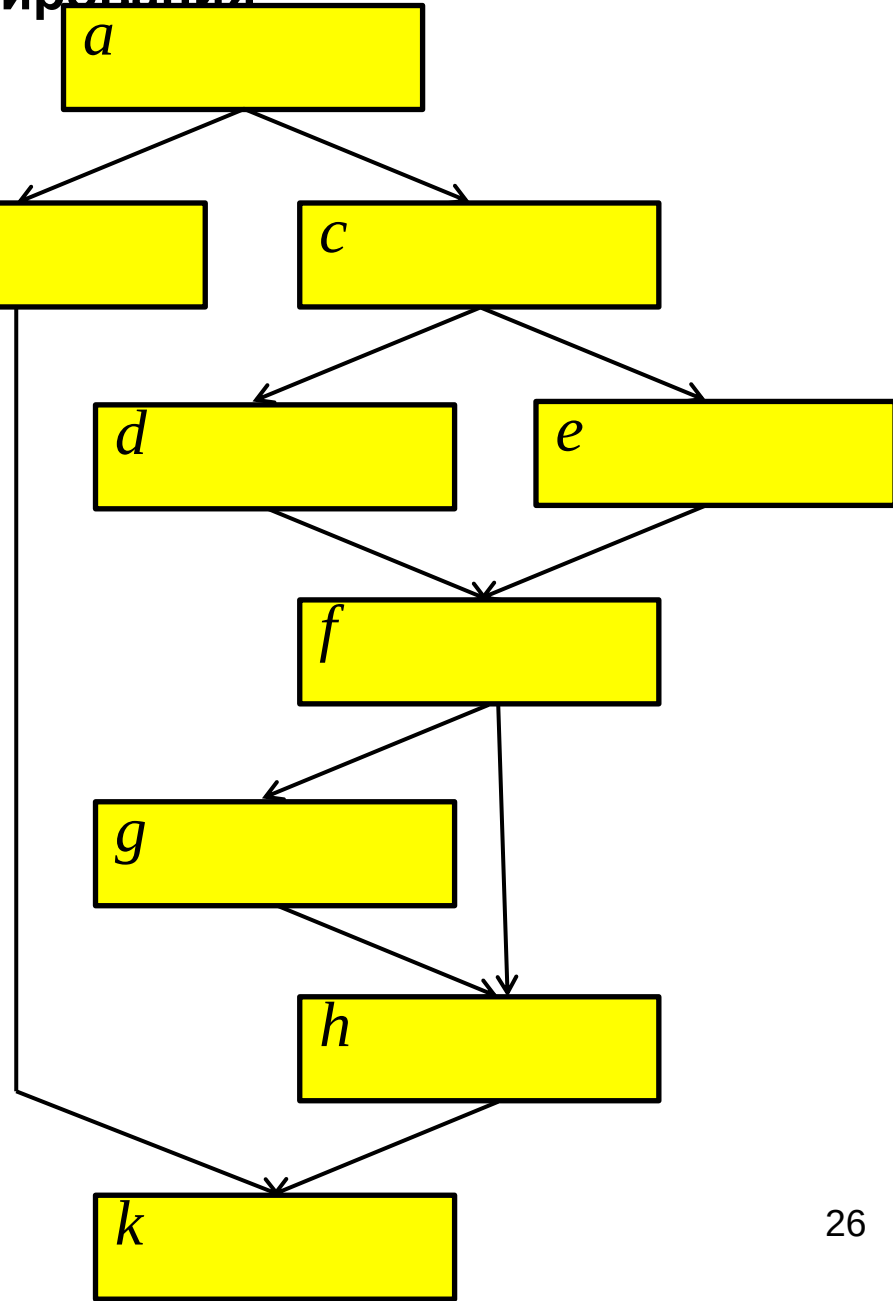
◇ На рисунке справа:

◇ Рассмотрим цепочку блоков:

a, c, f, h . Все они являются доминаторами блока k .

◇ Блок k входит в границу доминирования блока h . Он же входит в границы доминирования доминаторов блока h : блоков a, c, f .

◇ Поэтому при нахождении границы доминирования какого-нибудь блока, мы получаем и границы доминирования его доминаторов.



4.2 Граница доминирования

4.2.1. Определение границы доминирования

◇ На рисунке справа

$B_3 \in Dom(B_4), B_3 \in Dom(B_5),$

$B_3 \in Dom(B_6),$

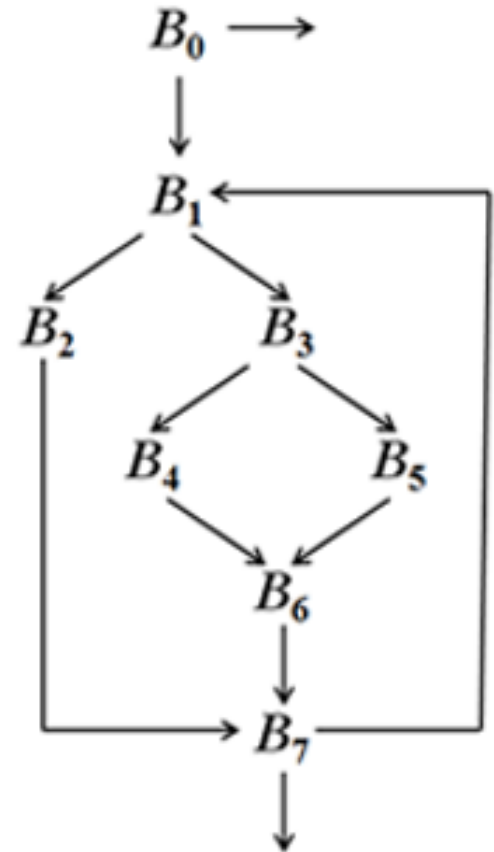
$B_3 \notin (Dom(B_7) - \{B_7\}),$

т.е. B_3 является доминатором B_4, B_5 и B_6 ,
но не является доминатором B_7 .

Более того, на любом пути, выходящем из B_3 ,
 B_7 – первая вершина, для которой
 B_3 не является доминатором

Следовательно, $B_7 \in DF(B_3)$

а так как узел B_3 не является доминатором
узлов B_0, B_1 и B_2 , то **$DF(B_3) = \{B_7\}$**



4.2 Граница доминирования

4.2.2. Построение границы доминирования

- ◇ Свойства узлов, входящих в границу доминирования
- (1) узел, принадлежащий границе доминирования должен быть точкой сбора графа потока.
 - (2) если j – точка сбора: $n \in Pred(j) \ \& \ n \notin Dom(j)$,
то $j \in DF(n)$
т.е. точка сбора j входит в границу доминирования любого своего предшественника n , не являющегося доминатором j .
 - (3) если j – точка сбора:
 $m \in Pred(j) \ \& \ n \in Dom(m) \ \& \ n \notin Dom(j)$, то $j \in DF(n)$
т.е. доминаторы предшественников точки сбора j должны иметь j в своих множествах границ доминирования, если только они не доминируют над j .

4.2 Граница доминирования

4.2.2. Построение границы доминирования

- ◇ Свойства узлов границы доминирования позволяют составить простой алгоритм ее построения

- Шаг 1. Найти все точки сбора j графа потока, т.е. все узлы j , у которых $|Pred(j)| > 1$.
- Шаг 2. Исследовать каждый узел $p \in Pred(j)$ и продвинуться по дереву доминаторов, начиная с p и вплоть до непосредственного доминатора j : при этом j входит в состав границы доминирования каждого из пройденных узлов, за исключением непосредственного доминатора j .

4.2 Граница доминирования

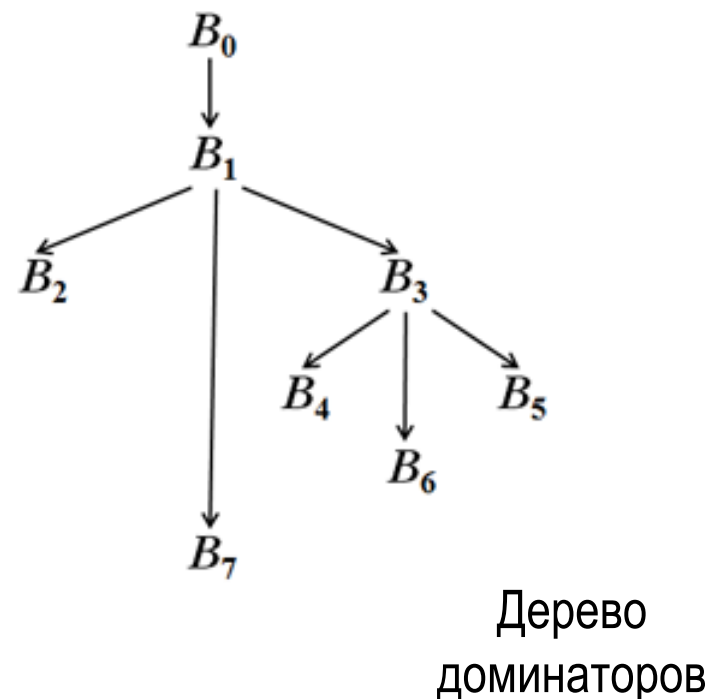
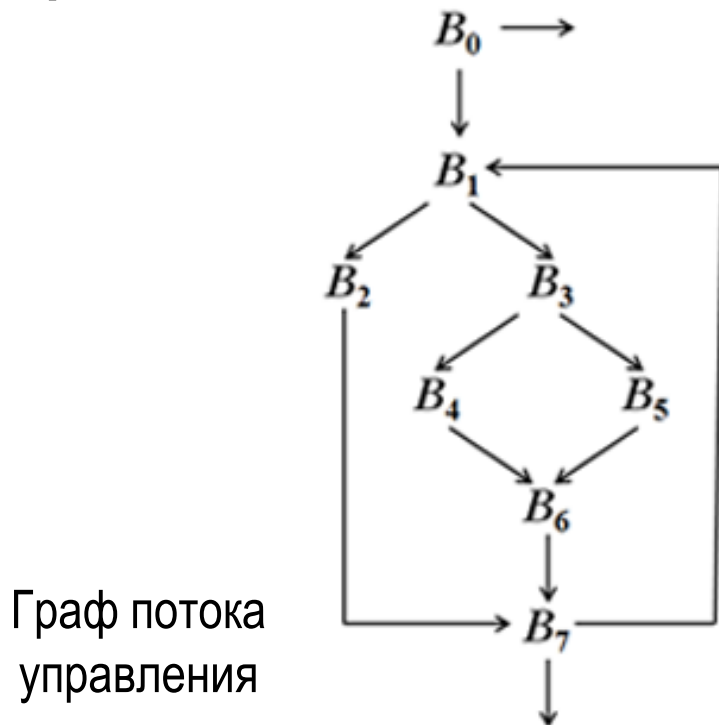
4.2.3. Алгоритм построения границ доминирования

- ◇ **Вход:** граф потока
- ◇ **Выход:** множество границ доминирования для узлов графа потока
- ◇ **Метод:** выполнить следующую программу:

```
for all  $n \in N$  do  $DF(n) = \emptyset$ ;  
for all  $n \in N$  do  
    {if  $|Pred(n)| > 1$  then  
        for each  $p \in Pred(n)$  do  
             $\{r = p$ ;  
                while  $r \neq IDom(n)$  do  
                     $\{DF(r) = DF(r) \cup \{n\}$ ;  
                         $r = Idom(r)$ ;  
                    }  
            };  
    }
```

4.2 Граница доминирования

4.2.4. Пример применения алгоритма построения границ доминирования



n	0	1	2	3	4	5	6	7
$Pred(n)$	$\emptyset$	$\{0, 7\}$	$\{1\}$	$\{1\}$	$\{3\}$	$\{3\}$	$\{4, 5\}$	$\{2, 6\}$
$Dom(n)$	$\{0\}$	$\{0, 1\}$	$\{0, 1, 2\}$	$\{0, 1, 3\}$	$\{0, 1, 3, 4\}$	$\{0, 1, 3, 5\}$	$\{0, 1, 3, 6\}$	$\{0, 1, 7\}$
$Idom(n)$	$\emptyset$	$\{0\}$	$\{1\}$	$\{1\}$	$\{3\}$	$\{3\}$	$\{3\}$	$\{1\}$

4.2 Граница доминирования

4.2.4. Пример применения алгоритма построения границ доминирования

◇ У графа три точки сбора – входы в узлы B_1 , B_6 и B_7 .

Узел B_6 : $Pred(B_6) = \{B_4, B_5\}$, $Idom(B_6) = \{B_3\}$,

проходим от B_5 до B_3 , добавляем B_6 к $DF(B_5)$,

проходим от B_4 до B_3 , добавляем B_6 к $DF(B_4)$.

Узел B_7 : $Pred(B_7) = \{B_2, B_6\}$, $Idom(B_7) = \{B_1\}$,

проходим от B_2 до B_1 , добавляем B_7 к $DF(B_2)$,

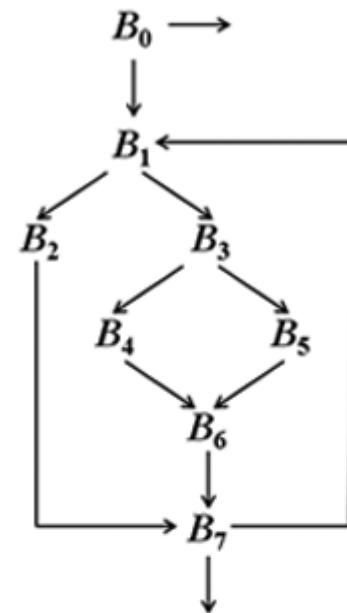
проходим от B_6 до B_3 , добавляем B_7 к $DF(B_6)$

проходим от B_3 до B_1 , добавляем B_7 к $DF(B_3)$

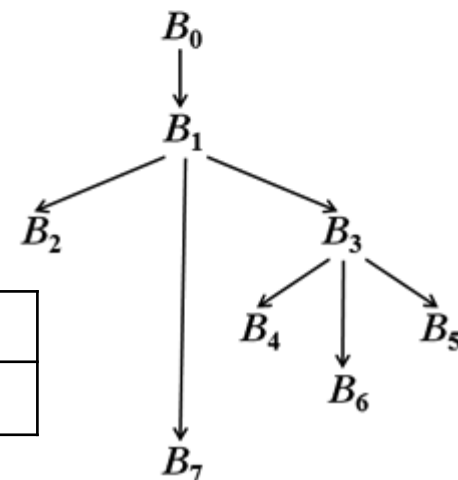
◇ Таблица текущих результатов:

n	0	1	2	3	4	5	6	7
$DF(B_n)$	$\emptyset$	$\emptyset$	$\{B_7\}$	$\{B_7\}$	$\{B_6\}$	$\{B_6\}$	$\{B_7\}$	

Граф потока управления



Дерево доминаторов



4.2 Граница доминирования

4.2.4. Пример применения алгоритма построения границ доминирования

◇ Узел B_1 : $Pred(B_1) = \{B_0, B_7\}$, $Idom(B_1) = \{B_0\}$,

У B_0 нет непосредственного доминатора:

$$Idom(B_0) = \emptyset,$$

значит $B_1 \notin DF(B_0)$.

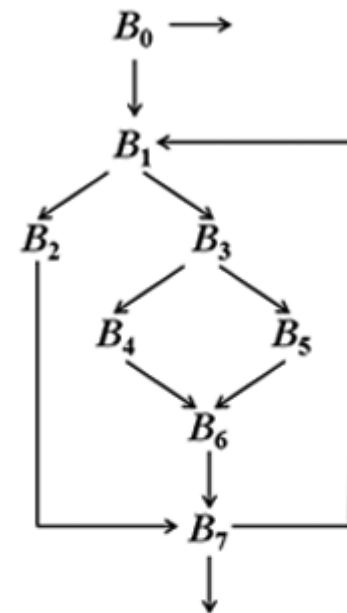
проходим от B_7 до B_1 (по обратному ребру),

добавляем B_1 к $DF(B_7)$.

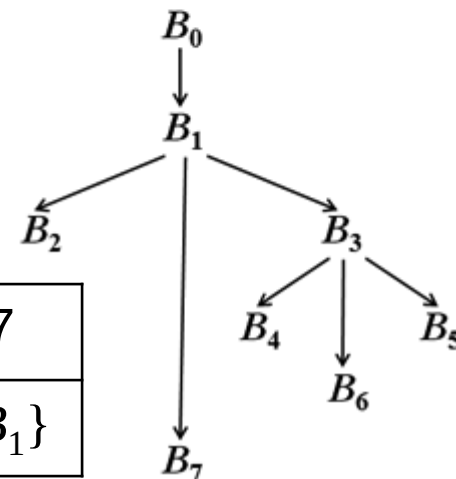
◇ Окончательная таблица результатов

n	0	1	2	3	4	5	6	7
$DF(B_n)$	$\emptyset$	$\{B_7\}$	$\{B_7\}$	$\{B_7\}$	$\{B_6\}$	$\{B_6\}$	$\{B_7\}$	$\{B_1\}$

Граф потока управления



Дерево доминаторов



4.3 Постдоминаторы

4.3.1 Определение

- ◇ В ГПУ вершина p является *постдоминатором* вершины n (этот факт записывается как $p \text{ postdom } n$ или $p = \text{Postdom}(n)$), если любой путь от вершины n до вершины Exit проходит через вершину d .
- ◇ **Замечание.** Из определения 4.2.1 следует, что каждая вершина n является постдоминатором самой себя: путь от n до Exit проходит через n .

4.3 Постдоминаторы

4.3.2 Определения

- ◇ *Обратным графом* ориентированного графа $G = \langle N, E \rangle$ называется ориентированный граф $G^R = \langle N, E^R \rangle$, у которого направления всех ребер противоположны.
- ◇ **Постдоминаторы ГПУ – это доминаторы его *обратного графа*.**
- ◇ *Обратная граница доминирования* ($RDF(n)$) вершины $n \in G$ это обычная граница доминирования в обратном графе G^R .

4.3 Постдоминаторы

4.3.3 Применение постдоминаторов. Зависимость по управлению.

- ◇ По определению, вершина m ГПУ *зависит по управлению* от вершины n тогда, и только тогда, когда:
 - ◇ существует непустой путь T от n до m , такой что $\forall k \in T - \{n\}: m = \text{Postdom}(k)$, т.е. если выполнение программы пошло по пути T , то, чтобы достичь *exit*, оно обязательно пройдет через m .
 - ◇ m не обязательно является строгим постдоминатором n : у n может быть несколько выходов, так что помимо T возможны и другие пути, проходящие через n , но потом ведущие не в m , а в другие вершины.
- ◇ Обратная граница доминирования позволяет определять границы зависимостей по управлению.

4.3 Постдоминаторы

4.3.4 Эквивалентность по управлению

- ◇ **Определение.** Два базовых блока B_i и B_j эквивалентны по управлению, если B_i выполняется тогда, и только тогда, когда выполняется B_j .
- ◇ **Утверждение.** Если выполняются соотношения:
$$B_i = \text{Dom}(B_j) \text{ и } B_j = \text{Postdom}(B_i)$$
то базовые блоки B_i и B_j эквивалентны по управлению

5. Машинно-независимая оптимизация

5.1. Простые оптимизации

5.1.1 Сворачивание констант

- ◇ **Сворачивание констант**, или вычисление константных выражений — это вычисление выражений, все операнды которых — константы, значения которых известны во время компиляции и подстановка свернутых констант в выражения, операндами которых они являются.
- ◇ *Оптимизация состоит в том, что часть вычислений выполняется во время компиляции и убирается из программы.*
- ◇ **Основная проблема** — добиться, чтобы все операции выполнялись точно так же, как они выполнялись бы во время выполнения программы.
Прежде всего это связано с *исключительными ситуациями*.

5.1. Простые оптимизации

5.1.1 Сворачивание констант

- ◇ В случае *булевских вычислений* исключительные ситуации не возбуждаются и потому проблем не возникает.
- ◇ В случае вычисления *целых констант* для некоторых исходных данных компилируемой программы в процессе вычисления может получиться «*деление на ноль*», или «*переполнение*». В этом случае компилятор должен выдать пользователю соответствующее *сообщение об ошибке, либо предупреждение*.
- ◇ Наибольшее количество проблем, естественно, возникает, если сворачивается константа одного из плавающих типов.

5.1. Простые оптимизации

5.1.2 Алгебраические упрощения и перегруппировка

◇ Применение тождеств (i и j - переменные типа *int*):

$$i + 0 = 0 + i = i - 0 = i$$

$$0 - i = -i$$

$$i * 1 = 1 * i = i / 1 = i$$

$$i * 0 = 0 * i = 0$$

$$-(-i) = i$$

$$i + (-j) = i - j$$

◇ Применение тождеств (b - переменная типа *Boolean*):

$$b \vee true = true \vee b = true$$

$$b \vee false = false \vee b = b$$

$$b \& true = true \& b = b$$

$$b \& false = false \& b = false$$

5.1. Простые оптимизации

5.1.3 Распространение копий

- ◇ Пусть в оптимизируемой процедуре есть инструкция копирования

$$\mathbf{x} \leftarrow \mathbf{y}$$

Распространение копий означает замену всех последующих вхождений переменной $\mathbf{x}$ на переменную $\mathbf{y}$.

- ◇ Рассмотрим множество всех команд копирования анализируемой процедуры. Каждая команда копирования описывается четверкой $\langle \mathbf{x}, \mathbf{y}, b, p \rangle$, где $\mathbf{x}$ и $\mathbf{y}$ представляют инструкцию копирования

$\mathbf{x} \leftarrow \mathbf{y}$, находящуюся в строке p базового блока b .

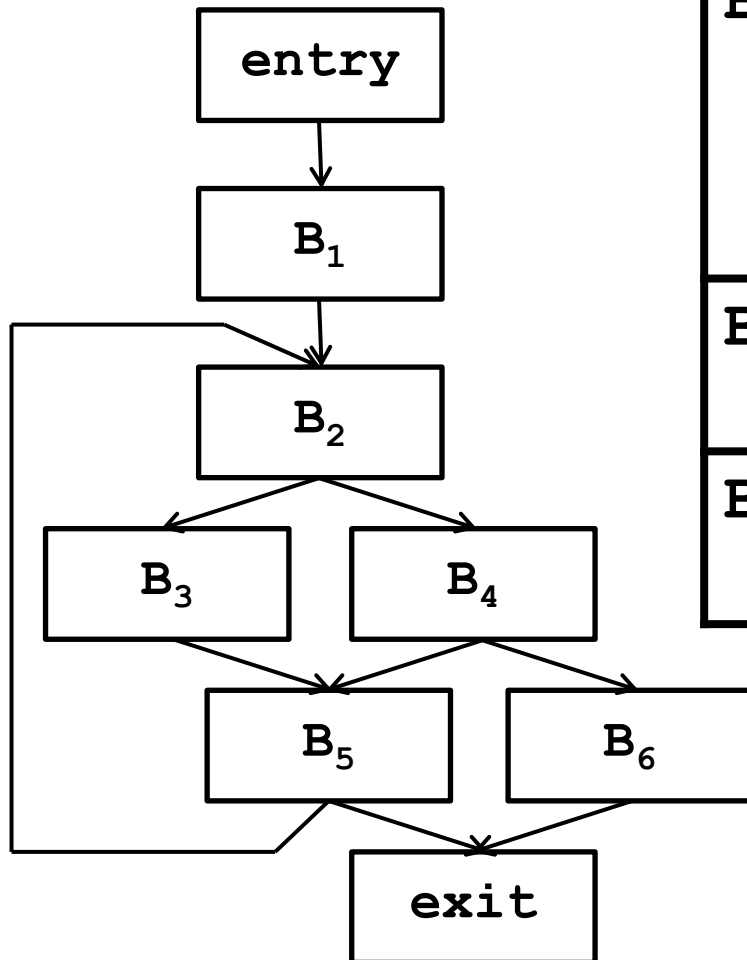
Множество всех таких четверок обозначим через U .

Множество U содержит все инструкции копирования анализируемой процедуры.

5.1. Простые оптимизации

5.1.3 Распространение копий

- ◇ В процедуре, ГПУ которой представлен на рисунке две команды копирования

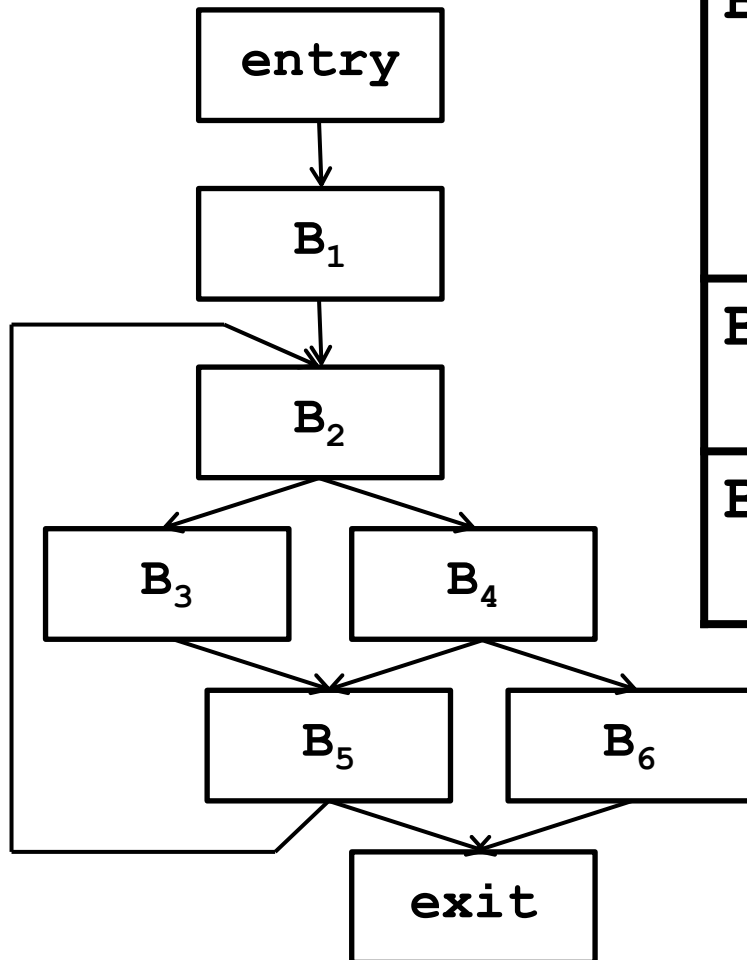


B_1 $c \leftarrow +, a, b$ $d \leftarrow c$ $e \leftarrow *, d, d$	B_2 $f \leftarrow +, a, c$ $g \leftarrow e$ $a \leftarrow +, g, d$ $\text{if}(a < c) \text{goto}$
B_3 $h \leftarrow +, g, 1$	B_4 $f \leftarrow -, d, g$ $\text{if}(f > a) \text{goto}$
B_5 $b \leftarrow *, g, a$ $\text{if}(f > h) \text{goto}$	B_6 $c \leftarrow 2$

5.1. Простые оптимизации

5.1.3 Распространение копий

- ◇ В процедуре, ГПУ которой представлен на рисунке, две команды копирования



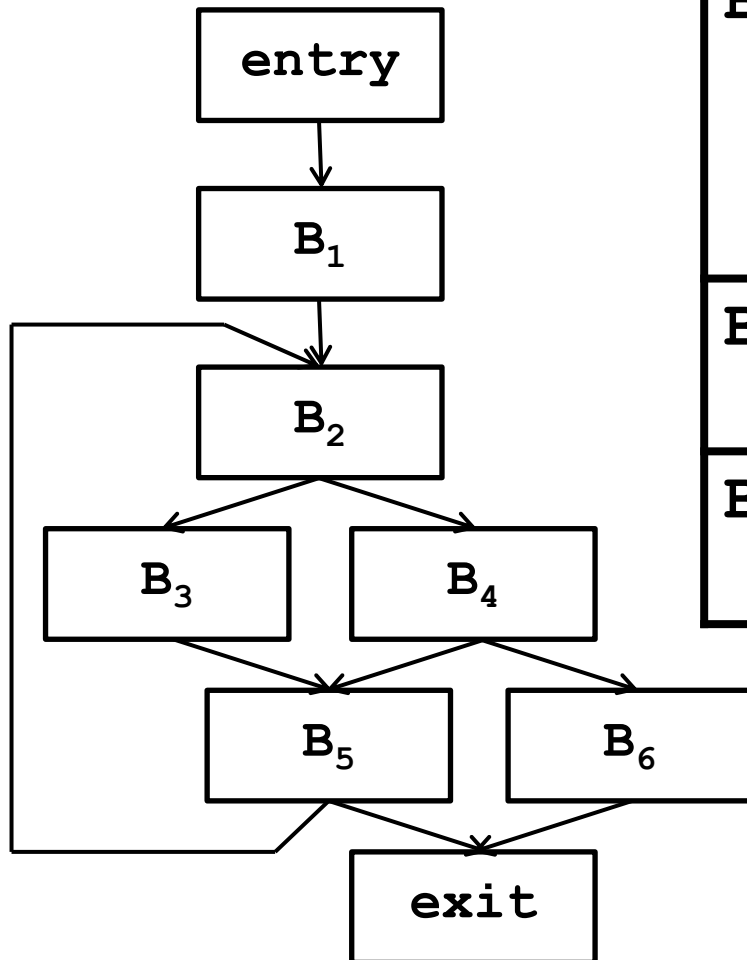
B_1 $c \leftarrow +, a, b$ $d \leftarrow c$ $e \leftarrow *, d, d$	B_2 $f \leftarrow +, a, c$ $g \leftarrow e$ $a \leftarrow +, g, d$ $\text{if}(a < c) \text{goto}$
B_3 $h \leftarrow +, g, 1$	B_4 $f \leftarrow -, d, g$ $\text{if}(f > a) \text{goto}$
B_5 $b \leftarrow *, g, a$ $\text{if}(f > h) \text{goto}$	B_6 $c \leftarrow 2$

- ◇ $d \leftarrow c$ (2-я строка блока B_1):
четверка $\langle d, c, B_1, 2 \rangle$
- ◇ $g \leftarrow e$ (2-я строка блока B_2):
четверка $\langle g, e, B_2, 2 \rangle_7$

5.1. Простые оптимизации

5.1.3 Распространение копий

- ◇ В процедуре, ГПУ которой представлен на рисунке две инструкции копирования



B_1 $c \leftarrow +, a, b$ $d \leftarrow c$ $e \leftarrow *, d, d$	B_2 $f \leftarrow +, a, c$ $g \leftarrow e$ $a \leftarrow +, g, d$ $\text{if}(a < c) \text{goto}$
B_3 $h \leftarrow +, g, 1$	B_4 $f \leftarrow -, d, g$ $\text{if}(f > a) \text{goto}$
B_5 $b \leftarrow *, g, a$ $\text{if}(f > h) \text{goto}$	B_6 $c \leftarrow 2$

- ◇ Следовательно множество U инструкций копирования равно

$$U = \{\langle d, c, B_1, 2 \rangle, \langle g, e, B_2, 2 \rangle\}$$

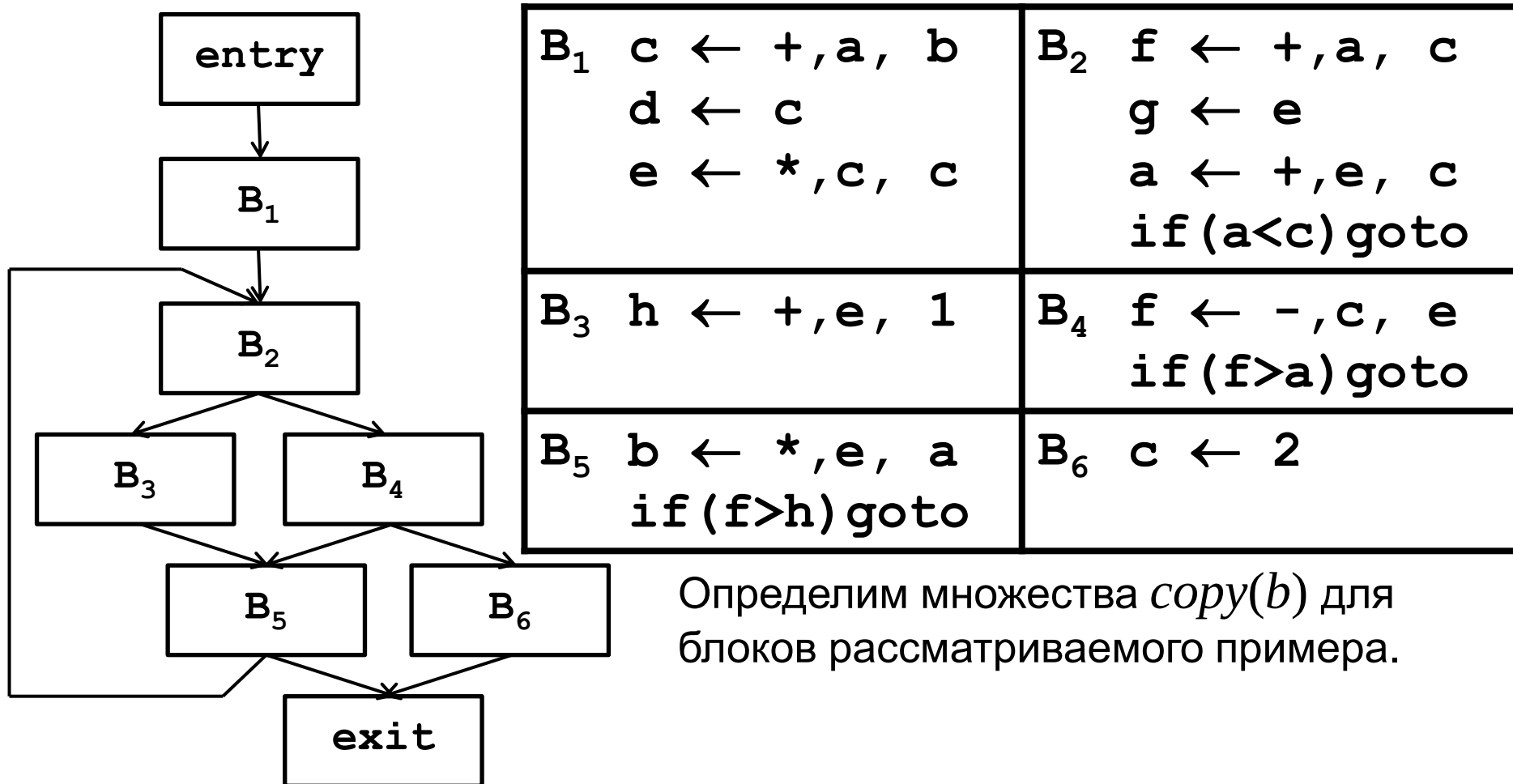
5.1. Простые оптимизации

5.1.3 Распространение копий

- ◇ Для каждого базового блока b определим
 - ◇ множество $copy(b)$ команд копирования (четверок $\langle \mathbf{x}, \mathbf{y}, b, p \rangle$), содержащихся в блоке b
 - ◇ множество $kill(b)$ переопределений $\mathbf{y}$ (четверок $\langle \mathbf{x}, \mathbf{y}, b, p \rangle$), убиваемых в блоке b

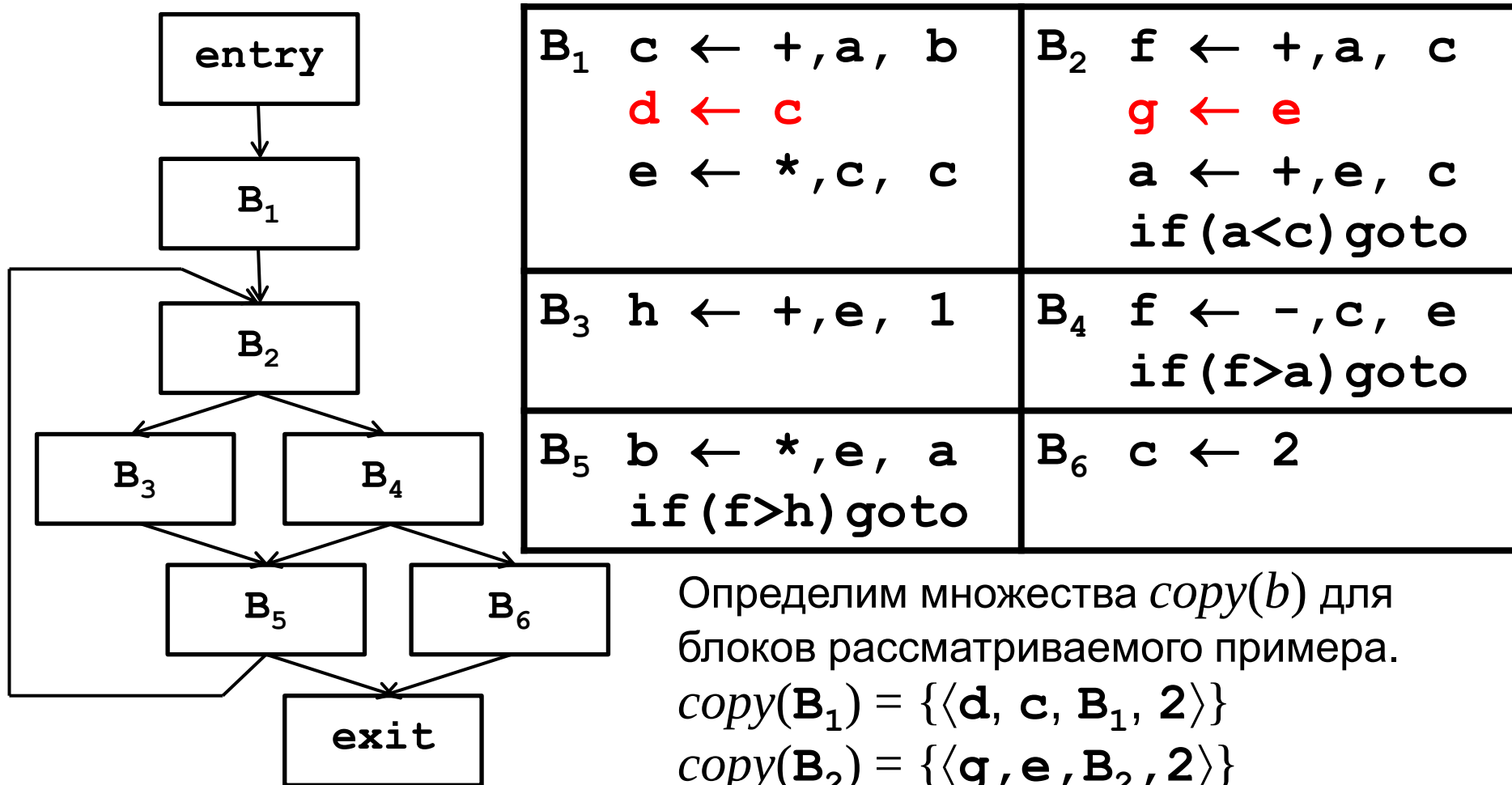
5.1. Простые оптимизации

5.1.3 Распространение копий



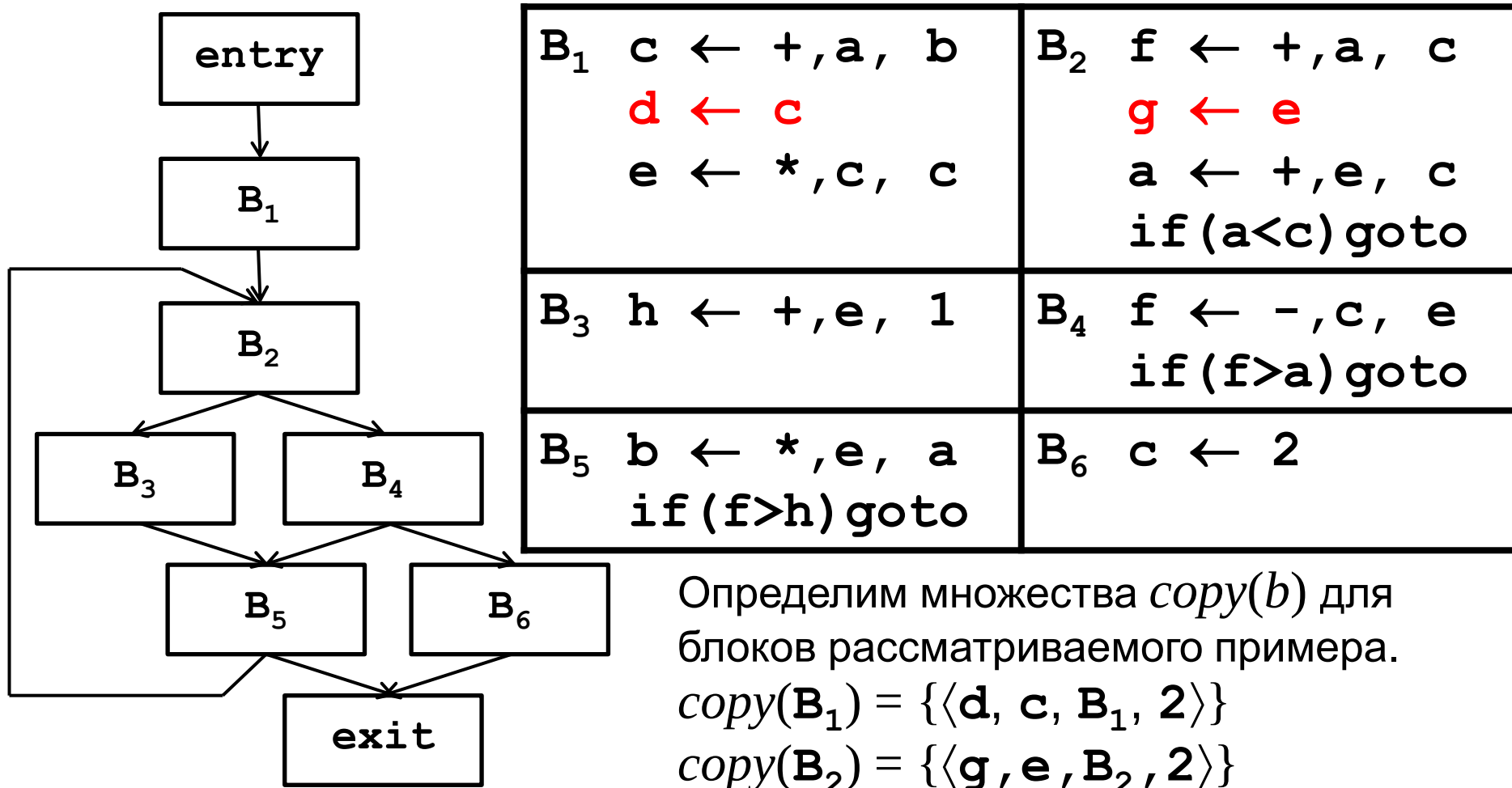
5.1. Простые оптимизации

5.1.3 Распространение копий



5.1. Простые оптимизации

5.1.3 Распространение копий

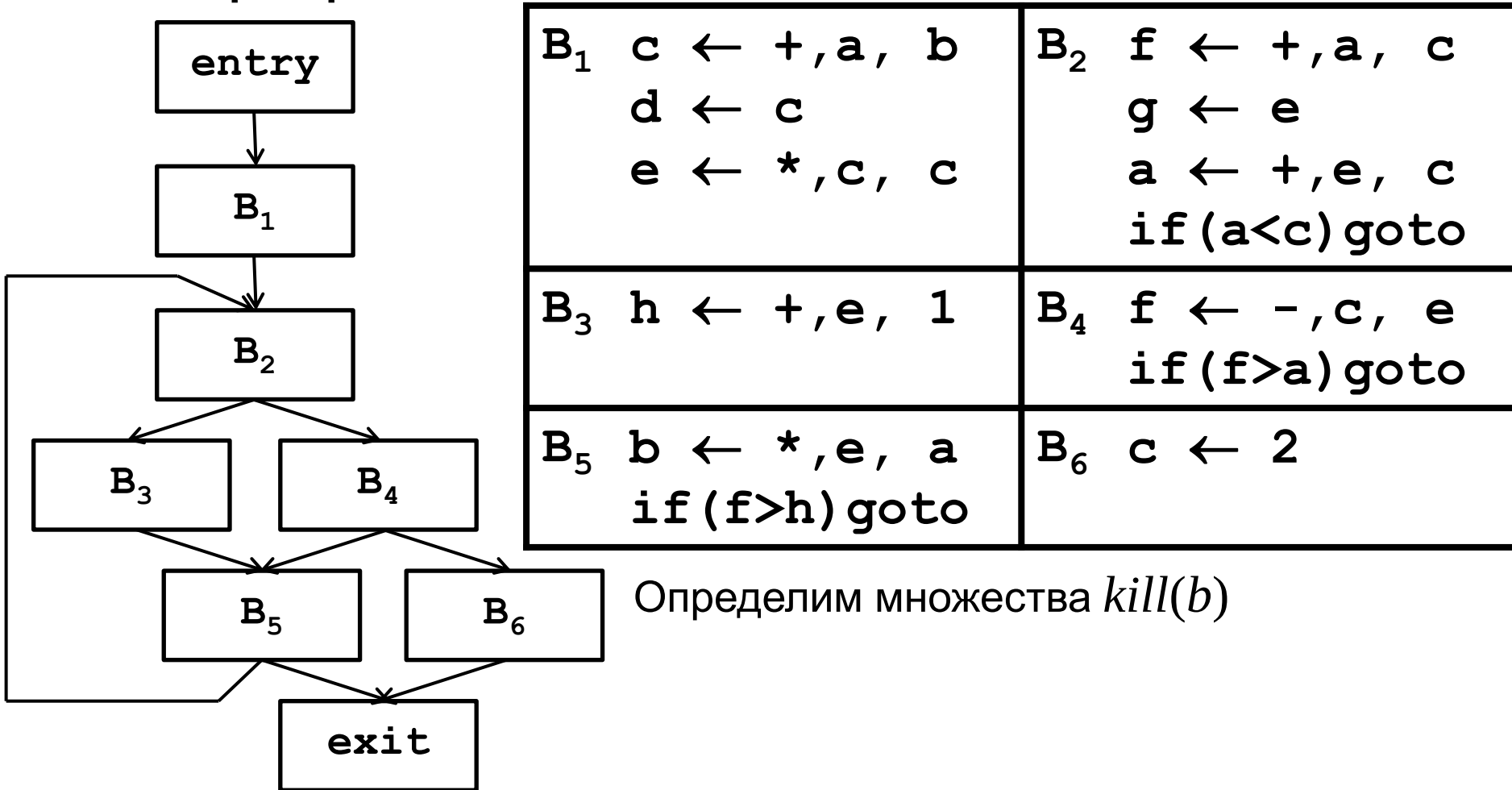


Остальные блоки не содержат инструкций копирования, поэтому для этих блоков множества $\text{copy}(b)$ пустые:

$$\text{copy}(\mathbf{B}_3) = \emptyset, \text{copy}(\mathbf{B}_4) = \emptyset, \text{copy}(\mathbf{B}_5) = \emptyset, \text{copy}(\mathbf{B}_6) = \emptyset$$

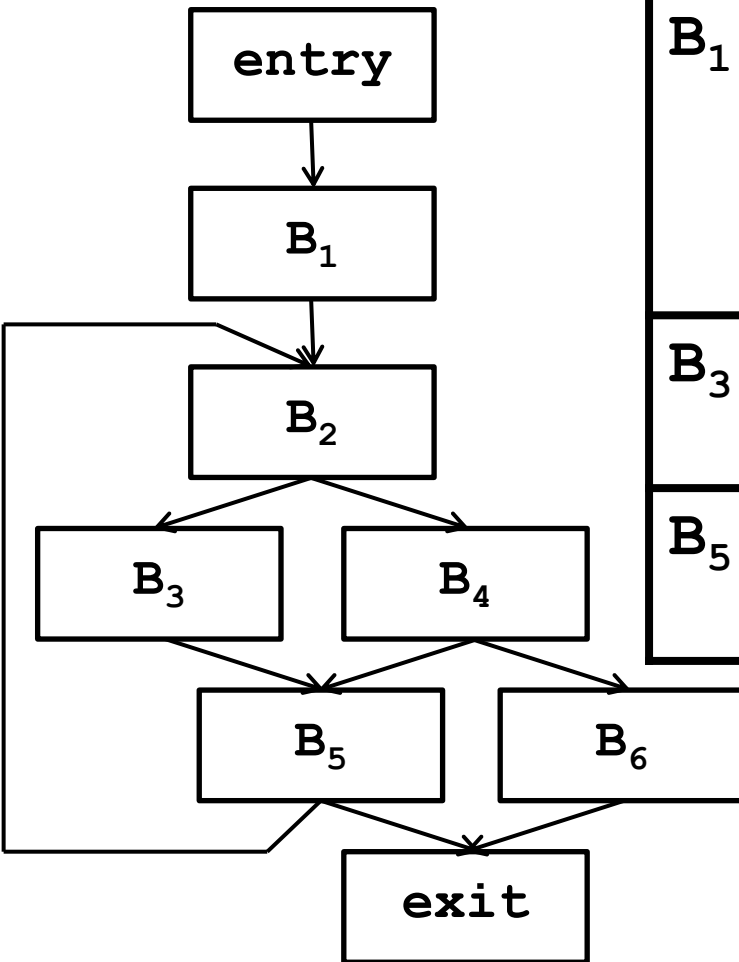
5.1. Простые оптимизации

5.1.3 Распространение копий



5.1. Простые оптимизации

5.1.3 Распространение копий



B_1 $c \leftarrow +, a, b$ $d \leftarrow c$ $e \leftarrow *, c, c$	B_2 $f \leftarrow +, a, c$ $g \leftarrow e$ $a \leftarrow +, e, c$ $\text{if}(a < c) \text{goto}$
B_3 $h \leftarrow +, e, 1$	B_4 $f \leftarrow -, c, e$ $\text{if}(f > a) \text{goto}$
B_5 $b \leftarrow *, e, a$ $\text{if}(f > h) \text{goto}$	B_6 $c \leftarrow 2$

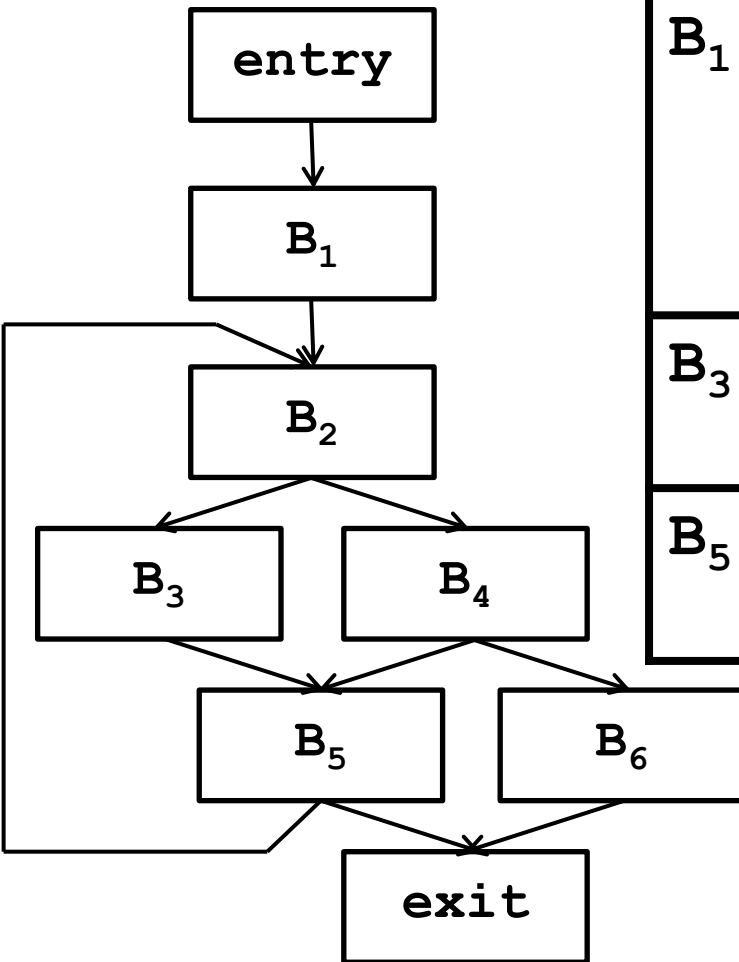
Определим множества $kill(b)$

$\langle g, e, B_2, 2 \rangle$ убивается в блоке **B₁**, так как в 3-ей строке этого блока переопределяется **e**. Следовательно $kill(B_1) = \{ \langle g, e, B_2, 2 \rangle \}$

$\langle d, c, B_1, 2 \rangle$ убивается в блоке **B₆**, так как единственная инструкция этого блока переопределяет **c**. Следовательно $kill(B_6) = \{ \langle d, c, B_1, 2 \rangle \}$

5.1. Простые оптимизации

5.1.3 Распространение копий



B_1	$c \leftarrow +, a, b$ $d \leftarrow c$ $e \leftarrow *, c, c$	B_2	$f \leftarrow +, a, c$ $g \leftarrow e$ $a \leftarrow +, e, c$ $\text{if}(a < c) \text{goto}$
B_3	$h \leftarrow +, e, 1$	B_4	$f \leftarrow -, c, e$ $\text{if}(f > a) \text{goto}$
B_5	$b \leftarrow *, e, a$ $\text{if}(f > h) \text{goto}$	B_6	$c \leftarrow 2$

Определим множества $kill(b)$

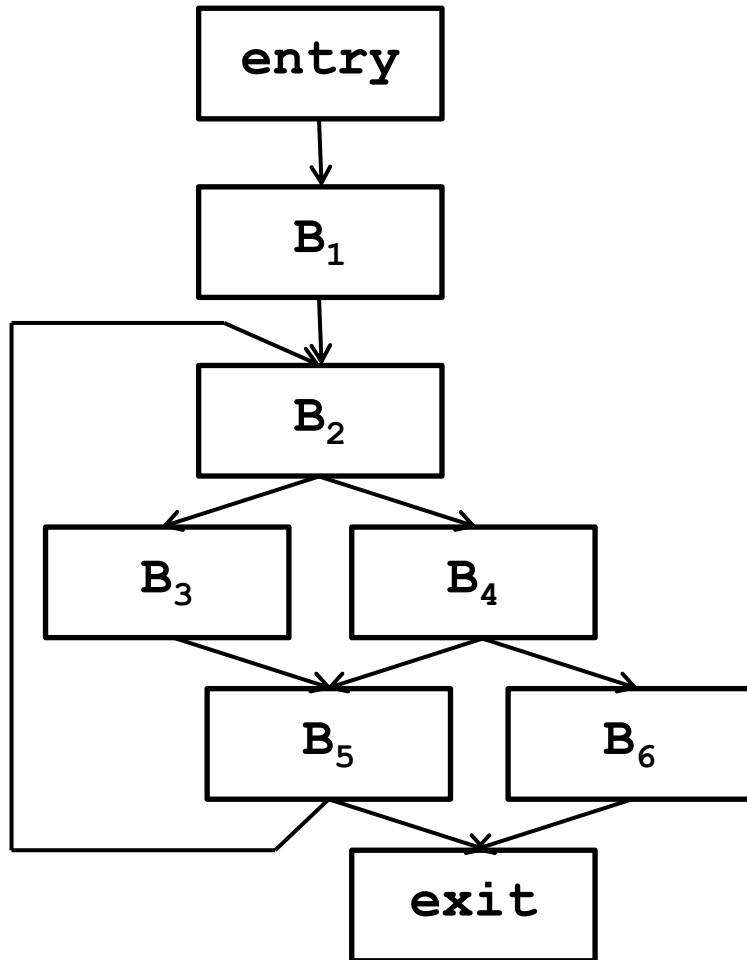
$\langle g, e, B_2, 2 \rangle$ убивается в блоке **B₁**, так как в 3-ей строке этого блока переопределяется **e**. Следовательно $kill(B_1) = \{ \langle g, e, B_2, 2 \rangle \}$

$\langle d, c, B_1, 2 \rangle$ убивается в блоке **B₆**, так как единственная инструкция этого блока переопределяет **c**. Следовательно $kill(B_6) = \{ \langle d, c, B_1, 2 \rangle \}$

Множества $kill(b)$ остальных блоков пустые.

5.1. Простые оптимизации

5.1.3 Распространение копий



Таким образом для базовых блоков рассматриваемой процедуры множества $copy(b)$ и $kill(b)$ следующие:

b	copy (b)	kill (b)
entry	$\emptyset$	$\emptyset$
B ₁	$\{\langle d, c, B_1, 2 \rangle\}$	$\{\langle g, e, B_2, 2 \rangle\}$
B ₂	$\{\langle g, e, B_2, 2 \rangle\}$	$\emptyset$
B ₃	$\emptyset$	$\emptyset$
B ₄	$\emptyset$	$\emptyset$
B ₅	$\emptyset$	$\emptyset$
B ₆	$\emptyset$	$\{\langle d, c, B_1, 2 \rangle\}$
exit	$\emptyset$	$\emptyset$

5.1. Простые оптимизации

5.1.3 Распространение копий

◇ Система уравнений составляется по аналогии с системой уравнений для достигающих определений:

◇ в передаточной функции сначала из множества инструкций копирования удаляются инструкции «убитые» в блоке b , потом в него добавляются инструкции копирования блока b

$$Out_{CP}(b) = copy(b) \cup (In_{CP}(b) - kill(b))$$

◇ подставив $Out_{CP}(b)$ в уравнение сбора по всем путям, (оно в отличие от соответствующего уравнения для достигающих содержит операцию пересечения, а не объединения, так как **по всем путям должны приходить одинаковые копии**)

$$In_{CP}(b) = \bigcap_{p \in Pred(b)} Out_{CP}(p)$$

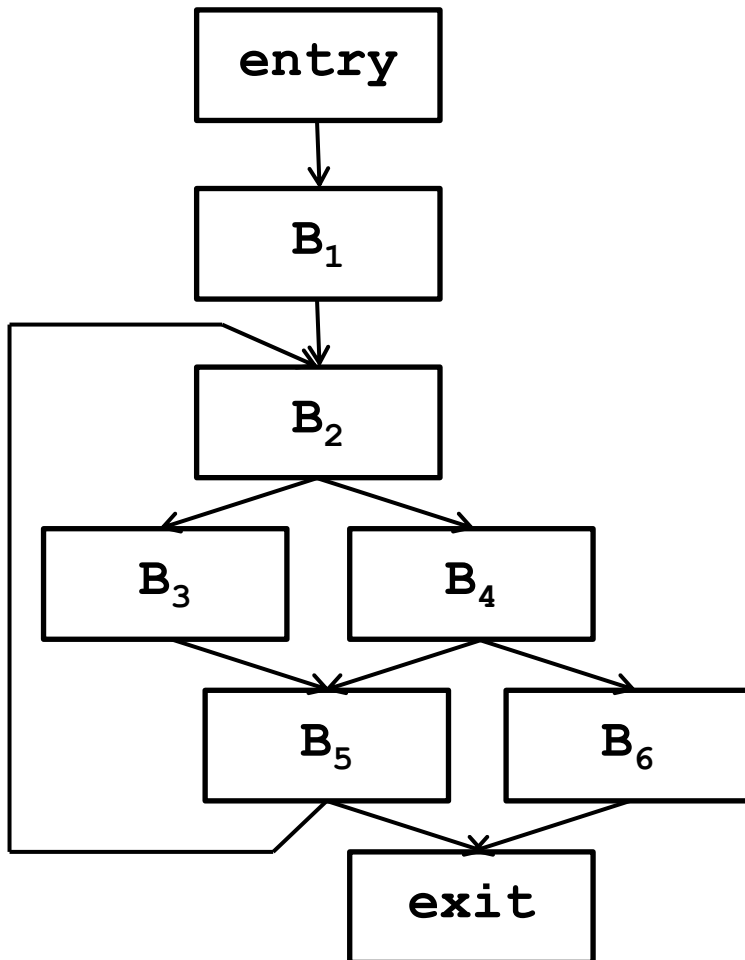
получим

$$In_{CP}(b) = \bigcap_{p \in Pred(b)} (copy(p) \cup (In_{CP}(p) - kill(p)))$$

5.1. Простые оптимизации

5.1.4 Распространение копий

- ◇ Решив систему уравнений для рассматриваемого примера методом итераций, получим следующие значения $\mathbf{In}_{CP}$



b	$\mathbf{In}_{CP}(b)$
Entry	$\emptyset$
B_1	$\emptyset$
B_2	$\{\langle d, c, B_1, 2 \rangle\}$
B_3	$\{\langle d, c, B_1, 2 \rangle, \langle g, e, B_2, 2 \rangle\}$
B_4	$\{\langle d, c, B_1, 2 \rangle, \langle g, e, B_2, 2 \rangle\}$
B_5	$\{\langle d, c, B_1, 2 \rangle, \langle g, e, B_2, 2 \rangle\}$
B_6	
exit	$\{\langle g, e, B_2, 2 \rangle\}$

5.2. Оптимизация циклов

5.2.1 Классификация дуг ГПУ

- ◇ Дуги ГПУ, являющиеся дугами и его остовного дерева, называются *остовными*.
- ◇ Дуги ГПУ, не являющиеся дугами его остовного дерева, но имеющие такое же направление, что и остовные, называются *прямыми*.
- ◇ Дуги ГПУ, направленные противоположно остовным, называются *обратно направленными*.
- ◇ Обратно направленная дуга ГПУ $\langle B_i, B_k \rangle$ называется *обратной*, если $B_k = Dom(B_i)$
- ◇ Остальные дуги ГПУ называются *поперечными*. Поперечные дуги соединяют различные поддеревья остовного дерева и в программах нормальных программистов не встречаются (поперечные и обратно направленные дуги любят Фортранные программисты, но и у них они постепенно выходят из моды).

5.2. Оптимизация циклов

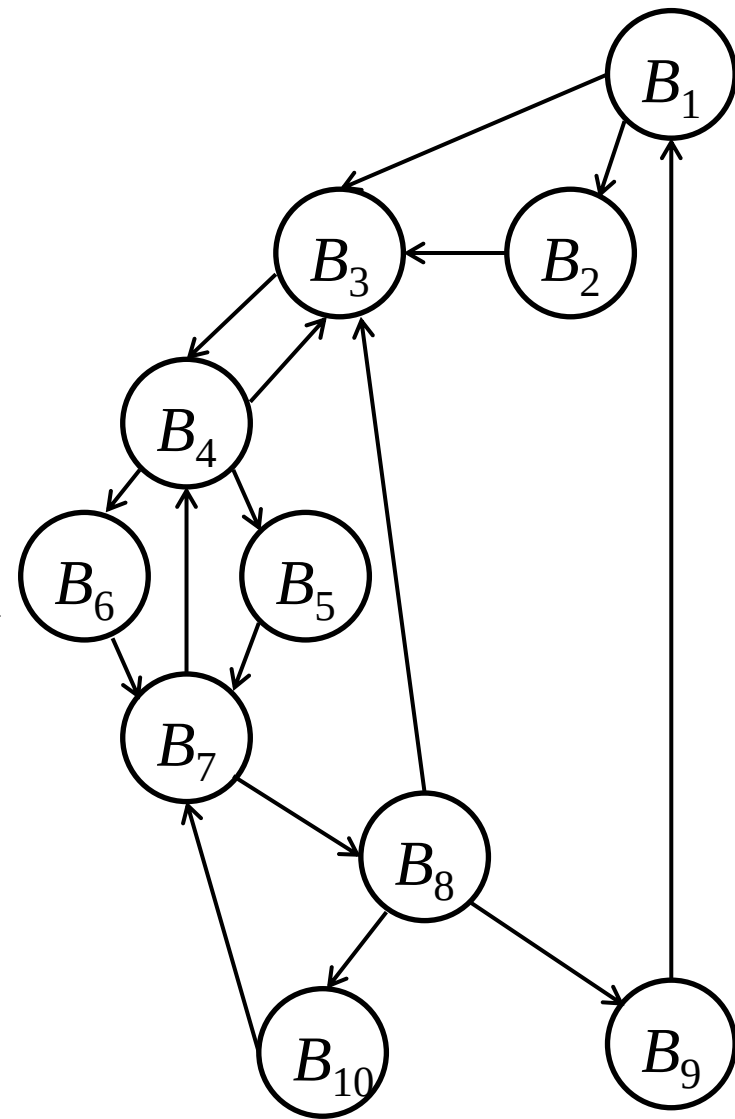
5.2.2 Естественные циклы

- ◇ **Определение.** *Естественным циклом* называется цикл со следующими свойствами:
 - ◇ Цикл имеет единственную входную вершину, называемую его *заголовком*,
 - ◇ Существует обратное ребро, ведущее в заголовок цикла
- ◇ **Определение.** *Естественный цикл обратного ребра* $\langle V_i, V_k \rangle$ составляют узел V_k (*заголовок цикла*) и все узлы ГПУ, из которых можно достичь узла V_i , не проходя через узел V_k . (эти узлы составляют *тело цикла*).

5.2. Оптимизация циклов

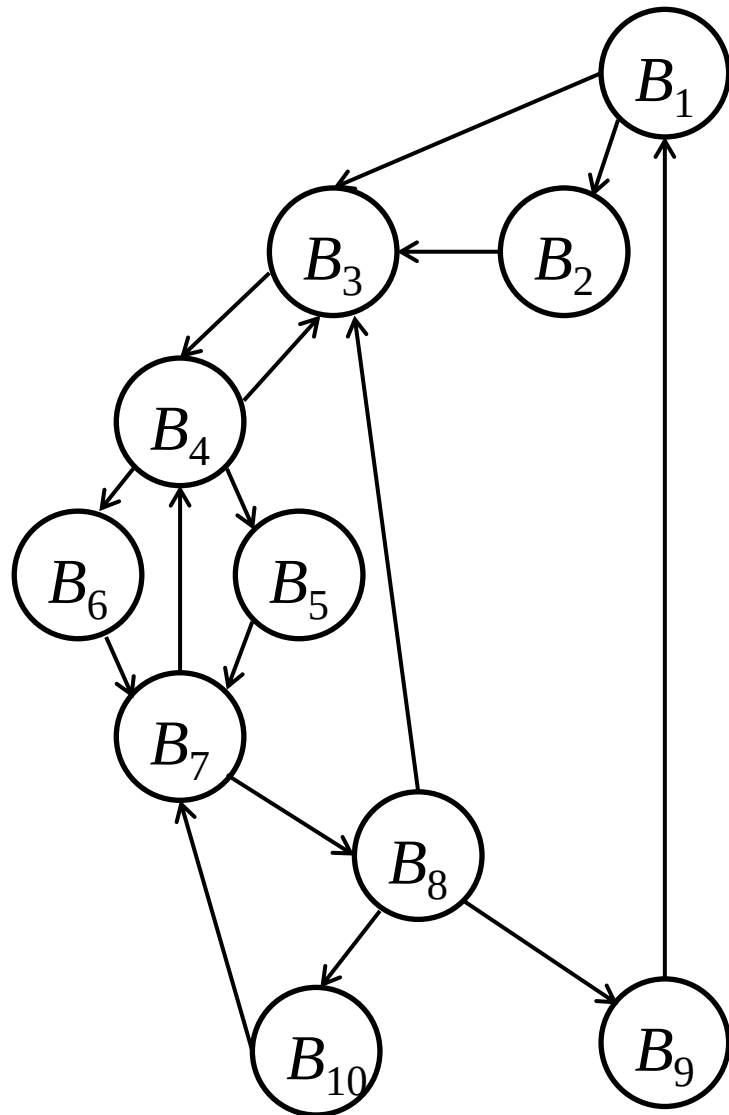
5.2.3 Естественные циклы. Пример.

- ◇ На рисунке справа – пять обратных дуг:
 $\langle B_{10}, B_7 \rangle$, $\langle B_7, B_4 \rangle$, $\langle B_4, B_3 \rangle$,
 $\langle B_8, B_3 \rangle$, $\langle B_9, B_1 \rangle$
- ◇ Обратной дуге $\langle B_{10}, B_7 \rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и B_{10} можно достичь вершины B_{10} , не проходя через B_7 .
- ◇ Обратной дуге $\langle B_7, B_4 \rangle$ соответствует естественный цикл
 $\{B_4, B_5, B_6, B_7, B_8, B_{10}\}$
Этот цикл включает в себя цикл дуги
 $\langle B_{10}, B_7 \rangle$, который является вложенным циклом



5.2. Оптимизация циклов

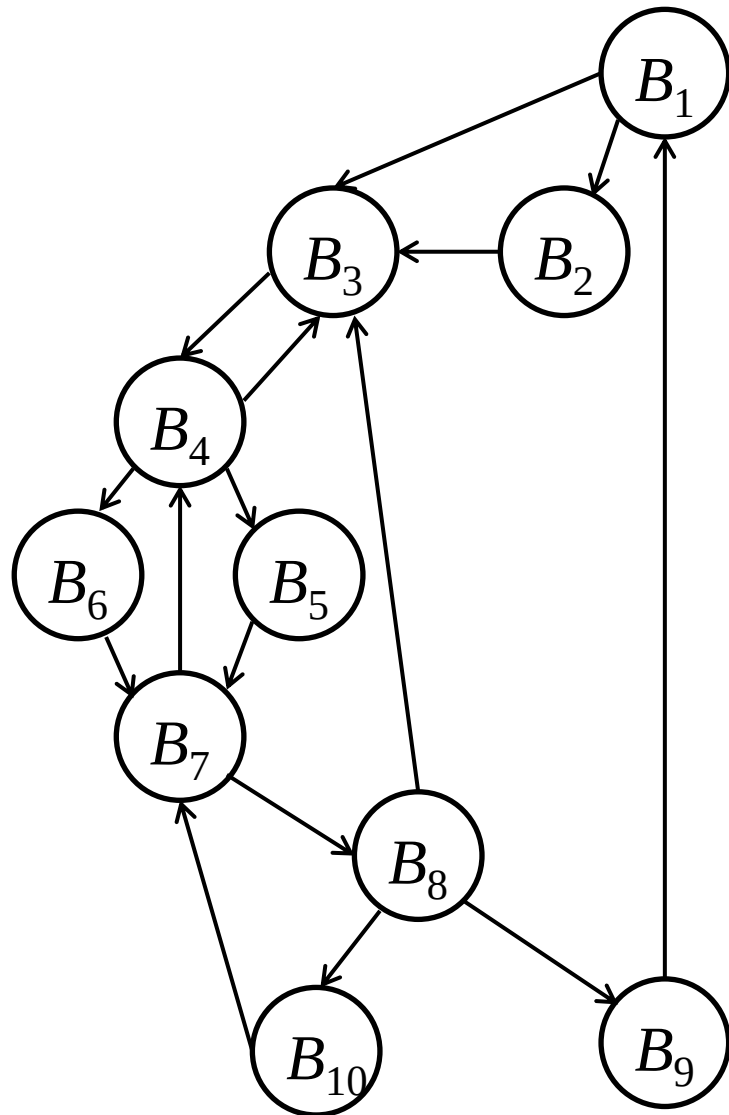
5.2.3 Естественные циклы. Пример.



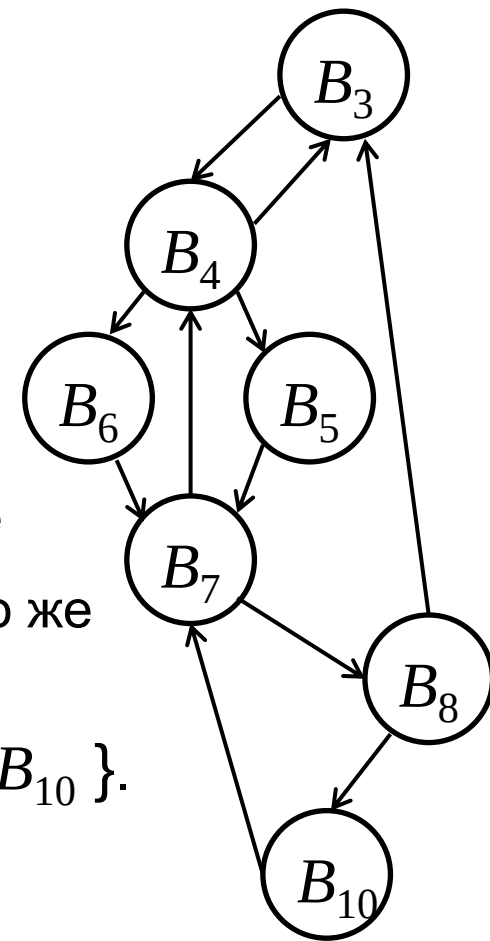
◇ У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один. В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$

5.2. Оптимизация циклов

5.2.3 Естественные циклы. Пример.



◇ У естественных циклов
 обратных дуг $\langle B_4, B_3 \rangle$
 и $\langle B_8, B_3 \rangle$ один и тот же
 заголовок B_3 и одно и то же
 множество вершин
 $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$.
 Эти два цикла можно
 объединить в один.
 В объединенный цикл
 вложены циклы обратных
 дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$



5.2. Оптимизация циклов

5.2.4 Алгоритм построения естественного цикла по обратному ребру

Вход: ГПУ $G = \langle N, E \rangle$ с входным узлом $Entry$.

Обратное ребро $e = \langle n, d \rangle \in E$

Выход: подграф $C \subseteq G$, являющийся естественным циклом.

Метод:

- (1) начальное значение C – множество $\{n, d\}$.
- (2) узел d помечается как «посещенный».
- (3) начиная с узла n выполняется поиск в глубину на обратном ГПУ (направления дуг заменены на противоположные).
- (4) все узлы, посещенные на шаге (3), добавляются в C .

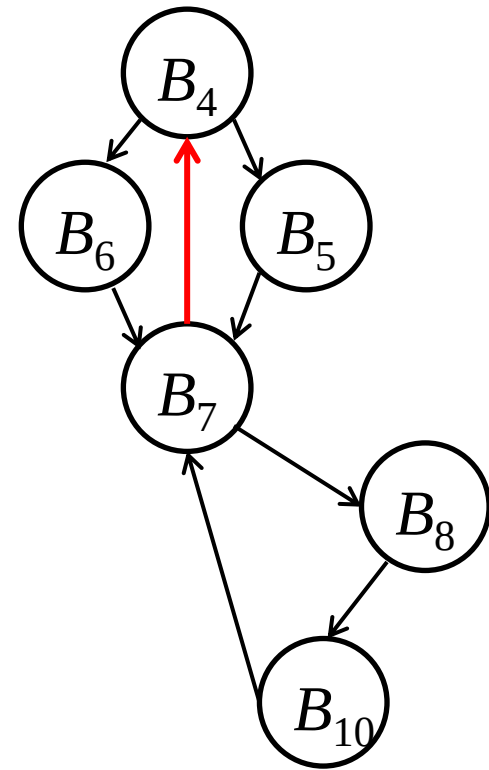
5.2. Оптимизация циклов

5.2.5 Построение естественного цикла по обратной дуге. Пример

◇ Применим алгоритм 5.2.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$.

Отметив вершину B_4 как посещенную, выполним поиск в глубину, начиная с вершины B_7 .

При этом будем считать, что на ГПУ стрелки соответствуют не концу, а началу дуги, т.е. роль множества $Succ(B_7)$ выполняет множество $Pred(B_7) = \{B_5, B_6, B_{10}\}$

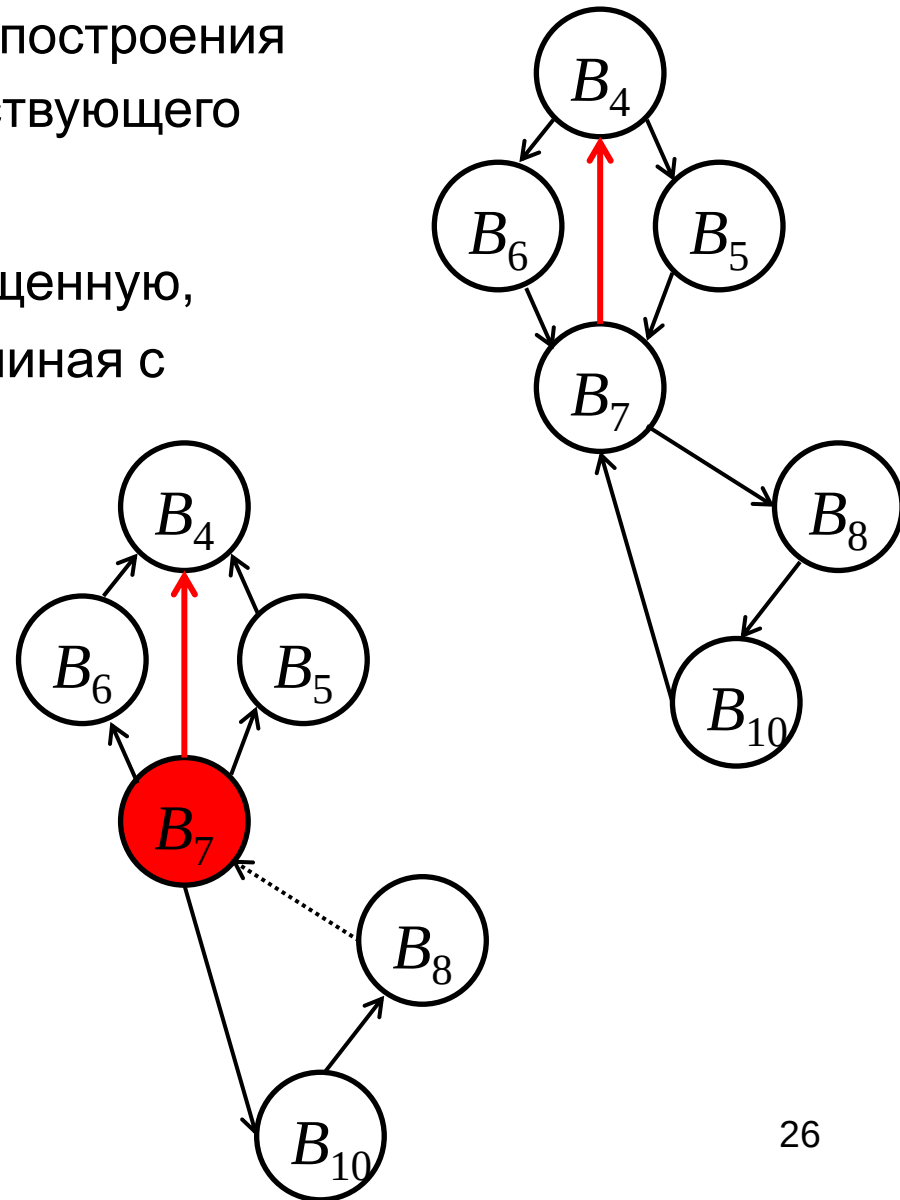


5.2. Оптимизация циклов

5.2.5 Построение естественного цикла по обратной дуге. Пример

◇ Применим алгоритм 5.2.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$.

Отметив вершину B_4 как посещенную, выполним поиск в глубину, начиная с вершины B_7 .



5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

- ◇ Инструкция *инвариантна относительно цикла*, если она удовлетворяет одному из следующих условий:
 - ◇ ее операнды – константы
 - ◇ все определения операндов, достигающие инструкции находятся вне цикла
 - ◇ внутри цикла имеется в точности одно определение операнда, но оно само инвариантно относительно цикла.

5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

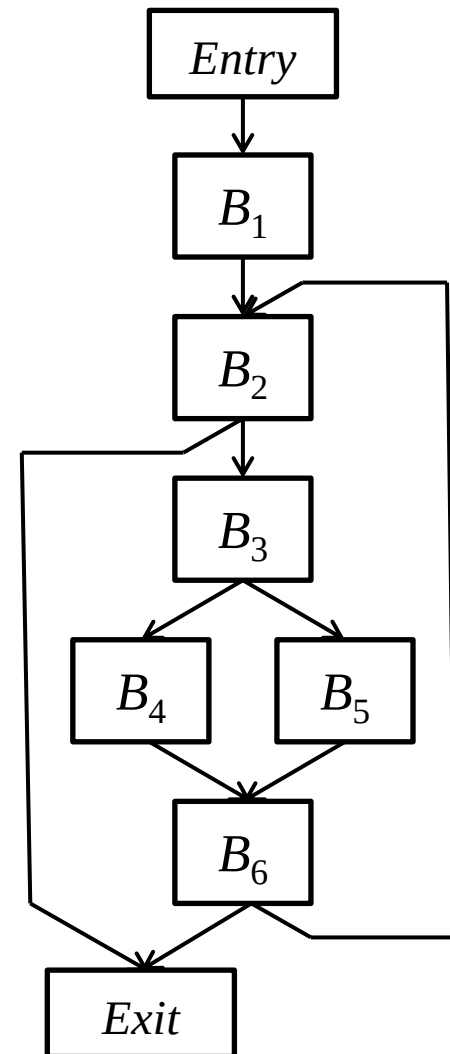
◇ Рассмотрим пример: цикл с заголовком B_2 .

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $\text{if}(i > 100)$	B_3 $a \leftarrow b + 1$ $c \leftarrow 2$ $\text{if}(i \% 2 = 0)$
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if}(a < 2)$

◇ Блок B_1 выполняется до цикла, все остальные – в цикле.

◇ Инструкции $a \leftarrow b + 1$, $c \leftarrow 2$, $a < 2$ инвариантны относительно цикла.

◇ Оптимизация состоит в том, что в ГПУ добавляется еще один блок – *предзаголовок* цикла B'_2 , в который выносятся из цикла все инвариантные инструкции.



5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

◇ **Алгоритм:**

1. Перед заголовком цикла вставить пустой базовый блок (будущий предзаголовок).

Для всех инструкций в теле цикла:

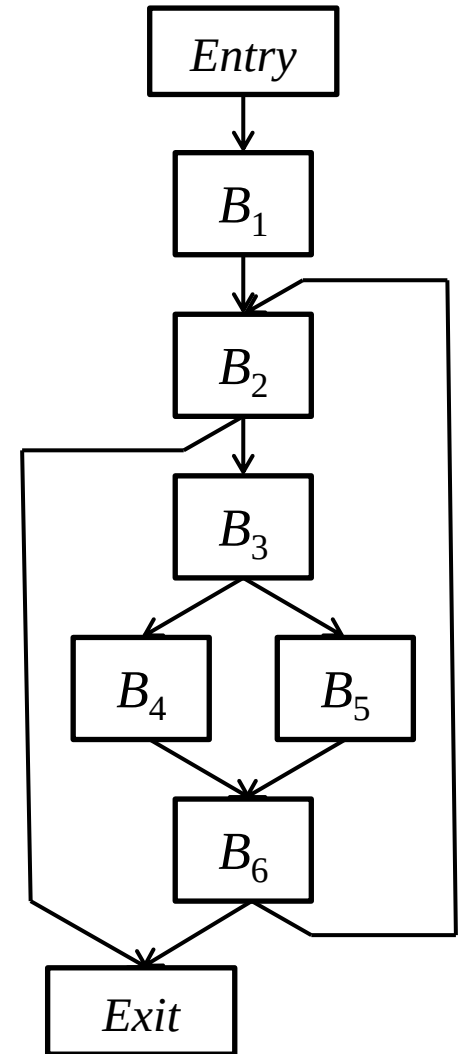
2. Отметить как инвариантные все операнды-константы
3. Отметить как инвариантные все операнды, у которых все определения, достигающие инструкции, находятся вне цикла
4. Отметить как инвариантные все инструкции, все операнды которых отмечены
5. Повторять шаги 2 – 4, пока инвариантные инструкции не перестанут выделяться
6. Переместить все выделенные инструкции в предзаголовок.

5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $\text{if}(i > 100)$	B_3 $a \leftarrow b + 1$ $c \leftarrow 2$ $\text{if}(i \% 2 = 0)$
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if}(a < 2)$

Исходный цикл

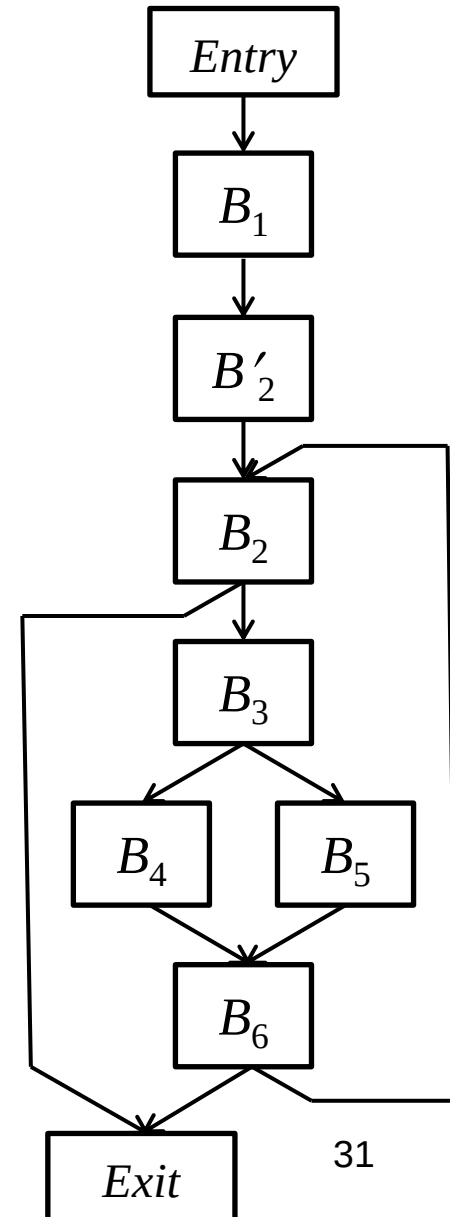


5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $\text{if}(i > 100)$	B'_2
	B_3 $a \leftarrow b + 1$ $c \leftarrow 2$ $\text{if}(i \% 2 = 0)$	
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if}(a < 2)$

После добавления предзаголовка (B'_2)



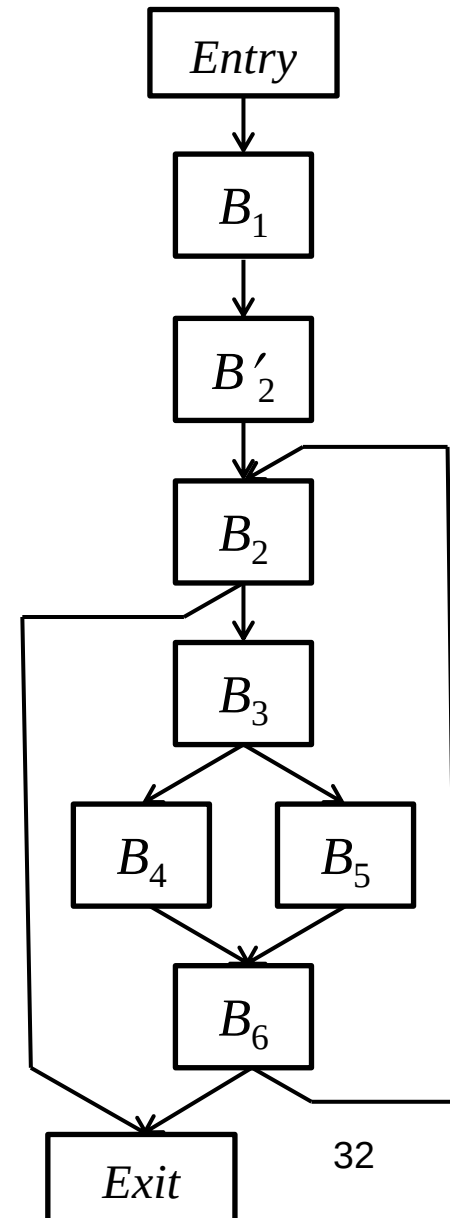
5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $\text{if } (i > 100)$	B'_2
	B_3 <u>$a \leftarrow b + 1$</u> <u>$c \leftarrow 2$</u> $\text{if } (i \% 2 = 0)$	
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if } (\underline{a} < \underline{2})$

Выделение инвариантных инструкций:

- ◇ операнды-константы отмечены зеленым цветом
- ◇ операнды, определенные вне цикла – коричневым цветом
- ◇ инвариантные инструкции подчеркнуты

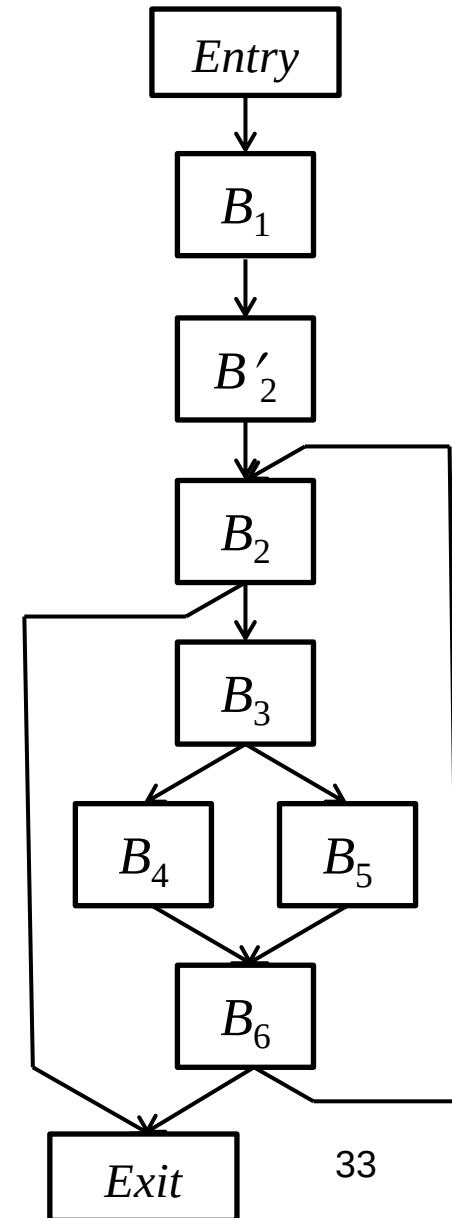


5.2. Оптимизация циклов

5.2.6 Перемещение кода, инвариантного относительно цикла

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $\text{if } (i > 100)$	B'_2 $a \leftarrow b + 1$ $c \leftarrow 2$ $t1 \leftarrow a < 2$
	B_3 $\text{if } (i \% 2 = 0)$	
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if } (t1)$

После вынесения инвариантного кода в
предзаголовок



5.3. Исключение бесполезного кода

5.3.1 Постановка задачи

- ◇ Программа может содержать *бесполезный код* – инструкции, не влияющие на результат вычислений. Как правило, бесполезный код появляется в программе в результате работы некоторых алгоритмов анализа и оптимизации, реализованных в компиляторе.
- ◇ Существует несколько разновидностей бесполезного кода:
 - ◇ *Мертвый код* – инструкции, результат которых не используется в дальнейших вычислениях.
 - ◇ *Недостижимый код* – инструкции, которые не содержатся ни в одном реальном пути выполнения.
 - ◇ *Избыточный код* – инструкции, повторно вычисляющие уже вычисленные значения (например, доступные выражения или инвариантные вычисления в циклах).
- ◇ Требуется обнаружить и удалить бесполезный код

5.3. Исключение бесполезного кода

5.3.2 Алгоритм *Mark & Sweep*.

- ◇ Алгоритм *Mark & Sweep* (двухпроходный), применяющийся для освобождения динамической памяти в сборщиках мусора, может использоваться и для исключения бесполезного кода.
- ◇ Инструкция называется *полезной*, если она:
 - ◇ вычисляет возвращаемое значение процедуры
 - ◇ является обращением к функции ввода-вывода
 - ◇ вычисляет значение глобальной переменной, доступной из других процедур
 - ◇ ее результат используется в других полезных инструкциях.
- ◇ Алгоритм состоит из двух проходов:
 - ◇ на первом проходе (*Mark*) выявляются и помечаются все полезные инструкции.
 - ◇ на втором проходе (*Sweep*) непомеченные инструкции удаляются.

5.3. Исключение бесполезного кода

5.3.3 Поток управления: переходы и ветвления.

- ◇ При удалении бесполезных инструкций необходимо учитывать *поток управления*.
- ◇ Поток управления определяется с помощью инструкций перехода. В промежуточном представлении определено два вида *инструкций перехода*:
 - ◇ *переходы (jump)* – инструкция **goto** (безусловный переход).
 - ◇ *ветвления (branch)* – инструкции условного перехода **ifTrue x goto L** и **ifFalse x goto L**, используемые для отображения в промежуточное представление операторов **if-then**, **if-then-else**, **switch** исходного языка.
- ◇ Для описания потока управления понадобится понятие зависимости по управлению.

5.3. Исключение бесполезного кода

5.3.4 Постдоминаторы (напоминание)

- ◇ *Обратным графом* ориентированного графа $G = \langle N, E \rangle$ называется ориентированный граф $G^R = \langle N, E^R \rangle$, у которого направления всех ребер противоположны.
- ◇ В ГПУ вершина p является *постдоминатором* вершины n ($p = \text{Postdom}(n)$), если каждый путь из вершины n в вершину *exit* проходит через вершину p .
Постдоминаторы ГПУ – это доминаторы его *обратного графа*.
- ◇ *Обратная граница доминирования* ($RDF(n)$) вершины $n \in G$ это обычная граница доминирования в обратном графе G^R .

5.3. Исключение бесполезного кода

5.3.5 Зависимость по управлению.

- ◇ По определению, вершина m ГПУ *зависит по управлению* от вершины n тогда, и только тогда, когда:
 - ◇ существует непустой путь T от n до m , такой что $\forall k \in T - \{n\}: m = \text{Postdom}(k)$, т.е. если выполнение программы пошло по пути T , то, чтобы достичь *exit*, оно обязательно пройдет через m .
 - ◇ m не обязательно является строгим постдоминатором n : у n может быть несколько выходов, так что помимо T возможны и другие пути, проходящие через n , но потом ведущие не в m , а в другие вершины.
- ◇ **Обратная граница доминирования позволяет определять границы зависимостей по управлению.**

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark*.

- ◇ На первом проходе (*Mark*) в каждом базовом блоке n :
 - ◇ выбирается очередная инструкция из *Worklist*
 - ◇ для этой инструкции
 - ◆ удаляется ее пометка, так как *Worklist* содержит только помеченные инструкции
 - ◆ с помощью специальной процедуры выясняется полезность инструкции
 - ◆ если инструкция полезна, ее пометка восстанавливается
 - ◆ помечаются ветви, по которым «приходят» операнды инструкции (операнды полезны, так как используются в полезной инструкции)
 - ◆ посещаются все блоки $b \in RDF(n)$ и помечается каждая ветвь, ведущая к этим блокам.

Каждая помеченная ветвь помещается в *Worklist*.

Проход завершается, когда *Worklist* становится пустым.

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark*.

- ◇ Базовый блок, содержащий хотя бы одну помеченную инструкцию, помечается для ускорения анализа.
 - ◇ Ветвь состоит из одного или более блоков. Она считается помеченной, если помечен хотя бы один из ее блоков.
- Каждая помеченная ветвь помещается в *Worklist*.
- Проход завершается, когда *Worklist* становится пустым.

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark* (псевдокод)

Mark ()

WorkList $\leftarrow \emptyset$;

for each инструкции i ($x \leftarrow op, y, z :: \text{пометка}$)

 убрать пометку у i

 if (i полезная) пометить i

 WorkList $\leftarrow$ WorkList $\cup \{i\}$

while (WorkList $\neq \emptyset$)

 remove i from WorkList

 if (def(y) не помечена)

 пометить def(y)

 WorkList $\leftarrow$ WorkList $\cup \{\text{def}(y)\}$

 if (def(z) не помечена)

 пометить def(z)

 WorkList $\leftarrow$ WorkList $\cup \{\text{def}(z)\}$

for each block $b \in \text{RDF}(\text{block}(i))$

 пусть j ветвь, оканчивающаяся в b

 if (j не помечена)

 пометить j

 WorkList $\leftarrow$ WorkList $\cup \{j\}$

5.3. Исключение бесполезного кода

5.3.6 Проход *Sweep*

Sweep ()

```
for each instruction i
    if (i is unmarked)
        if (i is a branch)
            rewrite i with a jump
            to i's nearest marked
            postdominator
        if (i is not a jump)
            delete i
```

- ◇ На втором проходе (*Sweep*) в каждый блок, с которого начинается непомеченная ветвь, помещается безусловный переход на его помеченный постдоминатор.
- Это правильно, так как если ветвь не помечена, потомки блока вплоть до его непосредственного постдоминатора, не могут содержать полезных инструкций, так как иначе они были бы помечены.

5.3. Исключение бесполезного кода

5.3.6 Проход *Sweep*

Sweep ()

```
for each instruction i
    if (i is unmarked)
        if (i is a branch)
            rewrite i with a jump
            to i's nearest marked
            postdominator
        if (i is not a jump)
            delete i
```

◇ Сказанное справедливо и для непосредственного непомеченного постдоминатора.

Чтобы найти ближайший помеченный постдоминатор, можно двигаться вверх по дереву постдоминаторов, пока не найдется помеченный блок. Поиск обязательно закончится, так как по определению блок *exit* помечен.

5.4. Исключение недостижимого кода

5.4.1 Постановка задачи

- ◇ Иногда ГПУ содержит *недостижимый код*. Компилятор должен найти недостижимые базовые блоки и исключить их.
- ◇ Две причины недостижимости блока:
 - ◇ в ГПУ отсутствует путь, ведущий к базовому блоку;
 - ◇ путь, достигающий блока, может быть невыполнимым (например, `if (i * (i + 1) % 2 != 0) { ... }`)
- ◇ Здесь рассматривается только первый случай, в котором для анализа можно использовать алгоритм типа *Mark & Sweep*.
- ◇ Этот анализ прост и недорог. Он может быть выполнен попутно во время обхода ГПУ для других целей или даже во время построения ГПУ.

5.4. Исключение недостижимого кода

5.4.2 Анализ достижимости

- ◇ Проход *Mark* сначала помечает каждый блок b как «недостижимый», потом он начинает обход ГПУ с *Entry* и помечает как «достижимый» каждый блок, которого он может достичь.
- ◇ Если все ветвления и переходы определяются однозначно, то все блоки, помеченные как недостижимые, действительно недостижимы и могут быть удалены на проходе *Sweep*.
- ◇ В случае неоднозначных условий ветвления, компилятор должен сохранить любой блок, достижимый ветвлением или переходом.

5.5. Оптимизация потока управления

5.5.1. Постановка задачи

- ◇ Некоторые оптимизации могут иметь побочный эффект, изменяющий форму ГПУ, добавляя в него бесполезные блоки и дуги.
Если компилятор содержит такие оптимизации, он должен также содержать проход, упрощающий ГПУ, исключая бесполезный поток управления.
- ◇ Функция *Clean* обрабатывает непосредственно ГПУ оптимизируемой процедуры, упрощая его.

5.5. Оптимизация потока управления

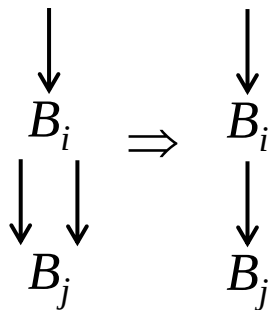
5.5.2. Основные преобразования. Преобразование 1.

◇ Функция *Clean* применяет следующие четыре основных преобразования (в указанном порядке):

- ◇ 1. *Свернуть избыточную ветвь*: Если последние инструкции блока B_i реализуют ветвление, и обе ветви выполняют условный переход на один и тот же блок B_j , то ветвление заменяется безусловным переходом на блок B_j .

Такая ситуация может возникнуть в результате других оптимизаций

(Например, у B_i могло быть два последователя, каждый из которых заканчивался переходом на B_j . Если другие оптимизации убрали из этих блоков все инструкции, то второе из основных преобразований могло породить левый граф, показанный на рисунке).



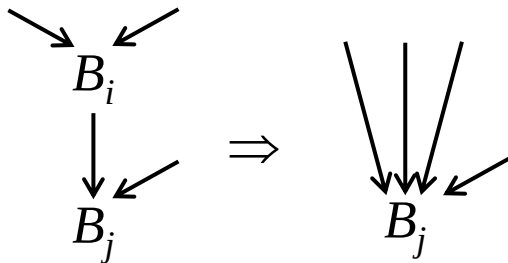
5.5. Оптимизация потока управления

5.5.2. Основные преобразования. Преобразование 2.

- ◇ 2. *Удалить пустой блок:* Если блок B_i содержит только инструкцию перехода, то он поглощается своим последователем – блоком B_j .

Такая ситуация возникает, когда предшествующие оптимизации удаляют все инструкции из блока B_i .

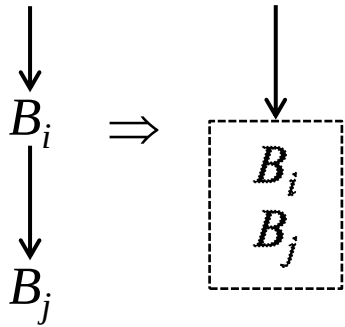
Так как у B_i всего один последователь, B_j , преобразование перенаправляет дуги, входящие в B_i , к B_j и исключает B_i из $Pred(B_j)$, что упрощает ГПУ и ускоряет выполнение.



5.5. Оптимизация потока управления

5.5.2. Основные преобразования. Преобразование 3.

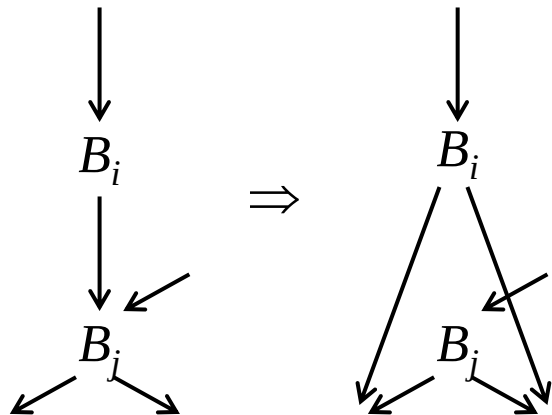
- ◇ 3. *Объединение блоков:* Если имеется блок B_i , который оканчивается переходом на B_j , у которого всего один предшественник, B_i , он может объединить эти блоки как показано на рисунке внизу, что позволяет исключить переход из B_i в B_j .



5.5. Оптимизация потока управления

5.5.2. Основные преобразования. Преобразование 4.

- ◇ 4. *Подъём ветвлений*. В ситуации, когда блок B_i , который оканчивается переходом в пустой блок B_j , а блок B_j оканчивается ветвлением, переход в конце блока заменяется на копию ветвления из блока B_j . Такое преобразование поднимает ветвление из B_j в B_i . Ситуация может возникнуть, если другие оптимизации удалят все операции из B_j , оставив только ветвление. К ГПУ добавится ребро. Объединить B_i и B_j нельзя, так как у B_j есть еще предшественники (если бы их не было, B_i и B_j уже были бы объединены Преобразованием 3).



5.5. Оптимизация потока управления

5.5.3. Функция Clean ()

Clean()

while ГПУ продолжает изменяться

compute

Postorder

OnePass()

OnePass()

for each block B_i || in postorder

if (B_i оканчивается ветвлением)

if (обе цели одинаковы)

заменить ветвление на переход

/* 1 */

if (B_i оканчивается переходом на B_j)

if (B_i пуст)

заменить все переходы на B_i переходами на B_j

/* 2 */

if (B_j имеет только одного предшественника)

совместить B_i и B_j

/* 3 */

if (B_j пуст и оканчивается ветвлением)

заменить B_i переход

на копию ветвления из B_j

/* 4 */

5.5. Оптимизация потока управления

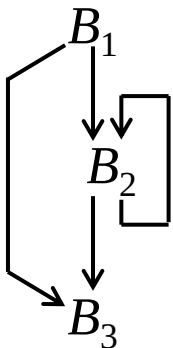
5.5.3. Функция **Clean ()**

- ◇ Функция **Clean ()** многократно вызывает функции **Postorder ()** (обычная нумерация блоков) и **OnePass ()** (однократный проход), выполняя последовательность преобразований 1 – 4 итеративно до тех пор, пока ГПУ оптимизируемой процедуры продолжает изменяться.
- ◇ В начале каждой итерации выполняется новая нумерация блоков, так как после каждого применения четверки преобразований ГПУ может сильно измениться.
- ◇ Функция **Clean ()** не может самостоятельно удалить пустой цикл (цикл с пустым телом). Это показывает следующий пример.

5.5. Оптимизация потока управления

5.5.4. Пример

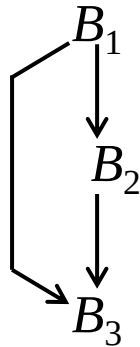
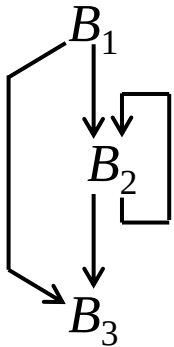
- ◇ Рассмотрим процедуру, ГПУ которой изображен на рисунке. Пусть блок B_2 пуст. Ни одно из преобразований функции $Clean()$ не может удалить блок B_2 :
 - ◇ ветвление в конце B_2 не избыточно;
 - ◇ B_2 не завершается переходом, и $Clean()$ не может объединить его с B_3 ;
 - ◇ предшественник B_2 блок B_1 оканчивается ветвлением, а не переходом, и $Clean()$ не может ни объединить его с B_1 , ни свернуть его ветвление в B_1 .



5.5. Оптимизация потока управления

5.5.4. Пример

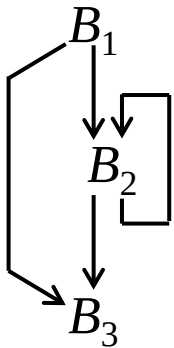
- ◇ Однако если исходный ГПУ предварительно обработать с помощью *Mark & Sweep*, рассматриваемый пустой цикл удастся удалить.
- ◇ Блоки B_1 и B_3 содержат полезные инструкции, а блок B_2 нет, проход *Mark* решит, что ветвление в конце B_2 бесполезно, так как $B_2 \notin RDF(B_3)$.
Если ветвление бесполезно, бесполезен и код, вычисляющий условие ветвления. Поэтому *Sweep* удалит из B_2 все инструкции и преобразует ветвление в конце B_2 в переход на ближайший полезный постдоминатор B_3 . Получится ГПУ на правом рисунке.



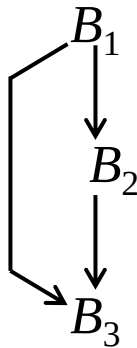
5.5. Оптимизация потока управления

5.5.4. Пример

- ◇ *Clean* загоняет B_2 в B_1 , в результате получается ГПУ, изображенный на третьем рисунке слева.
- ◇ Теперь ветвление в конце B_1 становится избыточным, и *Clean* заменяет его переходом, в результате получается ГПУ, изображенный на четвертом рисунке слева.
- ◇ Наконец, если окажется, что B_1 – единственный предшественник B_3 , *Clean* объединит эти два блока в один блок.



Исходный ГПУ



ГПУ после
Mark & Sweep



ГПУ после
удаления B_2 .



ГПУ после
сворачивания
избыточной
ветви.

6. Оптимизация циклов (часть I)

6.1. Выделение естественных циклов

6.1.1 Классификация дуг ГПУ

- ◇ Дуги ГПУ, являющиеся дугами и его остовного дерева, называются *остовными*.
- ◇ Дуги ГПУ, не являющиеся дугами его остовного дерева, но имеющие такое же направление, что и остовные, называются *прямыми*.
- ◇ Дуги ГПУ, направленные противоположно остовным, называются *обратно направленными*.
- ◇ Обратно направленная дуга ГПУ $\langle B_i, B_k \rangle$ называется *обратной*, если $B_k = Dom(B_i)$
- ◇ Остальные дуги ГПУ называются *поперечными*.
Поперечные дуги соединяют различные поддеревья остовного дерева и в программах нормальных программистов не встречаются, так как нормальные программисты не злоупотребляют применением **goto**.

6.1. Выделение естественных циклов

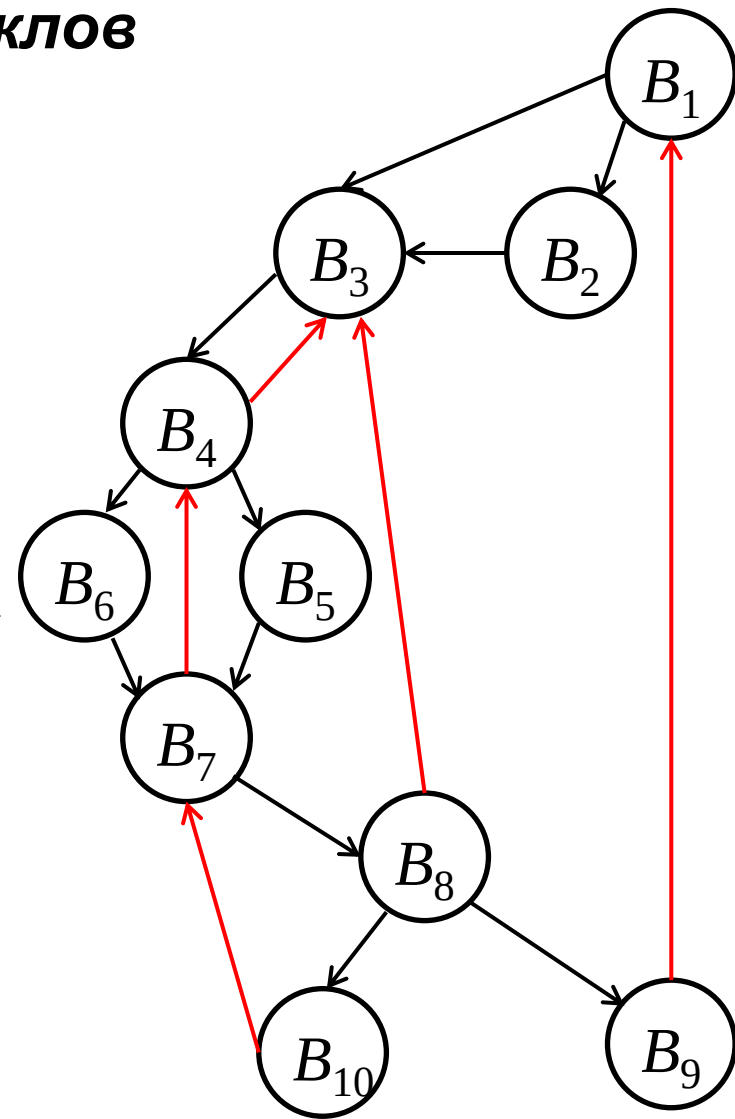
6.1.2 Определение естественного цикла

- ◇ **Определение.** *Естественным циклом* называется цикл со следующими свойствами:
 - ◇ Цикл имеет единственную входную вершину, называемую его *заголовком*,
 - ◇ Существует обратная дуга, ведущая в заголовок цикла
- ◇ **Определение.** *Естественный цикл обратной дуги* $\langle B_i, B_k \rangle$ составляют узел B_k (*заголовок цикла*) и все узлы ГПУ, из которых можно достичь узла B_i , не проходя через узел B_k . (эти узлы составляют *тело цикла*).

6.1. Выделение естественных циклов

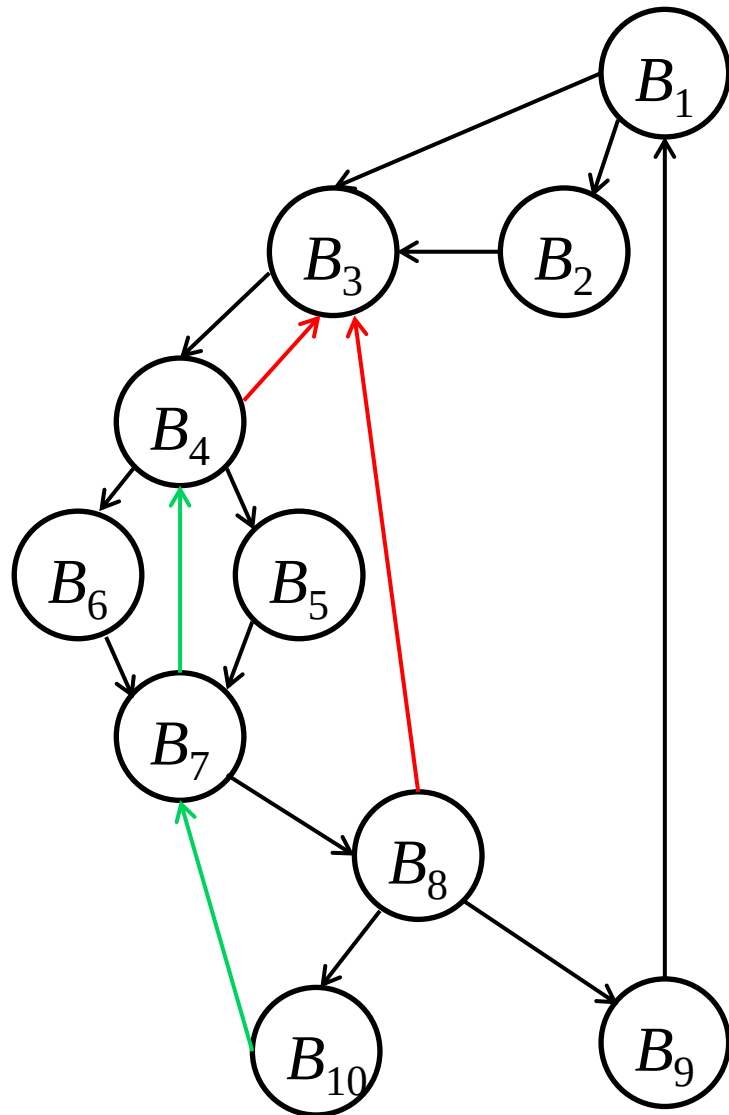
6.1.3 Примеры естественных циклов

- ◇ На рисунке справа – пять обратных дуг:
 $\langle B_{10}, B_7 \rangle$, $\langle B_7, B_4 \rangle$, $\langle B_4, B_3 \rangle$,
 $\langle B_8, B_3 \rangle$, $\langle B_9, B_1 \rangle$
- ◇ Обратной дуге $\langle B_{10}, B_7 \rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и B_{10} можно достичь вершины B_{10} , не проходя через B_7 .
- ◇ Обратной дуге $\langle B_7, B_4 \rangle$ соответствует естественный цикл
 $\{B_4, B_5, B_6, B_7, B_8, B_{10}\}$
Этот цикл включает в себя цикл дуги
 $\langle B_{10}, B_7 \rangle$, который является вложенным циклом



6.1. Выделение естественных циклов

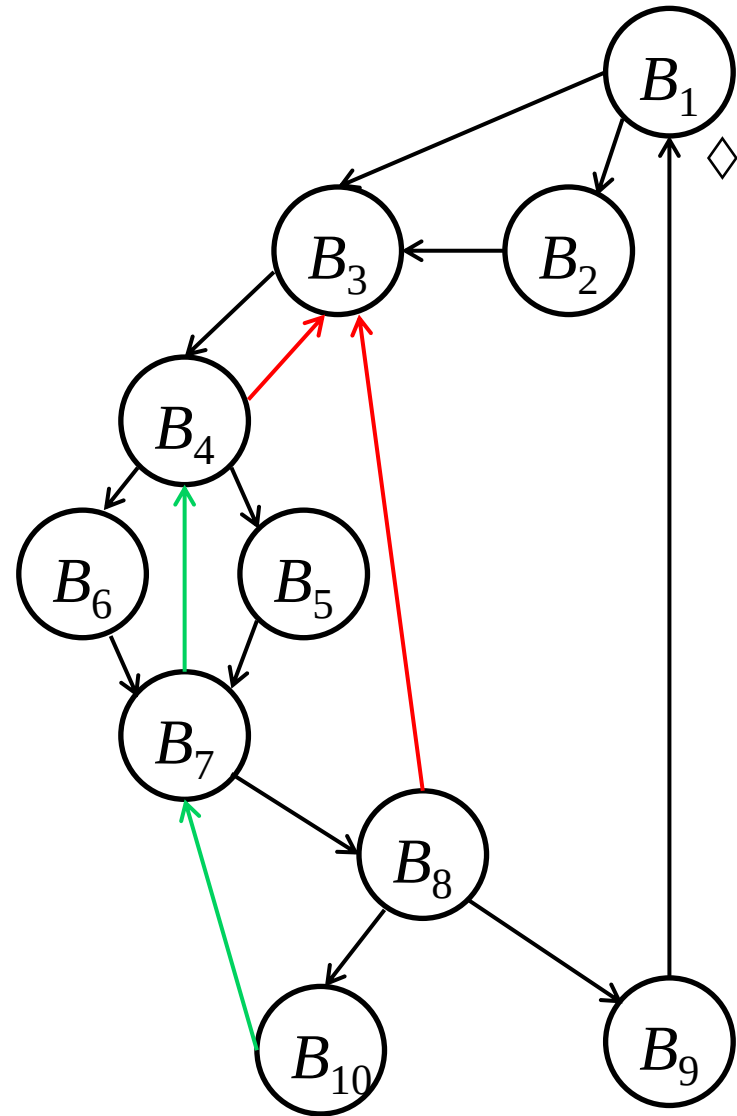
6.1.3 Примеры естественных циклов



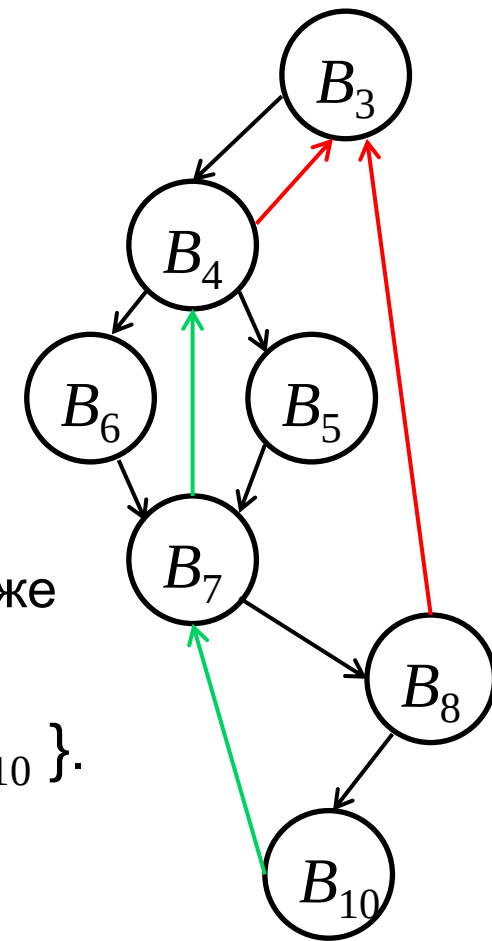
◇ У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один. В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$

6.1. Выделение естественных циклов

6.1.3 Примеры естественных циклов

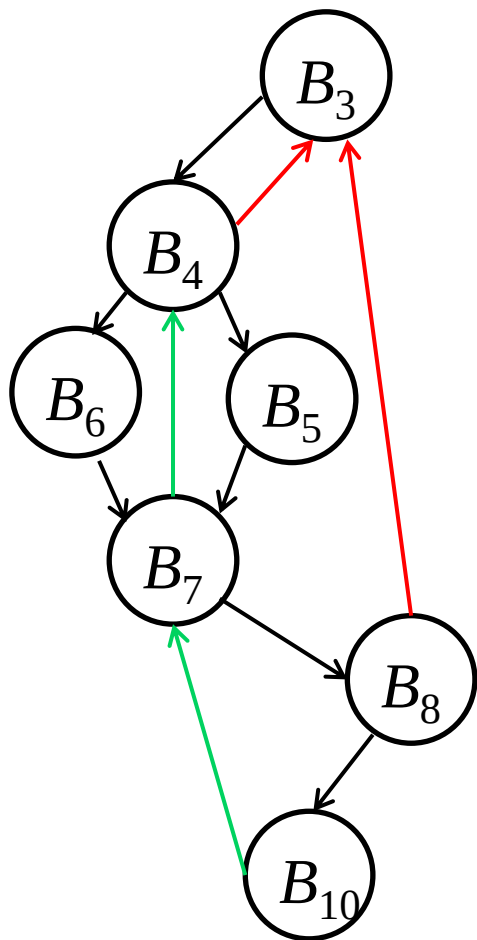


У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один. В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$

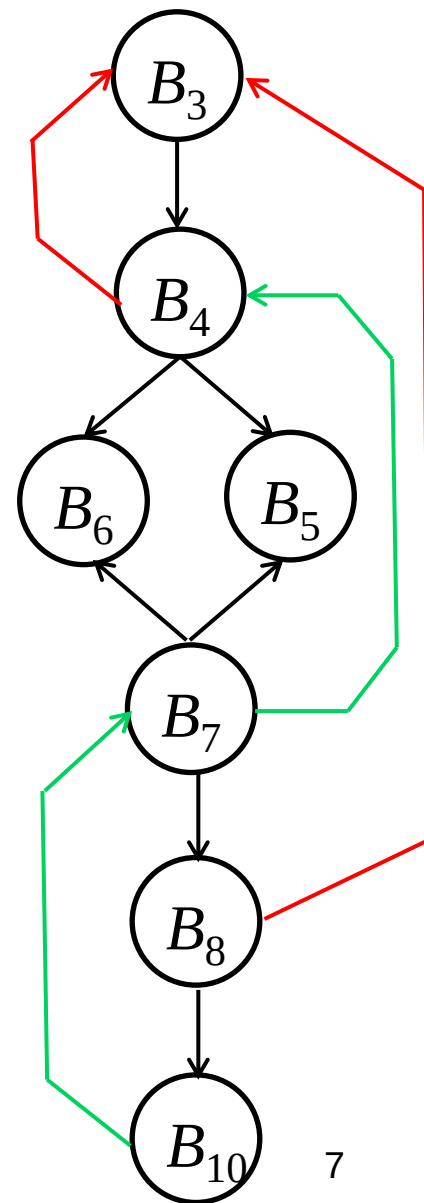


6.1. Выделение естественных циклов

6.1.3 Примеры естественных циклов



- ◇ У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один. В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$
- ◇ При этом изображение ГПУ справа понятнее, чем изображение слева.



6.1. Выделение естественных циклов

6.1.4 Алгоритм построения естественного цикла по обратной дуге

Вход: ГПУ $G = \langle N, E \rangle$ с входным узлом $Entry$.

Обратное ребро $e = \langle n, d \rangle \in E$

Выход: подграф $C \subseteq G$, являющийся естественным циклом.

Метод:

- (1) начальное значение C – множество $\{n, d\}$.
- (2) узел d помечается как «посещенный».
- (3) начиная с узла n выполняется поиск в глубину на обратном ГПУ (направления дуг заменены на противоположные).
- (4) все узлы, посещенные на шаге (3), добавляются в C .

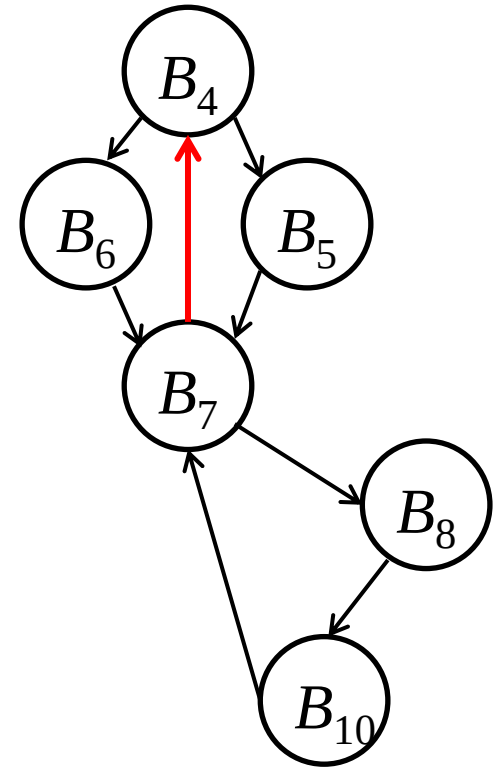
6.1. Выделение естественных циклов

6.1.5 Пример построения естественного цикла по обратной дуге.

◇ Применим алгоритм 6.1.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$.

Отметим вершину B_4 как посещенную и выполним поиск в глубину, начиная с вершины B_7 .

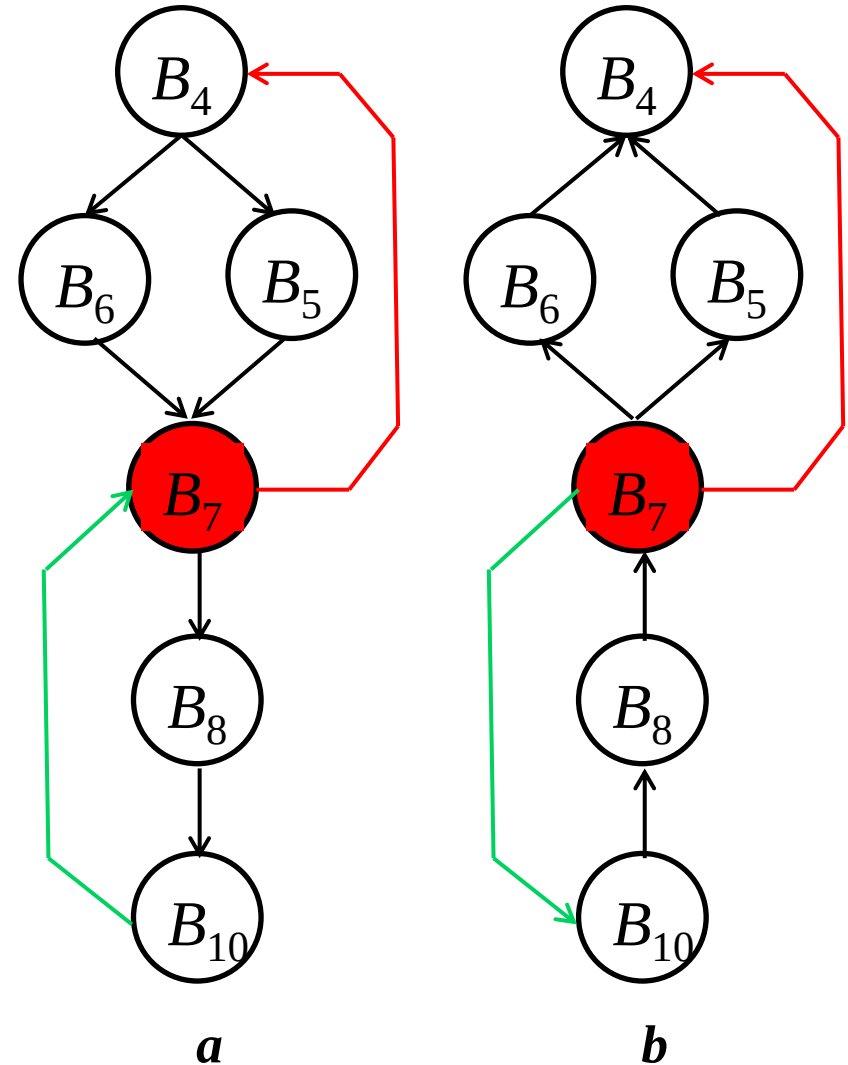
При этом будем считать, что на ГПУ стрелки соответствуют не концу, а началу дуги, т.е. роль множества $Succ(B_7)$ выполняет множество $Pred(B_7) = \{B_5, B_6, B_{10}\}$



6.1. Выделение естественных циклов

6.1.5 Пример построения естественного цикла по обратной дуге.

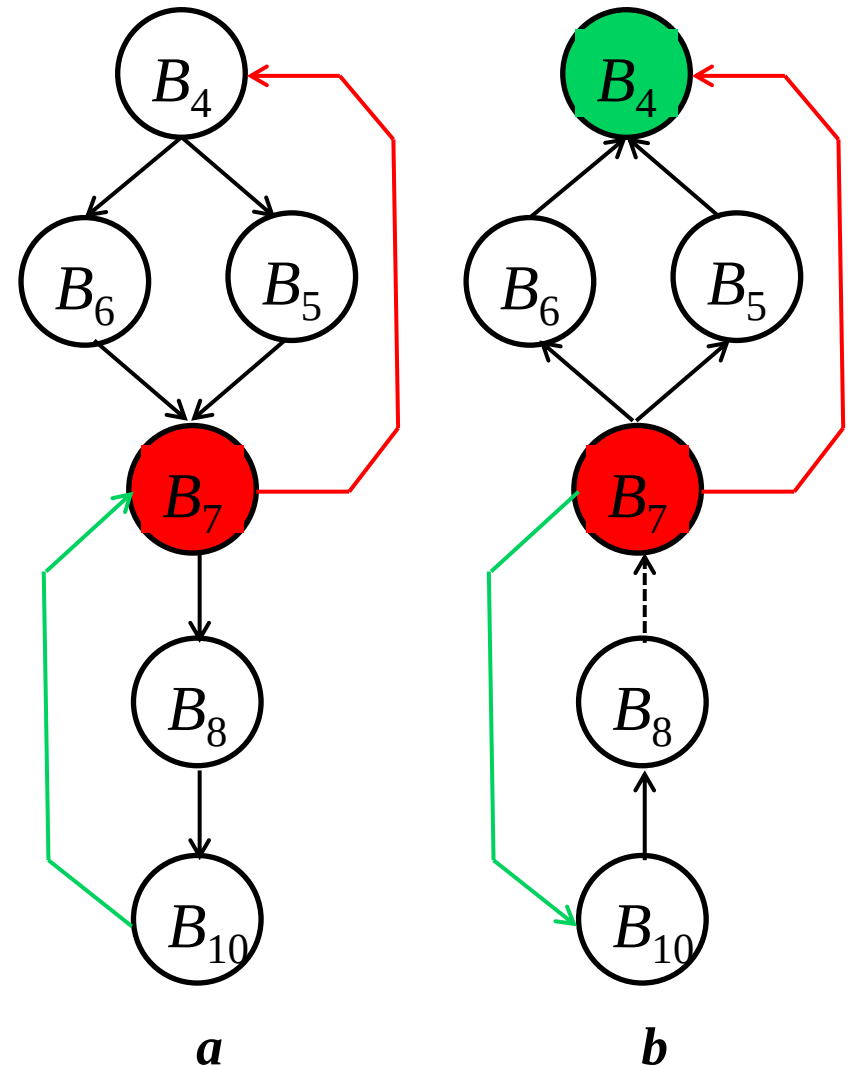
- ◇ Применим алгоритм 5.2.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$ (рисунок *a*)
- ◇ Изменим направление дуг на противоположное (рисунок *b*) и выполним поиск в глубину, начиная с вершины B_7 , отметив вершину B_4 как посещенную.



6.1. Выделение естественных циклов

6.1.5 Пример построения естественного цикла по обратной дуге.

- ◇ Применим алгоритм 5.2.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$
- ◇ На рисунке **b**
 $Succ(B_7) = \{B_5, B_6, B_{10}\}$.
 $Succ(B_{10}) = \{B_8\}$
Следовательно, в состав цикла помимо концов обратной дуги (блоков B_7 и B_4), войдут блоки B_5, B_6, B_{10}, B_8



6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла

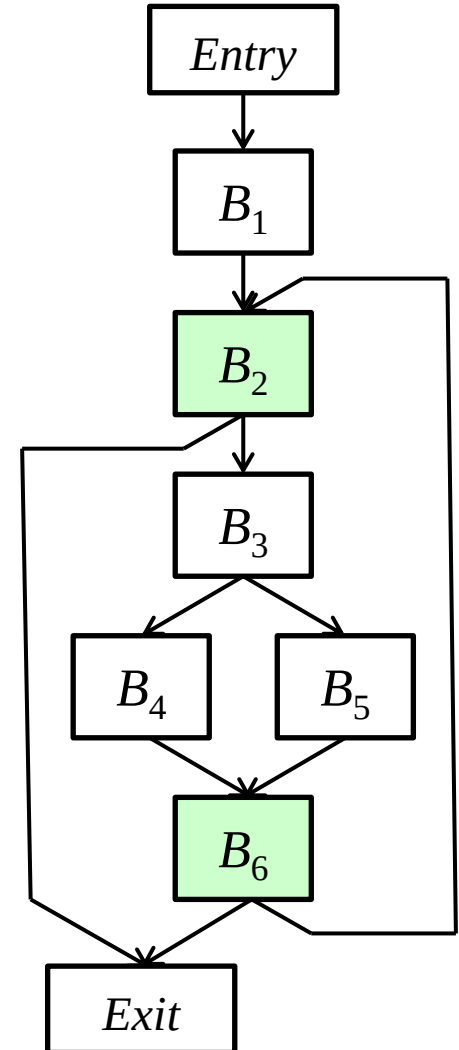
- ◇ Оптимизация состоит в том, что в ГПУ добавляется еще один базовый блок – *предзаголовок* цикла B'_2 , в который выносятся из цикла все инструкции, инвариантные относительно цикла.
- ◇ *Инвариантом цикла* является:
 - ◇ константа **c**;
 - ◇ переменная **u**, все определения которой находятся вне цикла;
 - ◇ переменная **v**, определяемая внутри цикла, если
 - ◆ у **v** есть всего одно определение
 - ◆ это определение находится в базовом блоке, являющемся доминатором всех выходов из цикла
- ◇ Инструкция *инвариантна относительно цикла*, если все ее операнды являются инвариантами цикла.

6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $c \leftarrow 2$ $d \leftarrow b + i$ $\text{if } (i > 100)$	B_3 $a \leftarrow b + 1$ $\text{if } (i \% 2 = 0)$
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if } (i < 100)$

- ◇ Исходный цикл
- ◇ Блок B_1 выполняется до цикла, все остальные – в цикле.
- ◇ Выходы из цикла – блоки B_2 и B_6
- ◇ Блоки B_3 , B_4 , B_5 и B_6 не являются доминаторами блока B_2 . Следовательно, из указанных блоков нельзя выносить код, так как иногда это может не оптимизировать, а pessимизировать программу

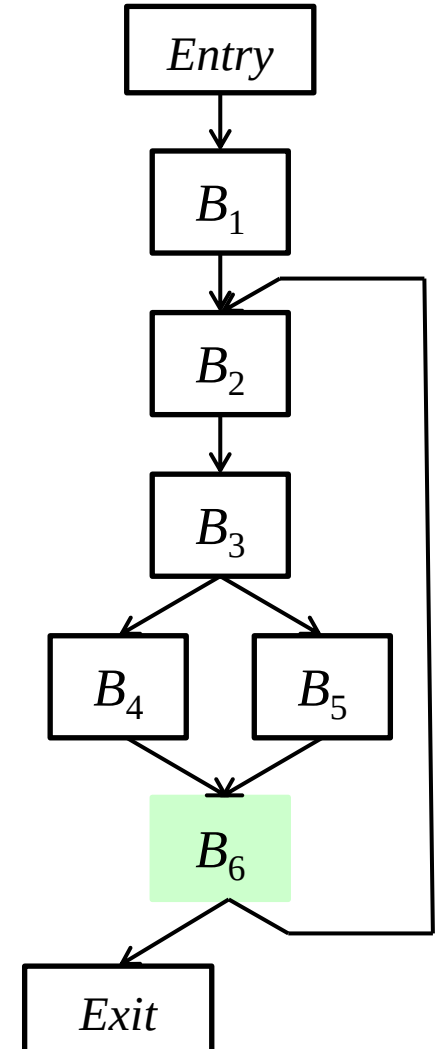


6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $d \leftarrow b + i$ $c \leftarrow 2$	B_3 $a \leftarrow b + 1$ $\text{if } (i \% 2 = 0)$
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if } (i < 100)$

◇ Изменим исходный цикл, убрав из блока ненужное сравнение. У цикла останется только один выход – блоки B_6



6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Код, инвариантный относительно цикла. Пример 1

◇ Рассмотрим цикл с заголовком B_2 .

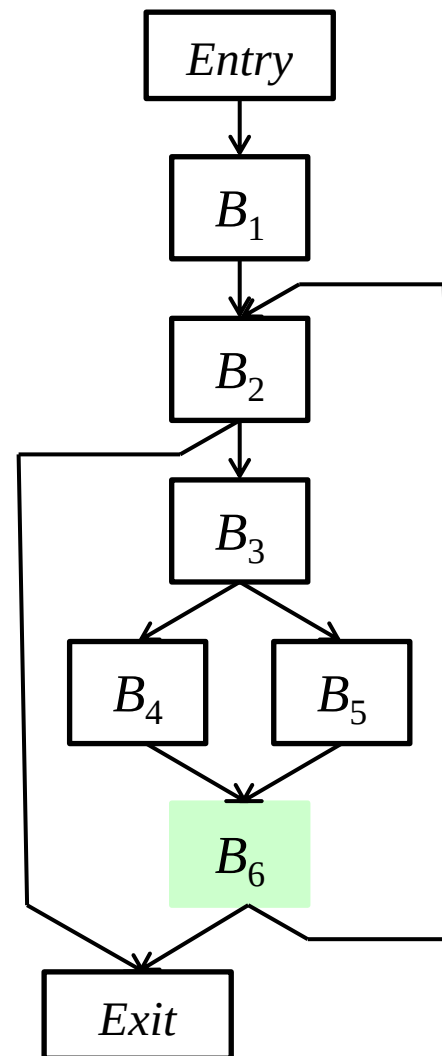
B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $d \leftarrow b+i$ $c \leftarrow 2$	B_3 $a \leftarrow b+1$ $\text{if } (i \% 2 = 0)$
B_4 $d \leftarrow a+d$ $e \leftarrow d+1$	B_5 $d \leftarrow c$ $f \leftarrow d+1$	B_6 $i \leftarrow i+1$ $\text{if } (i < 100)$

◇ Инструкции $a \leftarrow b+1$, $c \leftarrow 2$, $a < 2$ инвариантны относительно цикла:

$a \leftarrow b+1$: b – инвариант цикла, так как его определение находится вне цикла,

1 – инвариант цикла, как константа

$c \leftarrow 2$: 2 – инвариант цикла, как константа

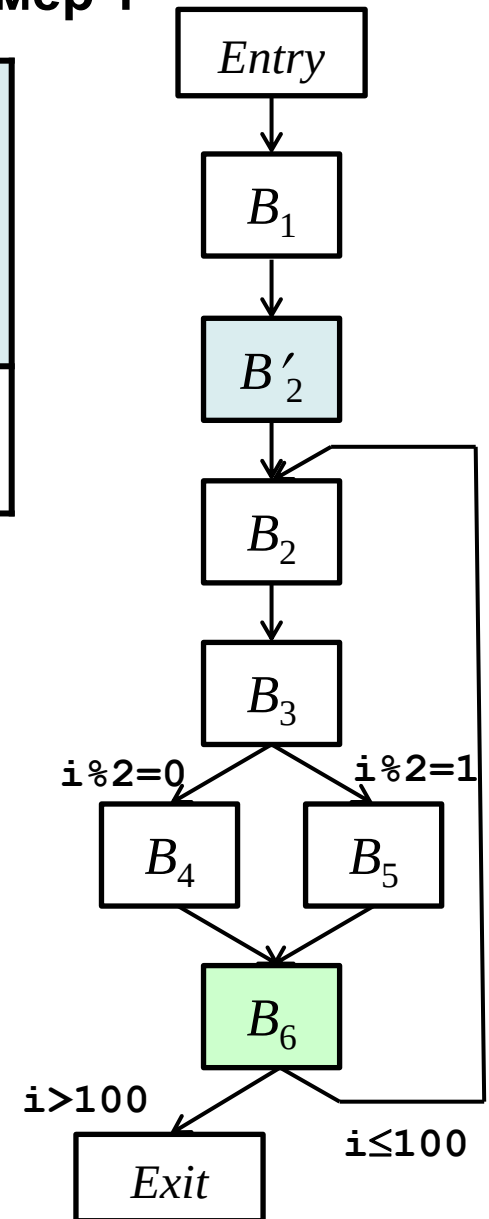


6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Код, инвариантный относительно цикла. Пример 1

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2 $d \leftarrow b + i$ $c \leftarrow 2$	B'_2
	B_3 $a \leftarrow b + 1$ $\text{if } (i \% 2 = 0)$	
B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$	B_5 $d \leftarrow c$ $f \leftarrow d + 1$	B_6 $i \leftarrow i + 1$ $\text{if } (i > 100)$

◇ После добавления предзаголовка (B'_2)

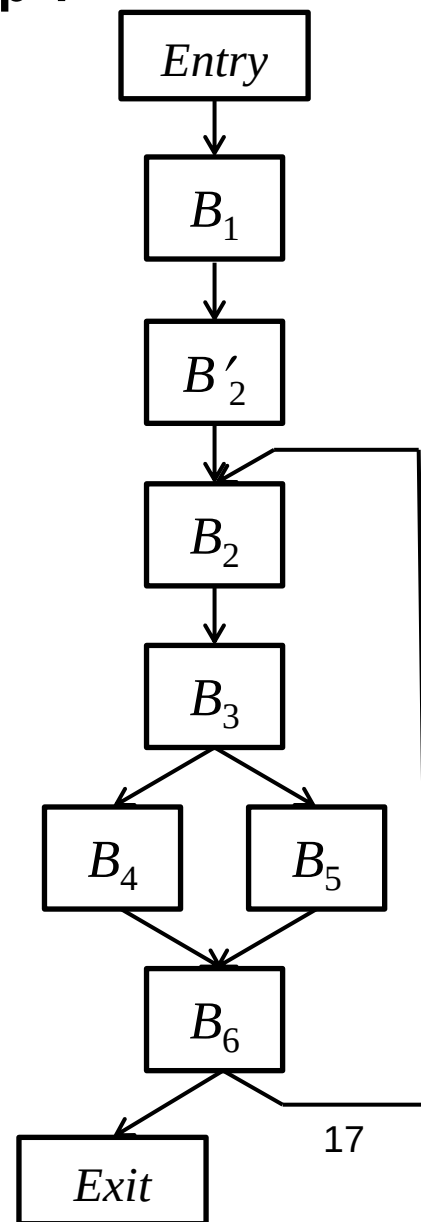


6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Код, инвариантный относительно цикла. Пример 1

B_1 $b \leftarrow 2$ $i \leftarrow 1$	B_2	B'_2 $a \leftarrow b+1$ $c \leftarrow 2$
	B_3 $\text{if } (i \% 2 = 0)$	
B_4 $d \leftarrow a+d$ $e \leftarrow d+1$	B_5 $d \leftarrow c$ $f \leftarrow d+1$	B_6 $i \leftarrow i+1$ $\text{if } (i > 100)$

◇ После вынесения инвариантного кода в предзаголовок

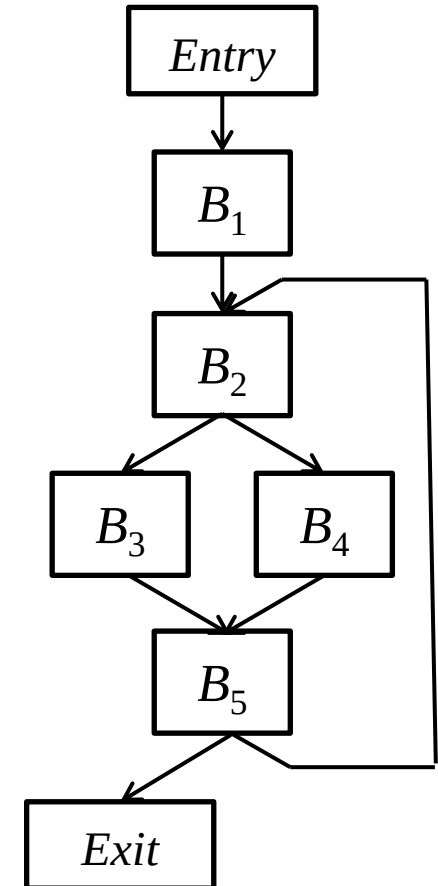


6.2. Перемещение кода, инвариантного относительно цикла

6.2.3 Код, инвариантный относительно цикла. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$	B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$		
B_4 $e \leftarrow 3$		

- ◇ Исходный цикл
- ◇ Блок B_1 выполняется до цикла, все остальные – в цикле

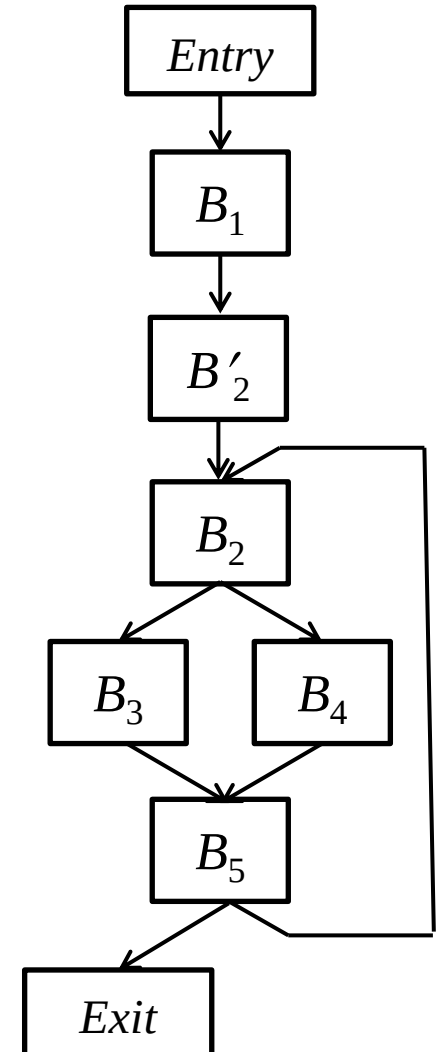


6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$	B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2	
B_4 $e \leftarrow 3$		

◇ После добавления предзаголовка (B'_2)

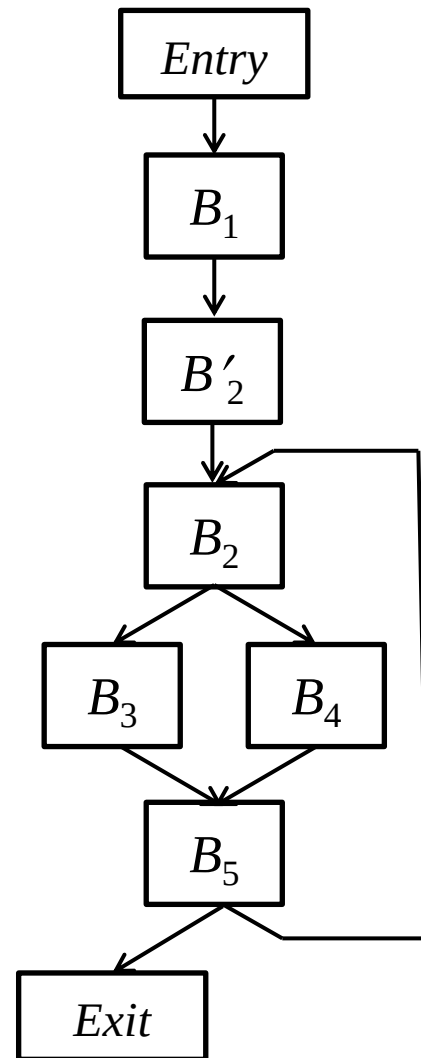


6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$	B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2	
B_4 $e \leftarrow 3$		

- ◇ Инструкции $a \leftarrow b + c$, $d \leftarrow a + 1$ инвариантны относительно цикла:
 - $a \leftarrow b + c$: определения b и c находятся вне цикла,
 - $d \leftarrow a + 1$: 1 константа
 a определено только один раз в блоке, который является доминатором выхода из цикла
- ◇ Инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла; почему?

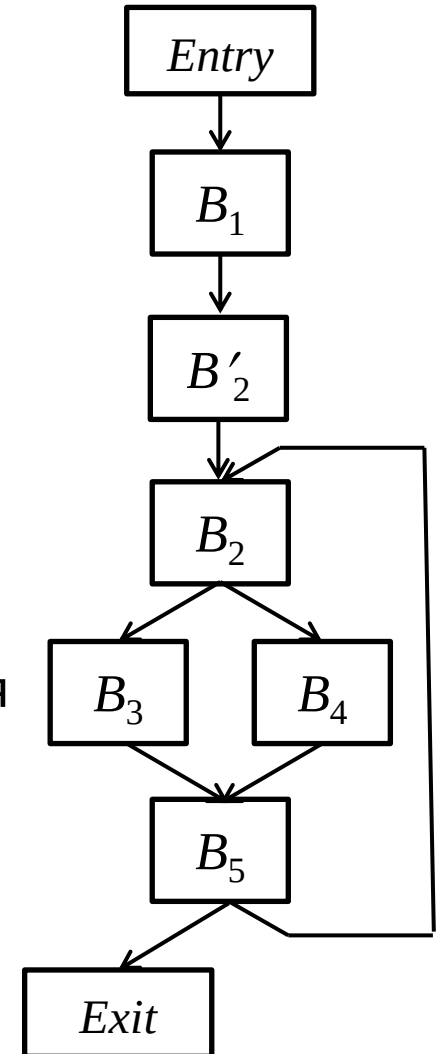


6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$	B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$	
B_4 $e \leftarrow 3$		

- ◇ Инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла; почему?
- ◇ Во-первых, e определяется внутри цикла не один, а два раза; следовательно, значение e изменяется внутри цикла; при вынесении инструкций в предзаголовок e будет определяться только один раз и в зависимости от порядка инструкций в предзаголовке будет всегда иметь только одно значение (2 или 3)

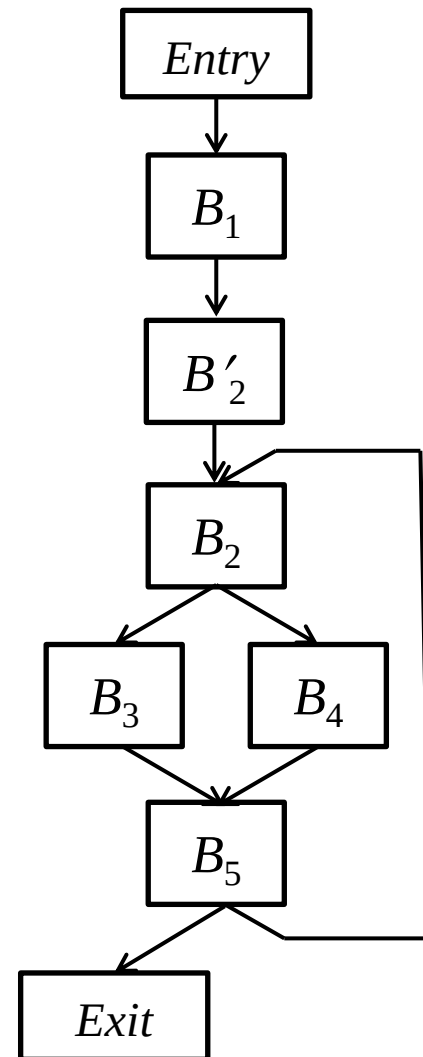


6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$	B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$	
B_4 $e \leftarrow 3$		

- ◇ Инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла; почему?
- ◇ Во-вторых, предзаголовок цикла является доминатором каждого базового блока, входящего в цикл, включая его заголовок (по построению); следовательно, предзаголовок является доминатором всех выходов из цикла (у рассматриваемого цикла всего один выход); значит в предзаголовок можно только инструкции из базовых блоков являющихся доминаторами всех выходов из цикла.

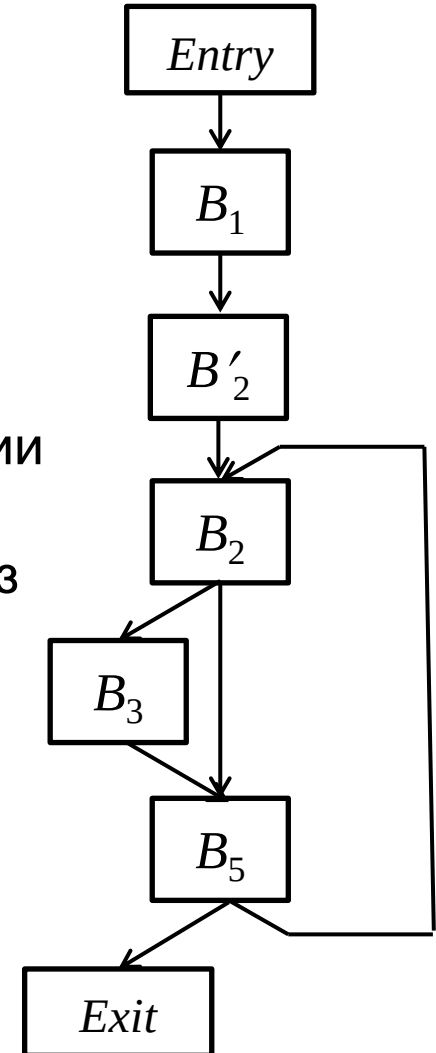


6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$	B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$	

◇ В предзаголовок можно выносить только инструкции из базовых блоков являющихся доминаторами всех выходов из цикла. Если, например, удалить из цикла блок B_4 , то e будет присваиваться только одно значение. Но даже в этом случае нельзя выносить инструкцию $e \leftarrow 2$ в предзаголовок, так как блок B_3 все равно не будет доминатором выхода и результат выполнения программы может измениться.

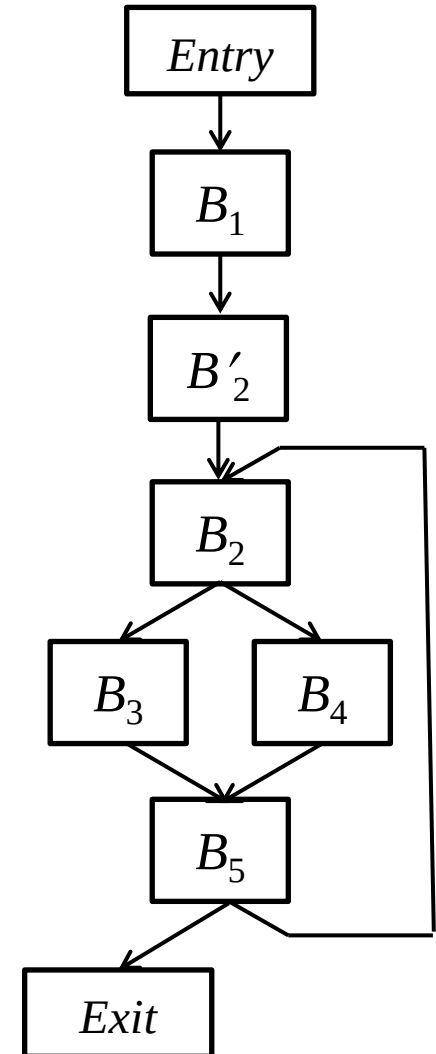


6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $\text{if}(i \% 2 = 0)$	B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if}(i \leq 100)$
B_3 $e \leftarrow 2$	B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$	
B_4 $e \leftarrow 3$		

- ◇ Таким образом, выносить в предзаголовок можно только те инструкции, которые выполняются в блоках, доминирующих над всеми выходами из цикла (в рассматриваемом примере у цикла всего один выход) и которые выполняют единственное присваивание соответствующей переменной.



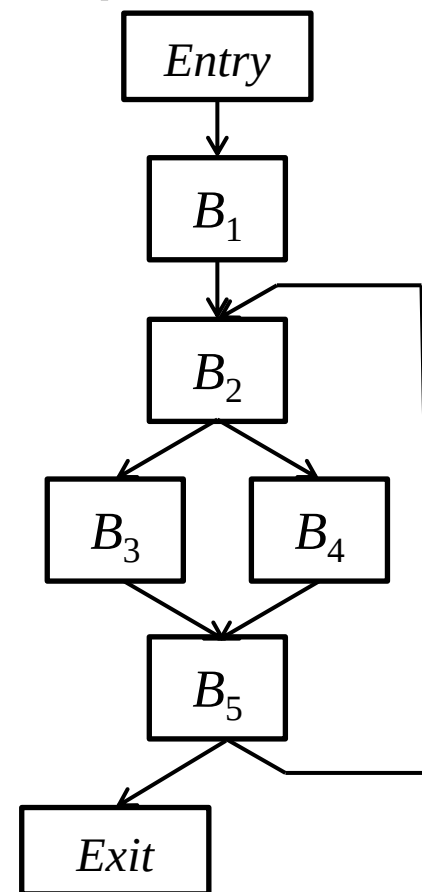
6.2. Перемещение кода, инвариантного относительно цикла

6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 b←2 c←3 i←1	B_2 if (i%2=0)	B_5 f←e+2 i←i+1 if (i≤100)
B_3 e←2	B'_2 a←b+c d←a+1	
B_4 e←3		

Итак:

- ◇ $a \leftarrow b + c$ инвариантный код и его можно вынести в предзаголовок B'_2 (b и c определяются вне цикла, a определяется в блоке B_2 , который доминирует над единственным выходом из цикла – блоком B_5)
- ◇ $d \leftarrow a + 1$ инвариантный код и его можно вынести в предзаголовок B'_2 (a определяется внутри цикла только в блоке B_2 , который доминирует над единственным выходом из цикла – блоком B_5)



6.2. Перемещение кода, инвариантного относительно цикла

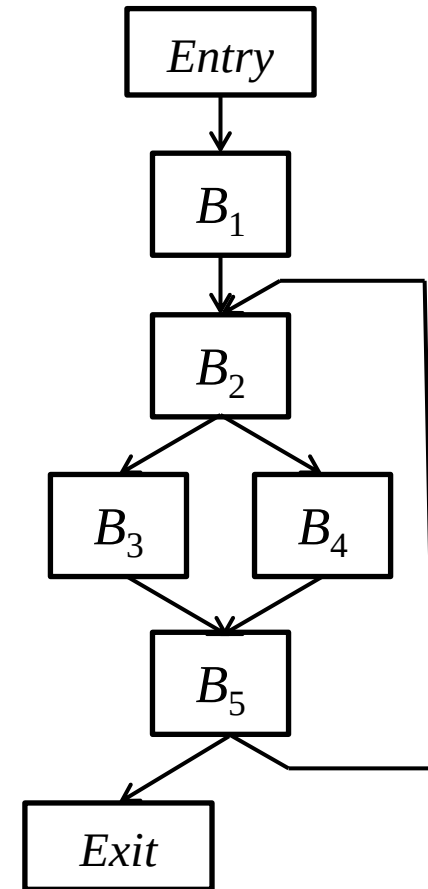
6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $\text{if } (i \% 2 = 0)$	B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$	
B_4 $e \leftarrow 3$		

Итак:

- ◇ инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла, так как выполняются в блоках, не являющихся доминаторами выхода из цикла – блока B_5
- ◇ инструкция $f \leftarrow e + 2$ не инвариантна относительно цикла, так как e может изменяться во время выполнения цикла

Эти инструкции нельзя выносить в предзаголовок B'_2 .



6.2. Перемещение кода, инвариантного относительно цикла

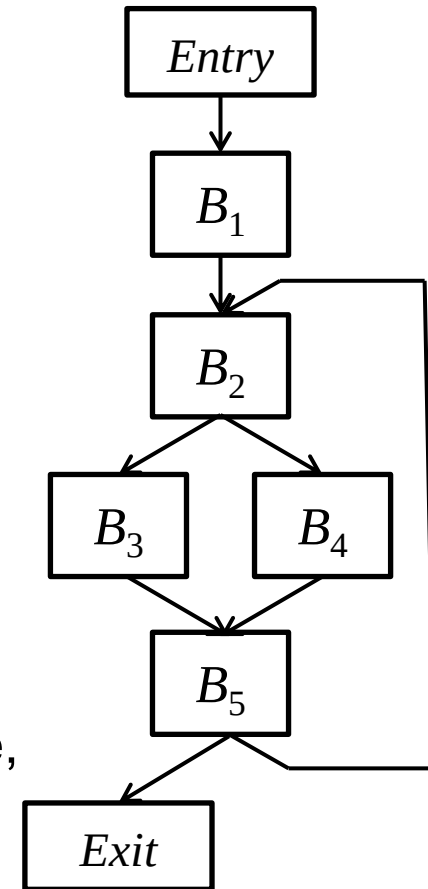
6.2.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$	B_2 $\text{if } (i \% 2 = 0)$	B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$
B_3 $e \leftarrow 2$	B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$	
B_4 $e \leftarrow 3$		



Таким образом выражение присваивания $x = e$, где e – инвариант цикла, можно выносить в предзаголовок цикла, если выполняются следующие три условия:

1. это единственное определение x в цикле,
2. оно является доминатором всех выходов из цикла, на которых x живо,
3. это единственное определение x , достигающее использований x внутри цикл: x не является живым на входе в заголовок цикла.



6.2. Перемещение кода, инвариантного относительно цикла

6.2.3 Алгоритм перемещения инвариантного кода

◇ Алгоритм:

1. Перед заголовком цикла вставить пустой базовый блок (будущий предзаголовок).

Для всех инструкций в теле цикла:

2. Отметить как инвариантные все операнды-константы
3. Отметить как инвариантные все операнды, у которых все определения, достигающие инструкции, находятся вне цикла
4. Отметить как инвариантные все инструкции, все операнды которых отмечены
5. Повторять шаги 2 – 4, пока инвариантные инструкции не перестанут выделяться
6. Переместить все выделенные инструкции в предзаголовок.

6.3. Другие оптимизации циклов

6.3.1 Обзор оптимизирующих преобразований циклов

- ◇ Вынесение инвариантного кода в предзаголовок (уже рассмотрели)
- ◇ Исключение умножений при вычислении индуктивных переменных (замена $t \leftarrow a*i + c$, где c и a – инварианты цикла, в частности – константы, на $t \leftarrow c + a$)
- ◇ Исключение проверок границ массивов
- ◇ Раскрутка циклов (чтобы сократить число проверок на окончание)
- ◇ Перестановка циклов в гнезде и другие преобразования, повышающие локальность данных, обрабатываемых в цикле (оптимизация работы с КЭШем)

6.4. Индуктивные переменные

6.4.1 Определение индуктивной переменной

- ◇ **Определение.** Переменная $\mathbf{v}$ называется *индуктивной переменной* цикла $\mathbf{L}$, если на каждой итерации цикла значение $\mathbf{v}$ увеличивается на значение переменной (или константы) $\mathbf{c}$, являющейся инвариантом цикла, то есть на каждом витке цикла выполняется инструкция:
$$\mathbf{v} \leftarrow +, \mathbf{v}, \mathbf{c}$$
- ◇ Тривиальным примером индуктивной переменной является счетчик цикла, то есть переменная $\mathbf{i}$, которой в начале цикла присваивается значение 0 (или 1) и значение которой на каждом витке цикла увеличивается на 1.
- ◇ Если индуктивная переменная $\mathbf{v}$ на каждой итерации цикла принимает значение $\mathbf{c} * \mathbf{i} + \mathbf{d}$, где $\mathbf{c}$ и $\mathbf{d}$ – инварианты цикла, то $\mathbf{v}$ является линейной индуктивной переменной.
- ◇ Когда в цикле удастся обнаружить индуктивные переменные, становится возможным выполнить различные оптимизации этого цикла.

6.4. Индуктивные переменные

6.4.2 Семейства индуктивных переменных

- ◇ Нетрудно доказать, что линейные функции от индуктивных переменных тоже индуктивные переменные.
Следовательно, в циклах, могут встречаться семейства индуктивных переменных. Обычно в циклах бывает несколько таких семейств.
- ◇ *Основной* индуктивной переменной по определению является индуктивная переменная i , все определения которой эквивалентны определению вида $i \leftarrow +, i, c$, где c – инвариант цикла (как правило, константа). Нет необходимости, чтобы значение c было одинаковым в каждом таком определении.
Основная индуктивная переменная является *линейной*, если в цикле имеется всего одно ее определение, причем это определение является доминатором (или постдоминатором) всех остальных вершин цикла.
- ◇ *Производная* индуктивная переменная j выражается через одну из основных индуктивных переменных i как $c * i + d$, где c и d – инварианты цикла.
- ◇ Основная индуктивная переменная i и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*.

6.4. Индуктивные переменные

6.4.2 Семейства индуктивных переменных

◇ Основная индуктивная переменная i и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*.

Для производной индуктивной переменной $j = c * i + d$ удобно использовать обозначение $j = \langle i, c, d \rangle$.

Применяя это обозначение к i , получим $i = \langle i, 1, 0 \rangle$

◇ В примере справа:

◇ i – основная индуктивная переменная;

◇ j – линейная основная индуктивная переменная;

◇ k и l – линейные производные индуктивные переменные семейства j ;

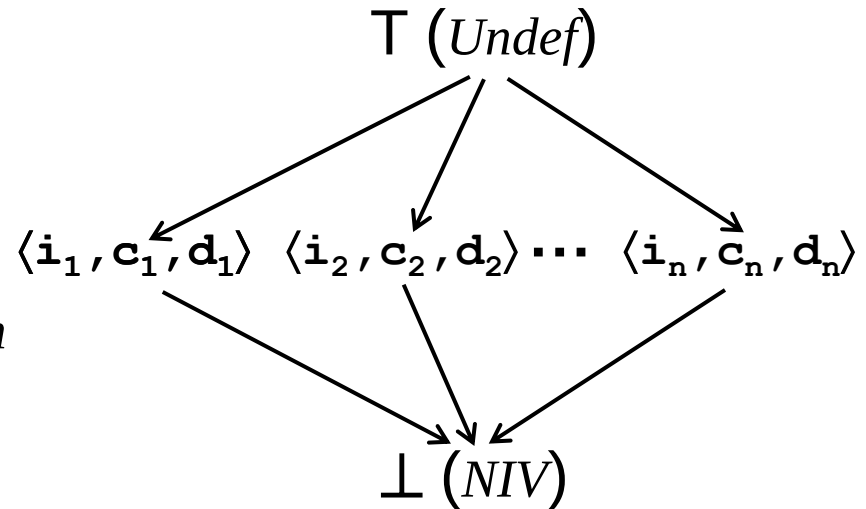
◇ m – производная индуктивная переменная семейства i .

```
while (i < 10) {  
  j = j + 2;  
  if (j > 4)  
    i = i + 1;  
  i = i - 1;  
  k = j + 10;  
  l = k * 4;  
  m = i * 8;  
}
```

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Для обнаружения индуктивных переменных используется анализ потока данных в пределах цикла.
- ◇ Элементами полурешетки, на которой проводится анализ, являются «тройки» $\langle i_k, c_k, d_k \rangle$ (см. рисунок).
- ◇ По определению элементы полурешетки, отличные от $\top$ и $\perp$, несравнимы один с другим.
- ◇ На элемент $\perp$ отображаются переменные, не являющиеся индуктивными (он так и называется *NIV* – *Not an Induction Variable*)
- ◇ На элемент $\top$ отображаются переменные, которым не было присвоено никаких значений, и по этой причине невозможно установить, является ли переменная индуктивной. Поэтому на элемент $\top$ отображаются переменные, имеющие значение «неопределенный (неинициализированный) элемент» – *Undef*.
В начале анализа всем переменным присваивается значение *Undef*.



6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

◇ Рассмотрим переменную x .

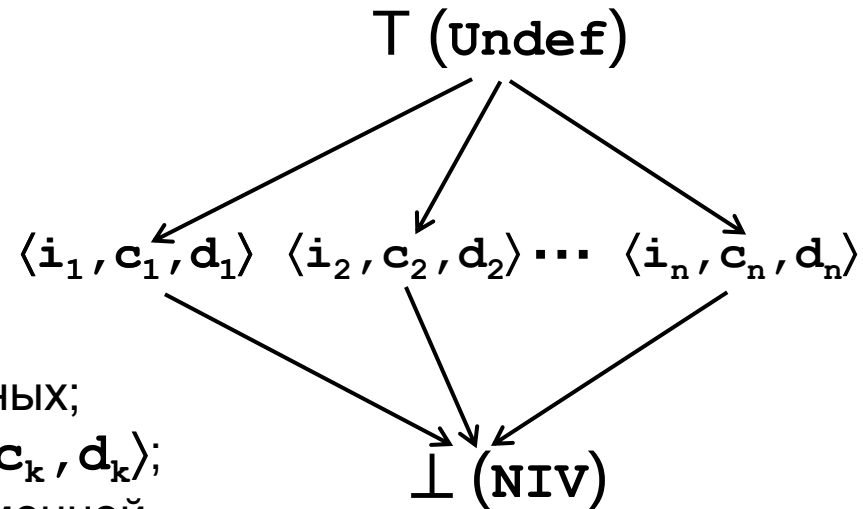
Возможны следующие варианты:

◇ x в рассматриваемом цикле ничего не присваивается; тогда над x надписывается **Undef**;

◇ x присваивается значение $c_k * i_k + d_k$, где i_k – одна из основных индуктивных переменных; тогда над x надписывается $\langle i_k, c_k, d_k \rangle$; над основной индуктивной переменной i_k надписывается $\langle i_k, 1, 0 \rangle$.

◇ Если x , над которым уже надписано $\langle i_k, c_k, d_k \rangle$ присваивается значение $c_m * i_m + d_m$, где i_m – другая основная индуктивная переменная; тогда старая надпись над x заменяется на **NIV**.

◇ Надписывание означает, что существует отображение (функция), которое каждой переменной, встречающейся в исследуемом цикле, ставит в соответствие необходимый элемент рассматриваемой полурешетки. На значениях этой функции будет задана передаточная функция анализа потока данных.



6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Чтобы начать анализ потока данных необходимо построить начальное отображение (**var**, **val**), где **var** – переменная, а **val** – ее значение. Эти пары помещаются в очередь **Worklist**.
- ◇ Сначала находят **основные** индуктивные переменные. Для этого просматриваются все инструкции цикла и находятся все переменные **i**, которым сначала присваивается целое значение **c** ($i \leftarrow c$), а потом ее значения изменяются только инструкциями вида $i \leftarrow +, i, d$, или $i \leftarrow +, d, i$, где **d** – инвариант цикла.
- ◇ Отметим, что присваивание $i \leftarrow c$ должно быть **только одно**. Если таких присваиваний несколько, т. е. есть **n** инструкций $i \leftarrow c_k$ с разными c_k , ($k = 1, \dots, n$), то переменная **i** **расщепляется** на **k** переменных i_k .
- ◇ Значением основной индуктивной переменной **i**, т. е. **val(i)** является «тройка» $\langle i, 1, 0 \rangle$, так как очевидно, что $i = 1*i + 0$. Каждой основной индуктивной переменной **i** соответствует пара $(i, \langle i, 1, 0 \rangle)$, которая помещается в **Worklist**.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Чтобы начать анализ потока данных необходимо построить начальное отображение (**var**, **val**), где **var** – переменная, а **val** – ее значение. Эти пары помещаются в очередь **Worklist**.
- ◇ Если обнаруживаются инструкции вида
$$p \leftarrow +, j, c \text{ или } q \leftarrow *, j, c,$$
где **j** – индуктивная переменная семейства **i**: $\text{val}(j) = \langle i, a, b \rangle$, а **c** – инвариант цикла, то **p** и **q** – тоже индуктивные переменные семейства **i**, причем
$$\text{val}(p) = \langle i, a, b+c \rangle \text{ и } \text{val}(q) = \langle i, a*c, b*c \rangle.$$
- ◇ Если обнаруживаются инструкции вида $p \leftarrow e$, где **e** – выражение, не содержащее индуктивных переменных, то $\text{val}(p) = \text{NIV}$
- ◇ Всем остальным переменным сопоставляется значение **T (Undef)**
- ◇ Все пары (**var**, **val**), найденные на первом этапе, кроме пар, у которых **val** = **NIV** помещаются в том порядке, в котором они были найдены в очередь **Worklist**.
- ◇ На этом первый этап алгоритма обнаружения индуктивных переменных заканчивается.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

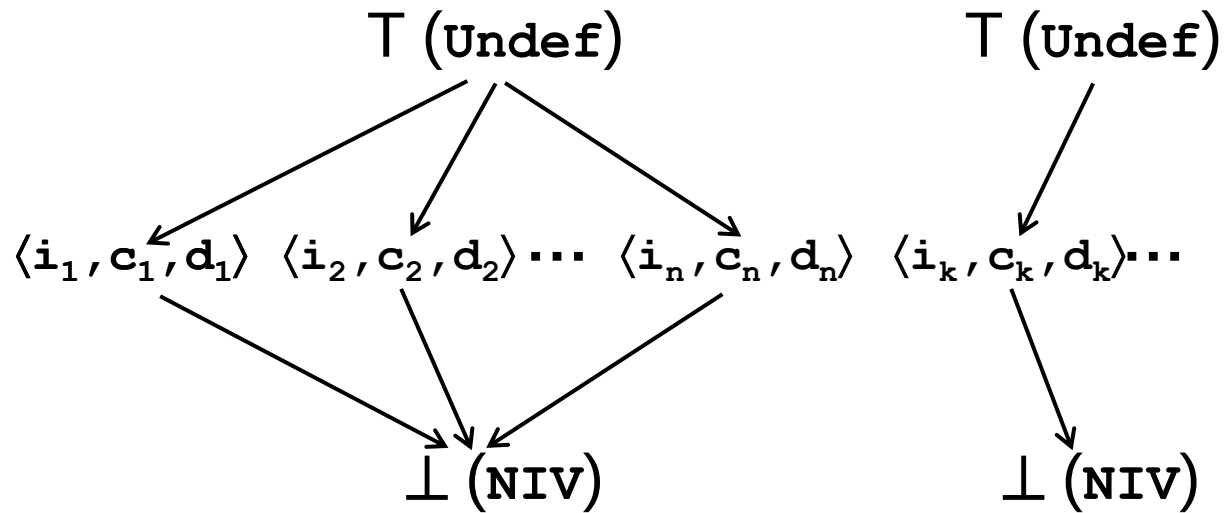
- ◇ После того, как начальное отображение построено и **Worklist** заполнен, начинается второй этап алгоритма обнаружения индуктивных переменных.
 - ◇ Из очереди **worklist** выбирается очередная пара (**var**, **val**)
 - ◇ Если эта пара имеет вид (**i**, $\langle \mathbf{i}, 1, 0 \rangle$), она помещается в список найденных индуктивных, начиная семейство **i**.

Теперь все индуктивные переменные, значение которых имеет первым членом «тройки» **i**, будут помещаться в это семейство.
 - ◇ Если эта пара имеет вид (**j**, $\langle \mathbf{i}, \mathbf{a}, \mathbf{b} \rangle$), выполняется обход ГПУ, чтобы выяснить, не превратится ли **val(j)** в **NIV**. Каждая пара анализируется независимо от других пар. Это связано со структурой полурешетки. В основном уточняются кандидаты в индуктивные переменные, имеющие значение **Undef**.
 - ◇ Анализ состоит в том, чтобы просмотреть все базовые блоки, в которых встречается **j** и убедиться, что во всех блоках **val(j)** остается в своем семействе индуктивных переменных. Основную роль при таком анализе играет операция сбора для функции **val**.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Каждое значение анализируется самостоятельно на полурешетке, изображенной на правом рисунке.



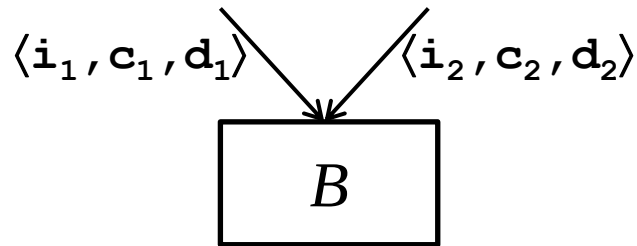
Анализ всех значений выполняется на декартовом произведении таких полурешеток

В результате анализа часть ИП станет **NIV**, а часть **Undef** станет ИП

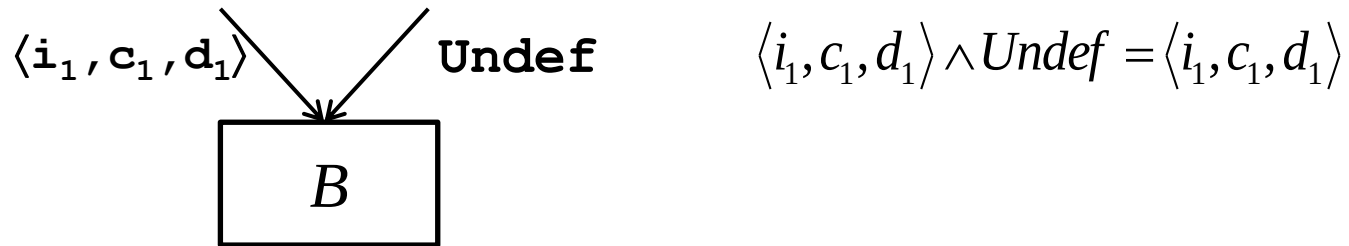
6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Каждое значение анализируется самостоятельно на полурешетке, изображенной на правом рисунке.



$$\langle i_1, c_1, d_1 \rangle \wedge \langle i_2, c_2, d_2 \rangle = \begin{cases} \langle i_1, c_1, d_1 \rangle & \text{если } i_1 = i_2, c_1 = c_2, d_1 = d_2 \\ NIV & \text{иначе} \end{cases}$$



$$\langle i_1, c_1, d_1 \rangle \wedge Undef = \langle i_1, c_1, d_1 \rangle$$

- ◇ В результате анализа будет получено несколько семейств индуктивных переменных и станут возможными оптимизации, связанные с индуктивными переменными – снижение сложности операций и исключение индуктивных переменных.

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций

- ◇ Будет показано, что для индуктивной переменной можно заменить умножение сложением (т. е. арифметическую операцию, выполняемую более сложным (и долгим) алгоритмом заменить на более простую и быструю). Причем при указанной замене количество выполняемых операций (инструкций) не изменится.
- ◇ Рассмотрим простейший пример:

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

Значения переменной j вычисляются с использованием умножения, но j – производная индуктивная переменная $\langle i, 3, a \rangle$, принадлежащая семейству основной индуктивной переменной i .

- ◇ Как вычислять j , используя сложение вместо умножения?

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, продолжение примера

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ Как вычислять j , используя сложение вместо умножения?
- ◇ Сделать это очень просто:
Для производной индуктивной переменной j из семейства основной индуктивной переменной i : $\text{val}(j) = \langle i, c, d \rangle$ необходимо выполнить следующий простой **алгоритм**:
 1. Создать новые переменные s и k , и в предзаголовке цикла выполнить присваивания $s = c*i + d$; и $k = c*h$;
 2. В теле цикла заменить определение $j = e$ на $j = s$;
 3. В теле цикла после присваивания $i = i + h$ вставить присваивание $j = j + k$;

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, продолжение примера

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

◇ Применив алгоритм к рассматриваемому циклу, получим

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    j = s;
    a[j] = a[j] + 1;
    i = i + 2;
    s = s + 6;
}
```

◇ И, что интересно, в преобразованном цикле нет явной зависимости между j и i : счетчик цикла и индекс элемента массива изменяются как бы независимо.

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, окончание примера



Таким образом, алгоритм снижения стоимости позволил заменить в теле цикла

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

умножение ($j = p + 3*i$) на сложение ($s = s + 6$),
в результате чего получился цикл

```
i = 1;
s = p + 3*i;          //предзаголовок цикла
while (i < a.length) {
    j = s;
    a[j] = a[j] + 1;
    i = i + 2;
    s = s + 6;
}
```

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

◇ Рассмотрим оптимизированный цикл

```
i = 1;  
s = p + 3*i;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```

◇ В нем три индуктивных переменных i , j и s .

Анализ покажет, что две из них лишние, и их можно исключить.

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    j = s;
    a[j] = a[j] + 1;
    i = i + 2;
    s = s + 6;
}
```



Во-первых, переменные j и s – это, по существу, одна переменная, так как s используется только для приращения индекса j .

Эти переменные (j и s) можно объединить в одну, например s и исключить лишнюю индуктивную переменную j .

Цикл после этого будет иметь вид (лишняя индуктивная переменная j исключена):

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    a[s] = a[s] + 1;
    i = i + 2;
    s = s + 6;
}
```

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    a[s] = a[s] + 1;
    i = i + 2;
    s = s + 6;
}
```

◇ **Во-вторых**, переменная i используется только в качестве счетчика цикла. Такие переменные называются *почти бесполезными*. Для того, чтобы исключить i , нужно вычислить в предзаголовке верхнюю границу t для s .

Вспомнив, что s – производная индуктивная переменная $\langle i, 3, a \rangle$ семейства i , получим $t = a + 3*a.length$

◇ После исключения i цикл примет окончательный вид:

```
i = 1;
s = p + 3*i;
t = 3*a.length
while (s < t) {
    a[s] = a[s] + 1;
    s = s + 6;
}
```

◇ **Предзаголовок** можно не оптимизировать: он выполняется всего один раз.

6.4. Индуктивные переменные

6.4.5 Исключение индуктивных переменных

- ◇ Таким образом, снижение стоимости и исключение (лишних) индуктивных переменных позволили оптимизировать исходный цикл

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ Получив следующий оптимизированный цикл:

```
i = 1;
s = p + 3*i;
t = 3*a.length
while (s < t) {
    a[s] = a[s] + 1;
    s = s + 6;
}
```

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ В языках *со строгой типизацией* (например, в языке *Java*), доступ к элементу массива по индексу возможен только тогда, когда индекс элемента не выводит за границы массива. Следовательно, транслятор *Java*-программы на байт-код, а значит и *JIT* должны вставлять в циклы соответствующие проверки: прежде, чем вычислить адрес элемента `a[j]`, компилятор должен убедиться, что `j` – безопасный индекс, не выходящий за пределы массива `a[]`.
- ◇ В частности, в цикл из примера 6.4.4 будет добавлена проверка, которая гарантирует, что индекс `j`, по которому производится доступ, удовлетворяет условиям.

$$0 \leq j < a.length.$$

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ В цикл, рассматриваемый в примере 6.4.4, добавится строка (она выделена **ЦВЕТОМ**):

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    if (i < 0 | i ≥ a.length) goto L_err;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ Наличие указанной строки внутри цикла существенно замедлит его выполнение (ведь на каждом витке цикла добавится два сравнения).

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

- ◇ Как показывает опыт программирования на языках *со слабой типизацией* (например, на любимом языке физиков *Fortran*), где такие проверки должен вставлять программист (если сочтет нужным), исключение таких проверок повышало скорость выполнения программы больше чем на порядок, но...
одновременно помогало хакерам и их вирусам проникать внутрь компьютера на котором программа выполнялась.
- ◇ Итак, **проблема в том, чтобы научиться удалять из цикла хотя бы часть проверок границ массивов, не уменьшая безопасности программы.**

6.5. Исключение проверок границ массивов

6.5.2 Решение проблемы

- ◇ Можно реализовать оператор

```
if (0 < i & i < n)
```

с помощью одной проверки (а не двух).

- ◇ Предположим, что n – положительное целое со знаком (`int n > 0`), а i – целое со знаком (`int i`). Тогда единственная проверка

```
(unsigned)i < n
```

обеспечит и $i < n$, и $i > 0$, так как при переводе отрицательного i в тип `unsigned`, i получит очень большое положительное значение.

Отметим, что многие современные процессоры имеют операцию «беззнаковое сравнение», так что все это можно реализовать.

Примером является операция **JAE** (**J**ump **A**bove or **E**qual) в процессорах *Intel*.

6.5. *Исключение проверок границ массивов*

6.5.2 Решение проблемы.

- ◇ Лучше всего, конечно, вообще исключить проверку, или, по крайней мере, вынести ее в предзаголовок.
 - ◇ Идея в том, что выполнение условия выхода из цикла (в рассматриваемом примере `i < a.length`) часто гарантирует, что обращения к массиву не выводят за его пределы.
 - ◇ Если в этом можно убедиться во время статического анализа, проверка может быть удалена.
 - ◇ Если это выясняется во время динамического анализа (или во время выполнения программы), цикл может быть представлен в двух версиях:
 - ◇ Быстрой, в которой нет проверок границ и
 - ◇ Медленной, в которой такие проверки есть.
- А пользователь пусть сам разбирается, где и когда какую версию использовать.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◇ Рассмотрим проблему исключения проверок границ массивов, когда выполняются следующие условия:

1. Индуктивная переменная j семейства i сравнивается с инвариантом цикла u (проверка условия выхода из цикла $j < u$).
2. Индуктивная переменная k того же семейства i сравнивается с инвариантом цикла n (проверка условия выхода из цикла $k < n$), причем из $j < u$ следует $k < n$.
3. k и j изменяются в одном и том же направлении.
4. Проверка условия $j < u$ находится в блоке, являющемся доминатором блока, в котором находится проверка условия $k < n$.

Утверждение. При выполнении условий 1, 2, 3 и 4 проверка условия $k < n$ избыточна и может быть удалена.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.



Для обоснования утверждения, выясним,
когда из $j < u$ следует $k < n$?

Пусть $j = \langle i, c_j, d_j \rangle$, $k = \langle i, c_k, d_k \rangle$.

Если условие $j < u$ выполняется, то

$$c_j * i + d_j < u$$

Без потери общности можно считать, что $c_j > 0$. Тогда

$$i < \frac{(u - d_j)}{c_j}$$

Следовательно,

$$k = c_k * i + d_k < c_k * \left(\frac{(u - d_j)}{c_j} \right) + d_k$$

Если удастся показать в результате статического или динамического анализа, что правая часть последнего неравенства $\leq n$, то условие $k < n$ будет выполнено.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◇ Итак, необходимо выяснить, верно ли что

$$c_k * \left(\frac{(u - d_j)}{c_j} \right) + d_k \leq n \quad (*)$$

Это можно либо установить во время компиляции, либо (если статический анализ недостаточен) попытаться поместить соответствующую проверку в предзаголовок цикла.

```
s = p + 3*i;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```

◇ В нашем примере условие `i < a.length` оказывается достаточным для одновременной проверки и верхней границы массива `a[]`. В этом нетрудно убедиться во время компиляции подставив $c_i = 1$, $d_i = 0$, вместо (c_j, d_j) и (c_k, d_k) в формулу (*). Получим верное неравенство:

$$1 * (a.length - 0) / 1 + 0 \leq a.length, \text{ или } a.length \leq a.length.$$

Что касается 0, левой границы для `i`, то проверку условия `i > 0` можно вынести в предзаголовок (так как 0 – инвариант цикла).

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

```
s = p + 3*i;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```



В итоге, получим следующий цикл, в котором, конечно же, можно выполнить оптимизации, связанные с удалением лишних индуктивных переменных.

```
s = p + 3*i;  
if (i < 0) goto L_err;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```

L_{err} – метка, начиная с которой помещен код обработки исключения «выход за границы массива».

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

```
s = p + 3*i;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```



В итоге, получим следующий цикл, в котором, конечно же, можно выполнить оптимизации, связанные с удалением лишних индуктивных переменных.

```
s = p + 3*i;  
if (i < 0) goto L_err;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```

Вместо двух проверок
внутри цикла, одна простая
в предзаголовке

L_{err} – метка, начиная с которой помещен код обработки исключения «выход за границы массива».

6.6. Раскрутка циклов

6.6.1 Что это такое

◇ **Простой пример.** Выполним раскрутку внутреннего цикла в гнезде.

◇ Исходное гнездо циклов (на языке C):

```
for(j = 1; j <= nj; j++){  
    for(i = 1; i <= ni; i++){  
        y[i] += x[j] * m[i][j];  
    }  
}
```

◇ После раскрутки внутреннего цикла на 4:

```
for(j = 1; j <= nj; j++){  
    mod_i = ni % 4;  
    if(mod_i >= 1){  
        for(i = 1; i <= mod_i; i++){  
            y[i] += x[j] * m[i][j];  
        }  
    }  
    for(i = mod_i + 1; i <= ni; i += 4){  
        y[i] += x[j] * m[i][j];  
        y[i+1] += x[j] * m[i+1][j];  
        y[i+2] += x[j] * m[i+2][j];  
        y[i+3] += x[j] * m[i+3][j];  
    }  
}
```

6.6. Раскрутка циклов

6.6.1 Что это такое

- ◇ **Простой пример.** Выполним раскрутку внутреннего цикла в гнезде.
- ◇ Исходное гнездо циклов (на языке C):

```
for(i = 1; i <= ni; i++) {
```

Раскрутка цикла

копирует тело цикла для нескольких итераций и корректирует вычисление индексов соответствующим образом.

- ◇ После раскрутки внутреннего цикла на 4:

```
for(j = 1; j <= nj; j++) {  
    mod_i = ni % 4;  
    if(mod_i >= 1) {
```

Если ni не делится на 4, то остается кусок цикла, который выполняется в коротком *цикле-прологе*.

Назначение цикла-пролога – гарантировать, что количество итераций цикла раскручиваемого на m , кратно m .

Такое преобразование цикла (расщепление цикла для обеспечения удобных границ) называется *выравниванием цикла*.

6.6. Раскрутка циклов

6.6.1 Что это такое.

◇ Простой пример. Но можно выполнить и раскрутку внешнего цикла.

◇ После раскрутки внешнего цикла на 4:

```
mod_j = nj % 4;
if(mod_j >= 1){
    for(j = 1; j <= mod_j; j++){
        for(i = 1; i < ni; i++){
            y[i] += x[j] * m[i][j];
        }
    }
}
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+1] * m[i][j+1];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+2] * m[i][j+2];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+3] * m[i][j+3];
    };
}
```


6.6. Раскрутка циклов

6.6.1 Что это такое.

◇ Простой пример. Но можно

◇ После раскрутки вне

```
mod_j = nj % 4;
if(mod_j >= 1){
    for(j = 1; j <= mod_j; j++){
        for(i = 1; i < ni; i++){
            y[i] += x[j] * m[i][j];
        }
    }
}
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+1] * m[i][j+1];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+2] * m[i][j+2];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

```
for(j = 1; j < nj; j++){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    }
}
```

6.6. Раскрытие циклов

6.6.2 Слияние циклов.

- ◇ Слияние циклов – объединение двух циклов с одинаковыми границами изменения индекса и шагом в один.
Слияние корректно, когда каждое определение и каждое использование в результирующем (слитом) цикле имеют такие же значения, что и в исходных (сливаемых) циклах.
- ◇ В рассматриваемом простом примере можно выполнить три слияния циклов.

◇ До слияния внутренних циклов:

```
for(j = mod_j + 1; j <= nj; j += 4){  
    for(i = 1; i < ni; i++){  
        y[i] += x[j] * m[i][j];  
    };  
    for(i = 1; i < ni; i++){  
        y[i] += x[j+1] * m[i][j+1];  
    };  
    for(i = 1; i < ni; i++){  
        y[i] += x[j+2] * m[i][j+2];  
    };  
    for(i = 1; i < ni; i++){  
        y[i] += x[j+3] * m[i][j+3];  
    };  
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

- ◇ Слияние циклов – объединение двух циклов с одинаковыми границами изменения индекса и шагом в один.
- ◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ После первого слияния:

```
for(j = mod_j + 1; j <= nj; j += 4){  
    for(i = 1; i < ni; i++){  
        y[i] += x[j] * m[i][j];  
        y[i] += x[j+1] * m[i][j+1];  
    };  
    for(i = 1; i < ni; i++){  
        y[i] += x[j+2] * m[i][j+2];  
    };  
    for(i = 1; i < ni; i++){  
        y[i] += x[j+3] * m[i][j+3];  
    };  
}
```

6.6. Раскрытие циклов

6.6.2 Слияние циклов.

◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ После третьего слияния:

```
for(j = mod_j + 1; j <= nj; j += 4){  
    for(i = 1; i < ni; i++){  
        y[i] += x[j] * m[i][j];  
        y[i] += x[j+1] * m[i][j+1];  
        y[i] += x[j+2] * m[i][j+2];  
        y[i] += x[j+3] * m[i][j+3];  
    };  
}
```

◇ Последний цикл можно преобразовать к виду:

```
for(j = mod_j + 1; j <= nj; j += 4){  
    for(i = 1; i < ni; i++){  
        y[i] = y[i] + x[j] * m[i][j] + x[j+1] * m[i][j+1]  
            + x[j+2] * m[i][j+2] + x[j+3] * m[i][j+3];  
    };  
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

◇ В рассматриваемом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ Таким образом, оптимизируя цикл

```
mod_j = nj % 4;
if(mod_j >= 1){
    for(j = 1; j <= mod_j; j++){
        for(i = 1; i < ni; i++){
            y[i] += x[j] * m[i][j];
        }
    }
},
```

получаем окончательно следующий цикл:

```
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
        y[i] += x[j+1] * m[i][j+1];
        y[i] += x[j+2] * m[i][j+2];
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ Таким образом, оптимизируя цикл

```
mod_j = nj % 4;  
if(mod_j >= 1){  
    for(i = 1; i <= mod_j; i++) {
```

Рассмотренное оптимизирующее преобразование называется *раскруткой с последующим сжатием*.

```
    }
```

```
}
```

Если у сливаемых циклов границы изменения переменной цикла i не совпадают, можно применить *выравнивание* циклов.

```
for(j = mod_j + 1; j <= nj; j += 4){  
    for(i = 1; i < ni; i++){  
        y[i] += x[j] * m[i][j];  
        y[i] += x[j+1] * m[i][j+1];  
        y[i] += x[j+2] * m[i][j+2];  
        y[i] += x[j+3] * m[i][j+3];  
    };  
}
```

6.6. Раскрутка циклов

6.6.3 Сравнение двух видов раскрутки циклов.

- ◇ Раскрутка внутреннего цикла производит код, **выполняющий намного меньше проверок на выход из цикла**. Доступ к двумерному массиву $m[i][j]$ последователен, так как С-массив располагается по строкам.
- ◇ Раскрутка внешнего цикла не только сокращает количество проверок на выход из цикла, но и (особенно в случае раскрутки с последующим сжатием) обеспечивает повторное использование $y[i]$, а также последовательный доступ как к элементам x , так и к элементам m . Увеличение повторного использования данных существенно изменяет соотношение между арифметическими операциями и операциями доступа к памяти в цикле, повышая *локальность* данных.
- ◇ Кроме того, при каждом подходе могут проявляться и другие прямые и косвенные улучшения кода. Окончательная производительность цикла зависит от всех улучшений, как прямых, так и косвенных.
- ◇ Важным косвенным улучшением помимо увеличения локальности данных является увеличение количества операций в теле цикла.
- ◇ Раскрутка цикла позволяет реализовать его параллельное выполнение (этот вопрос будет рассмотрен при планировании кода).

6.7. Заключительные замечания

6.7.1 Другие преобразования циклов.

- ◇ В компиляторах применяются и другие преобразования циклов. Но эти преобразования применяются во время планирования кода для обеспечения параллельного выполнения инструкций программы на каждом ядре процессора. Эти преобразования будут рассмотрены во время изучения темы 17 «Оптимизация циклов (часть II)».
- ◇ В частности будут рассмотрены преобразования циклов, используемые для повышения локальности данных. Эти преобразования позволяют сократить количество перезагрузок кэша при выполнении цикла, что приводит к существенному увеличению скорости выполнения программы.
Более того, будет показано, что обеспечение локальности данных является необходимым условием эффективного распараллеливания программы.

7. Форма статического единственного присваивания (SSA-форма)

7.1 Форма статического единственного присваивания (SSA-форма)

7.1.1. Постановка задачи

- ◇ *Форма статического единственного присваивания (SSA)* позволяет в каждой точке программы объединить
 - ◇ информацию об имени переменной
 - ◇ и информацию о текущем значении этой переменной (или, что то же самое, информацию о том, *какое из определений данной переменной определяет ее текущее значение в данной точке в данный момент*).

7.1 Форма статического единственного присваивания (SSA-форма)

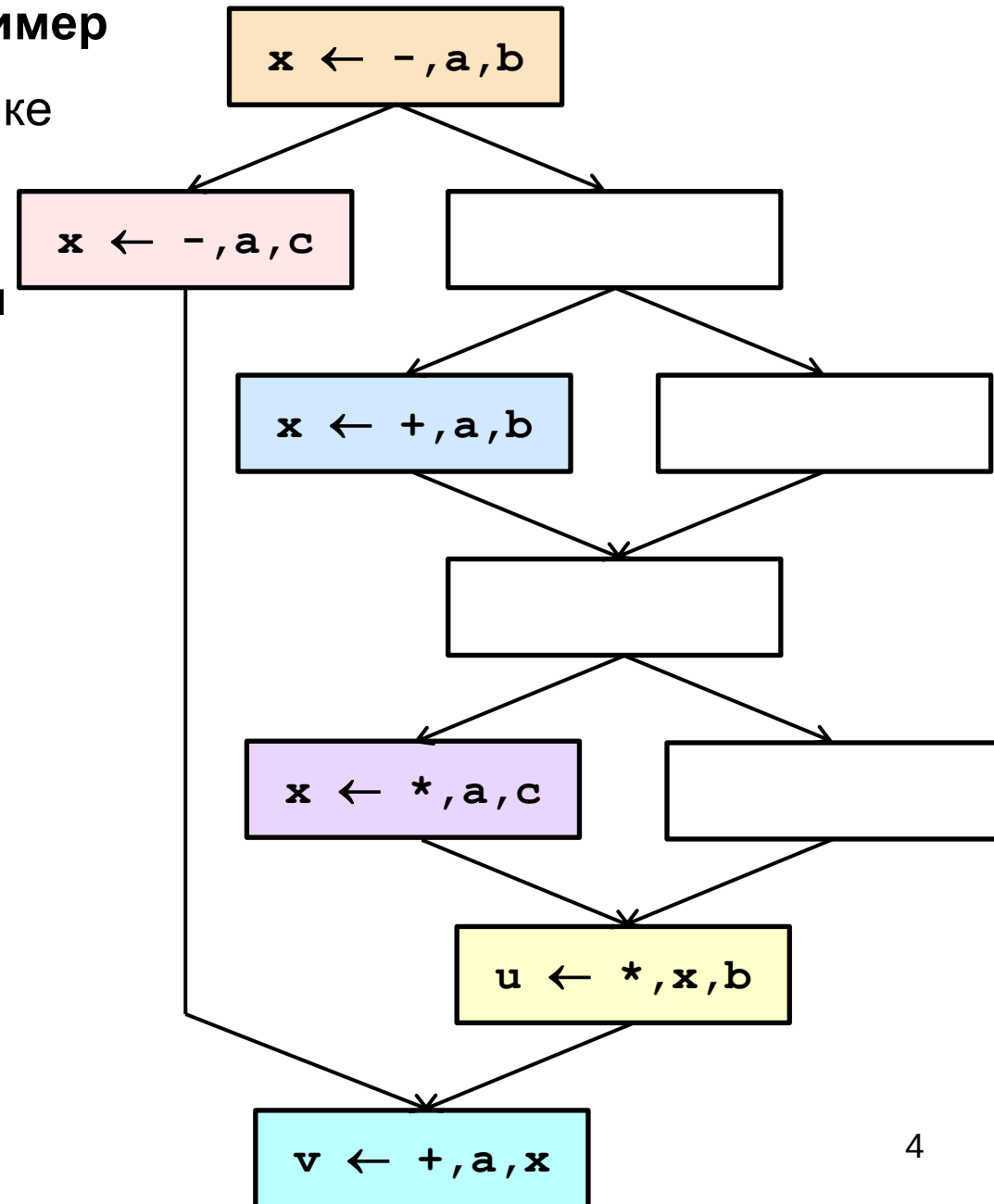
7.1.1. Постановка задачи

- ◇ *Форма статического единственного присваивания (SSA)* позволяет в каждой точке программы объединить
 - ◇ информацию об имени переменной
 - ◇ и информацию о текущем значении этой переменной (или, что то же самое, информацию о том, *какое из определений данной переменной определяет ее текущее значение в данной точке в данный момент*).
- ◇ Хотелось бы, чтобы программа в SSA-форме удовлетворяла двум условиям:
 - (1) каждое определение переменной имеет индивидуальное имя;
 - (2) каждое использование переменной ссылается на единственное определение.

7.1 SSA-форма

7.1.1. Постановка задачи. Пример

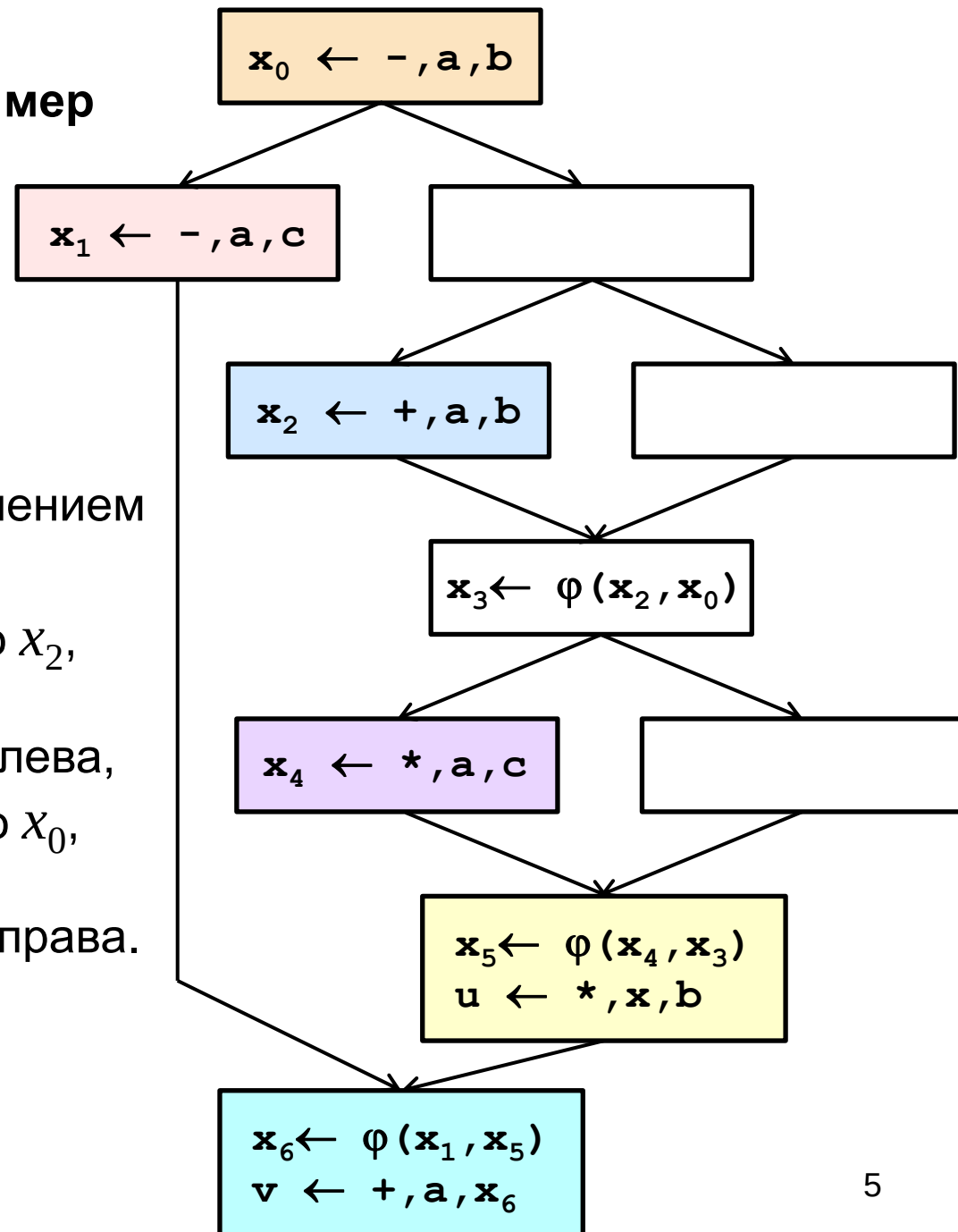
- ◇ Фрагмент ГПУ на рисунке содержит четыре определения **x**
- ◇ Исполновения в синем блоке достигают три определения **x**, в зеленом – четыре определения **x**
- ◇ Цель же состоит в том, чтобы *каждого использования* достигало *только одно определение*



7.1 SSA-форма

7.1.1. Постановка задачи. Пример

- ◇ Введем «функцию» объединения значений или ϕ -функцию
- ◇ По определению $x_3 \leftarrow \phi(x_2, x_0)$ является новым определением переменной x :
 - ◇ значение x_3 равно x_2 , когда управление попадает в блок слева,
 - ◇ значение x_3 равно x_0 , когда управление попадает в блок справа.



7.1 SSA-форма

7.1.2. Определение ϕ -функции

- ◇ ϕ -функция определяет *SSA-имя* для значения своего аргумента, соответствующего ребру, по которому управление входит в блок.
- ◇ **При входе в базовый блок все его ϕ -функции выполняются одновременно и до любого другого оператора, определяя целевые *SSA-имена*.**

7.1 SSA-форма

7.1.2. Определение ϕ -функции. Пример

```
x =  
y = ...  
while (x < 100) {  
    x = x + 1  
    y = y + x  
}
```

7.1 SSA-форма

7.1.2. Определение ϕ -функции. Пример

```
x =  
y = ...  
while (x < 100) {  
    x = x + 1  
    y = y + x  
}
```

```
      x0 = ...  
      y0 = ...  
      if (x0 ≥ 100) goto next  
loop:  x1 =  $\phi(x_0, x_2)$   
      y1 =  $\phi(y_0, y_2)$   
      x2 = x1 + 1  
      y2 = y1 + x2  
      if (x2 < 100) goto loop  
next:  x3 =  $\phi(x_0, x_2)$   
      y3 =  $\phi(y_0, y_2)$ 
```


7.1 SSA-форма

7.1.2. Определение ϕ -функции. Пример

```
x =  
y = ...  
while (x < 100) {  
    x = x + 1  
    y = y + x  
}
```

```
      x0 = ...  
      y0 = ...  
      if (x0 ≥ 100) goto next  
loop: x1 =  $\phi(x_0, x_2)$   
      y1 =  $\phi(y_0, y_2)$   
      x2 = x1 + 1  
      y2 = y1 + x2  
      if (x2 < 100) goto loop  
next: x3 =  $\phi(x_0, x_2)$   
      y3 =  $\phi(y_0, y_2)$ 
```

Каждая из ϕ -функций
объединяет значения своих
аргументов в новое значение,
определением которого она
является.

7.1 SSA-форма

7.1.2. Определение ϕ -функции. Пример

```
x =  
y = ...  
while (x < 100) {  
    x = x + 1  
    y = y + x  
}
```

```
      x0 = ...  
      y0 = ...  
      if (x0 ≥ 100) goto next  
loop: x1 =  $\phi(x_0, x_2)$   
      y1 =  $\phi(y_0, y_2)$   
      x2 = x1 + 1  
      y2 = y1 + x2  
      if (x2 < 100) goto loop  
next: x3 =  $\phi(x_0, x_2)$   
      y3 =  $\phi(y_0, y_2)$ 
```

Каждая из ϕ -функций
объединяет значения своих
аргументов в новое значение,
определением которого она
является.

До цикла –	x₀, y₀
На входе в цикл –	x₁, y₁
Внутри цикла –	x₂, y₂
После цикла –	x₃, y₃

7.1 SSA-форма

7.1.2. Определение ϕ -функции. Пример

<pre>x = y = ... while (x < 100) { x = x + 1 y = y + x }</pre>	<pre>x₀ = ... y₀ = ... if (x₀ ≥ 100) goto next loop: x₁ = $\phi(x_0, x_2)$ y₁ = $\phi(y_0, y_2)$ x₂ = x₁ + 1 y₂ = y₁ + x₂ if (x₂ < 100) goto loop next: x₃ = $\phi(x_0, x_2)$ y₃ = $\phi(y_0, y_2)$</pre>



Затруднение. Первый аргумент $\phi(x_0, x_2)$ определяется в блоке, который предшествует циклу, второй аргумент определяется позже в блоке, содержащем $\phi(x_0, x_2)$. Следовательно, при первом выполнении $\phi(x_0, x_2)$ ее второй аргумент еще не определен.

7.1 SSA-форма

7.1.2. Определение φ -функции. Пример

<pre>x = y = ... while (x < 100) { x = x + 1 y = y + x }</pre>	<pre>x₀ = ... y₀ = ... if (x₀ ≥ 100) goto next loop: x₁ = $\varphi(x_0, x_2)$ y₁ = $\varphi(y_0, y_2)$ x₂ = x₁ + 1 y₂ = y₁ + x₂ if (x₂ < 100) goto loop next: x₃ = $\varphi(x_0, x_2)$ y₃ = $\varphi(y_0, y_2)$</pre>



Выход из затруднения: по определению любая φ -функция (а значит и $\varphi(x_0, x_2)$) читает только один из своих аргументов, а именно тот аргумент, который соответствует последнему пройденному ребру ГПУ.

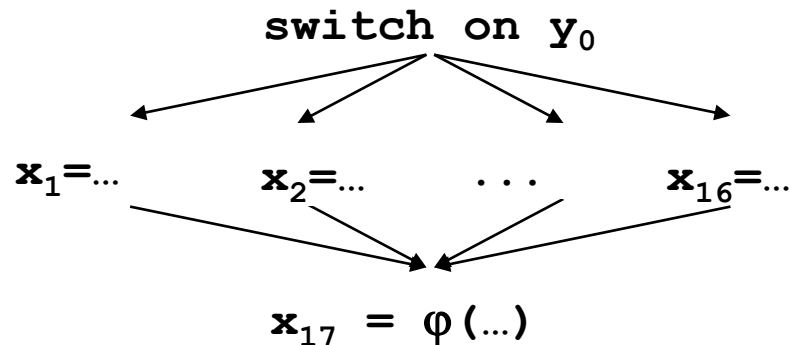
Поэтому φ -функция не прочитает неопределенного значения.

7.1 SSA-форма

7.1.3. Количество аргументов φ -функции

- ◇ По определению у φ -функции может быть любое число аргументов.

Например, на рисунке φ -функция, у которой 16 аргументов



7.2 Построение SSA-формы

7.2.1. Постановка задачи

- ◇ Для преобразования процедуры в SSA-форму **компилятор** должен:
 - ◇ вставить в точки сбора необходимые ϕ -функции для каждой переменной
 - ◇ переименовать переменные (в том числе и временные) таким образом, чтобы выполнялись следующие два правила:
 - (1) каждое определение имеет индивидуальное имя; и
 - (2) каждое использование ссылается на единственное определение.

7.2 Построение SSA-формы

7.2.2. Базовый алгоритм построения SSA-формы

- ◇ **Вход:** программа в промежуточном представлении
- ◇ **Выход:** промежуточное представление программы в SSA-форме
- ◇ **Метод:** Выполнить следующие действия:

(1) *Вставить φ -функции:*

в начало каждого блока B , у которого

$|Pred(B)| > 1$ вставить φ -функцию вида

$y = \varphi(y, y, \dots)$ для каждого имени y , которое либо определяется, либо используется в B .

Вставленная φ -функция должна иметь по одному аргументу для каждого $B' \in Pred(B)$:

Порядок вставляемых φ -функций несуществен

7.2 Построение SSA-формы

7.2.2. Базовый алгоритм построения SSA-формы

- ◇ **Вход:** программа в промежуточном представлении
- ◇ **Выход:** промежуточное представление программы в SSA-форме
- ◇ **Метод:** Выполнить следующие действия:
 - (2) *Переименовать переменные:*
 - (a) вычислить достигающие определения; при этом только одно определение будет достигать любого использования; это гарантируют вставленные ϕ -функции, которые тоже являются определениями;
 - (b) переименовать каждое использование (как основных, так и временных) переменных таким образом, чтобы новое имя соответствовало единственному определению, которое достигает его.

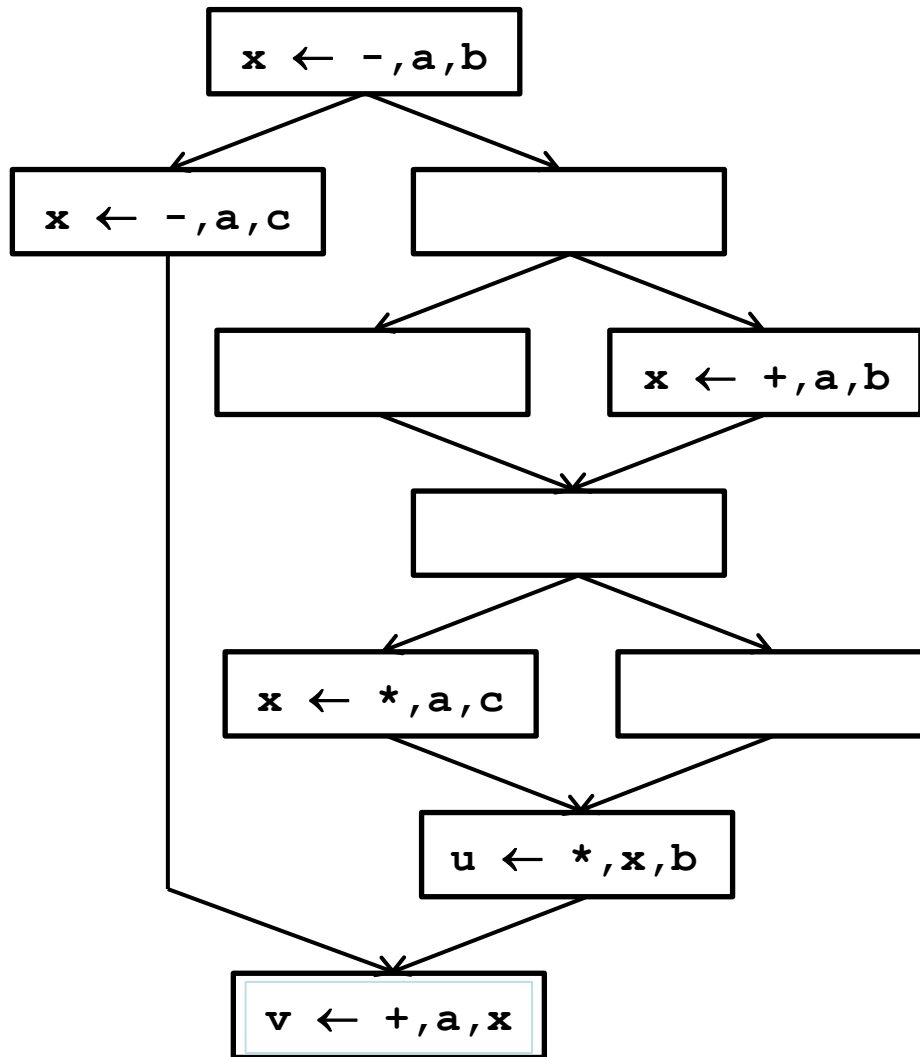
7.2 Построение SSA-формы

7.2.2. Базовый алгоритм построения SSA-формы

- ◇ **Вход:** программа в промежуточном представлении
- ◇ **Выход:** промежуточное представление программы в SSA-форме
- ◇ **Метод:** Выполнить следующие действия:
 - (3) *Отсортировать определения,*
достигающие каждой φ -функции и для каждой φ -функции обеспечить соответствие имен ее аргументов путям, по которым определения этих аргументов достигают блока, содержащего указанную φ -функцию.

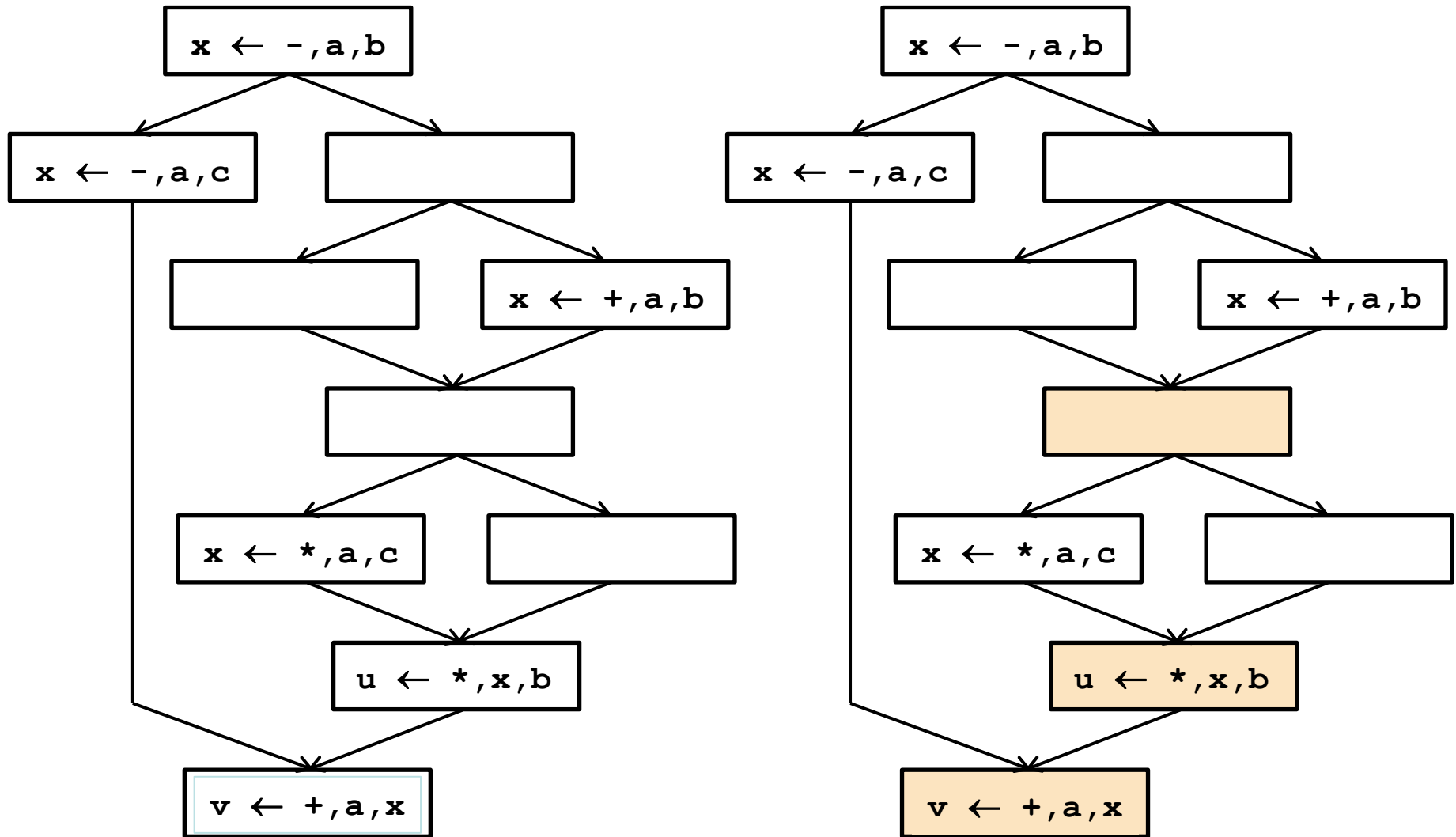
7.2 Построение SSA-формы

7.2.3. Базовый алгоритм построения SSA-формы. Пример



7.2 Построение SSA-формы

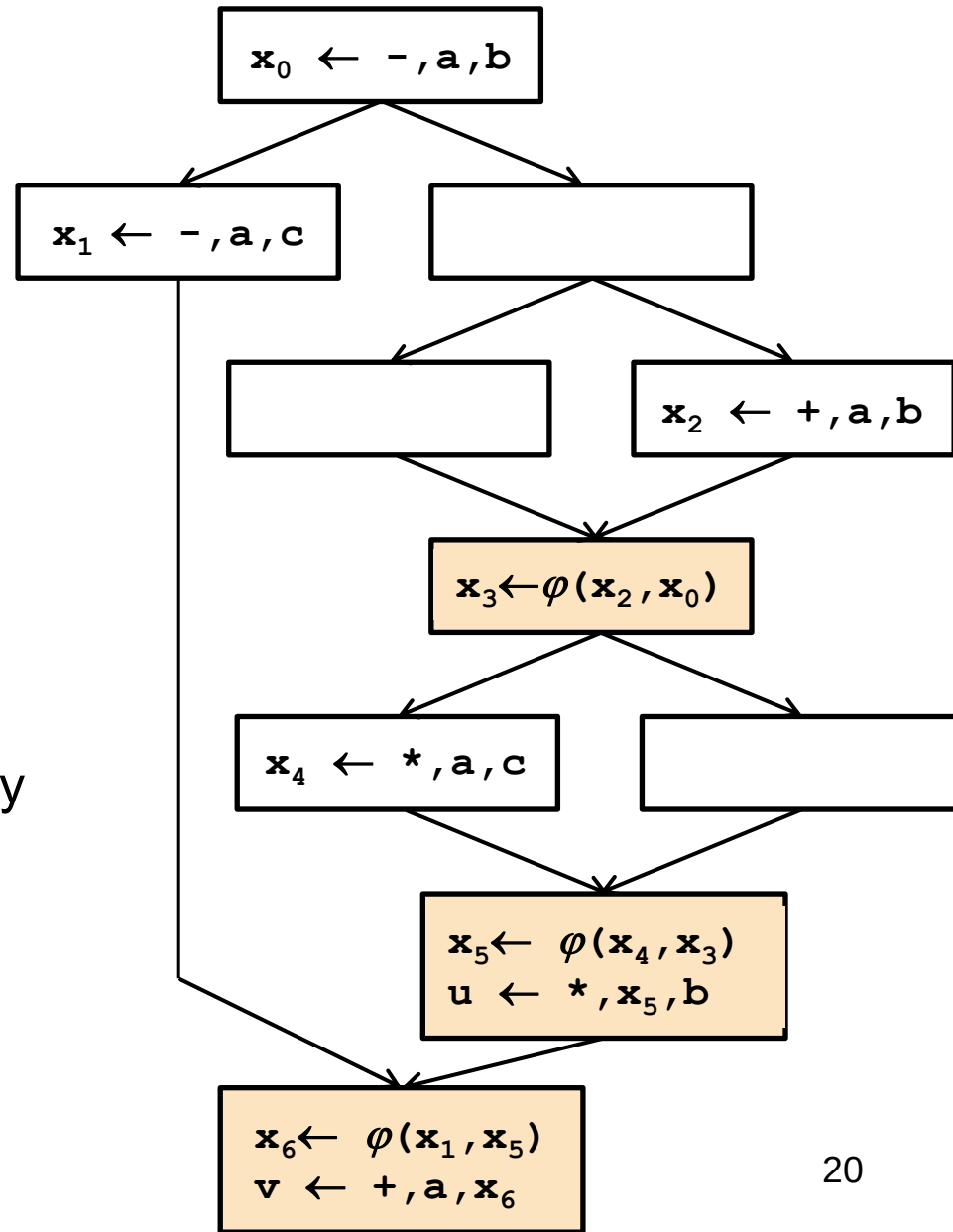
7.2.3. Базовый алгоритм построения SSA-формы. Пример



7.2 Построение SSA-формы

7.2.3. Базовый алгоритм построения SSA-формы. Пример

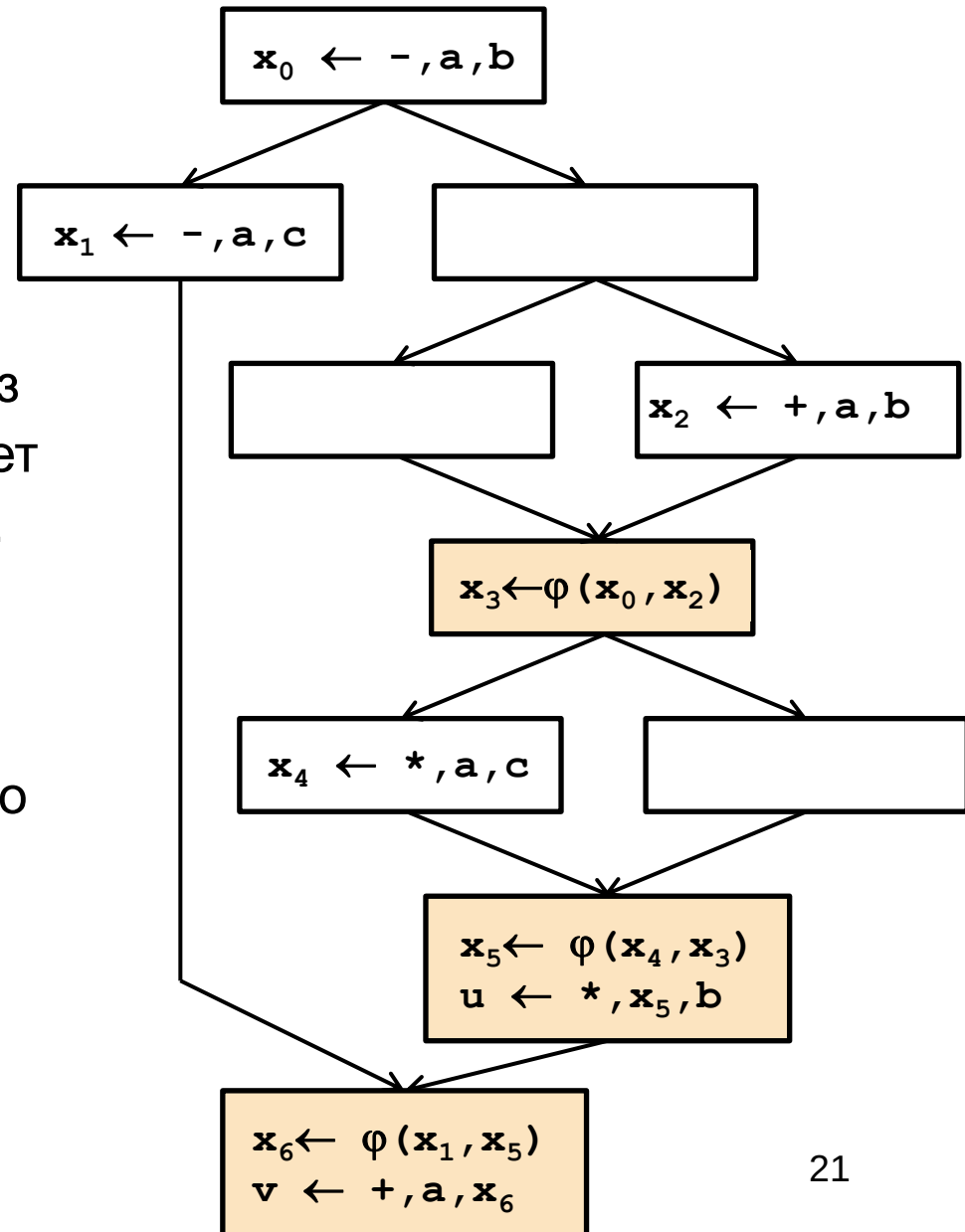
- ◇ Алгоритм вставил φ -функцию после каждой точки сбора ГПУ.
- ◇ После преобразования процедуры в SSA-форму
 - (1) внутри процедуры каждое определение создает уникальное имя;
 - (2) каждое использование обращается к единственному определению.



7.2 Построение SSA-формы

7.2.3. Базовый алгоритм построения SSA-формы. Пример

- ◇ Построенная *SSA-форма* корректна:
 - ◇ Каждая переменная определяется только один раз
 - ◇ Каждое обращение использует имя отдельного определения.
- ◇ Построенная *SSA-форма* называется *максимальной SSA-формой*, так как она обычно содержит **намного больше** ϕ -функций, чем необходимо.



7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи

- ◇ По определению *частично усеченная SSA-форма* содержит меньше φ -функций, чем максимальная (но, к сожалению, как следует и из ее названия она **не всегда** содержит минимально возможное количество φ -функций).
- ◇ Частично-усеченная SSA-форма – один из вариантов «минимальной» SSA-формы, разработанных в конце прошлого века. Практика показала, что все эти формы дают для большей части программ одинаковые или незначительно отличающиеся результаты.
Наиболее простой алгоритм построения у частично-усеченной SSA-формы

7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи

- ◇ По определению *частично усеченная SSA-форма* содержит меньше φ -функций, чем максимальная (но, к сожалению, как следует и из ее названия она **не всегда** содержит минимально возможное количество). Чтобы не всегда вставлять φ -функции необходимо **для каждой точки сбора** уметь выяснять, какие переменные нуждаются в φ -функциях, а какие нет.
 - ◇ Частично усеченная SSA-форма была предложена в начале прошлого века. «минимальная»
- Практика показала, что все эти формы дают для большей части программ одинаковые или незначительно отличающиеся результаты.
- Наиболее простой алгоритм построения у частично-усеченной SSA-формы

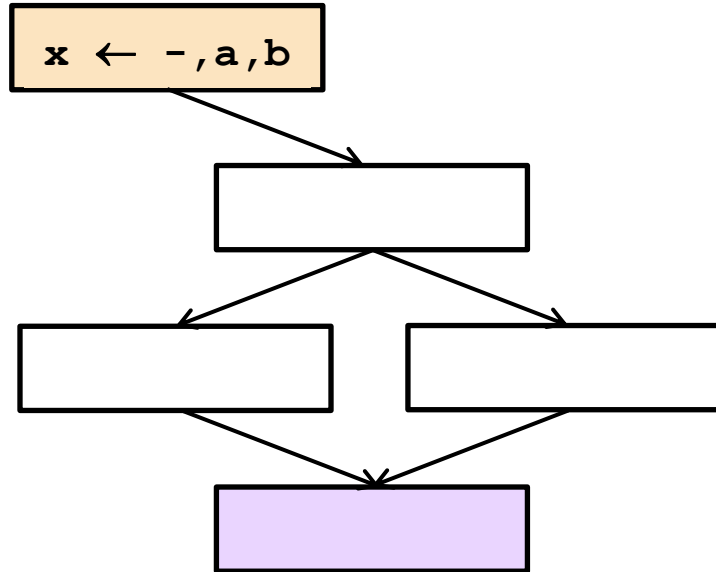
7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи

- ◇ По определению *частично усеченная SSA-форма* содержит меньше φ -функций, чем максимальная (но, к сожалению, как следует и из ее названия она **не всегда** содержит минимально возможное количество). Чтобы не всегда вставлять φ -функции необходимо **для каждой точки сбора** уметь выяснять, какие переменные нуждаются в φ -функциях, а какие нет. Или «минимализация» SSA-формы — задача прошлого века. Практика: **для каждого определения переменной** уметь найти часть программы, порожденной этим определением. Наименее эффективная SSA-форма
- ◇ Частично усеченная SSA-форма — это такая форма, в которой для каждой точки сбора уметь выяснять, какие переменные нуждаются в φ -функциях, а какие нет. Или «минимализация» SSA-формы — задача прошлого века. Практика: **для каждого определения переменной** уметь найти часть программы, порожденной этим определением. Наименее эффективная SSA-форма

7.3 Построение частично усеченной SSA-формы

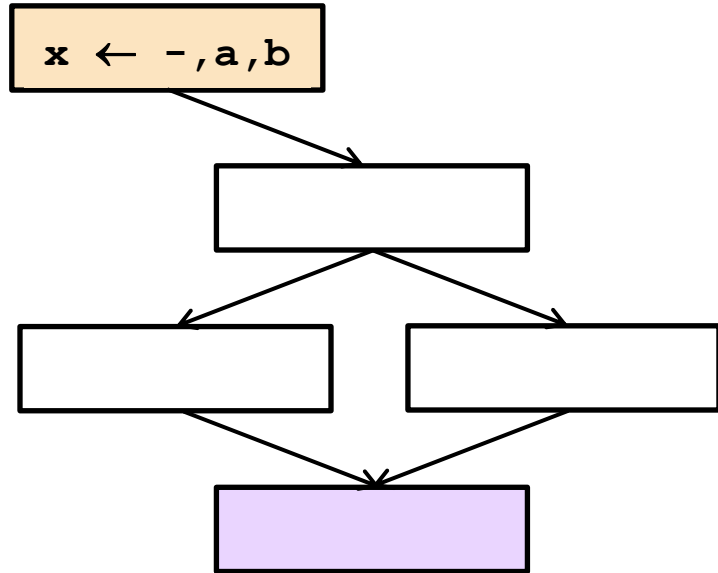
7.3.1. Постановка задачи



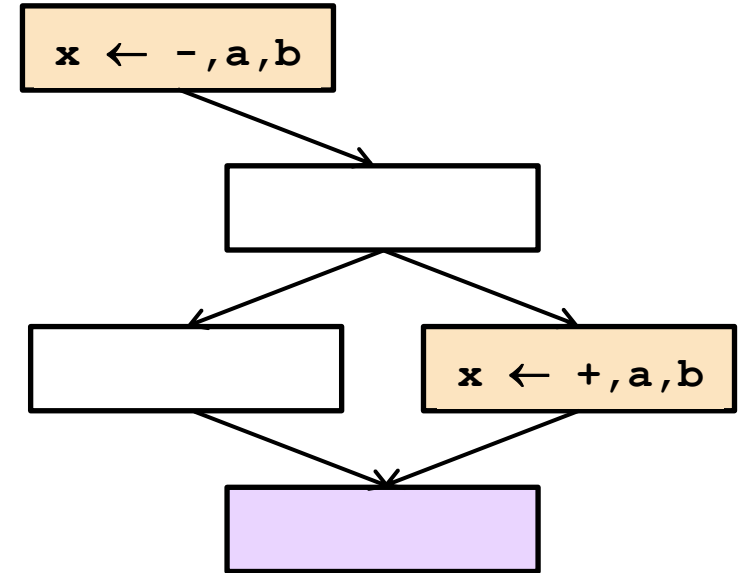
В лиловом блоке φ -функция для $\mathbf{x}$ не нужна, так как и слева, и справа приходит одно и то же значение $\mathbf{x}$

7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи



В лиловом блоке φ -функция для **x** **не нужна**, так как и слева, и справа приходит одно и то же значение **x**

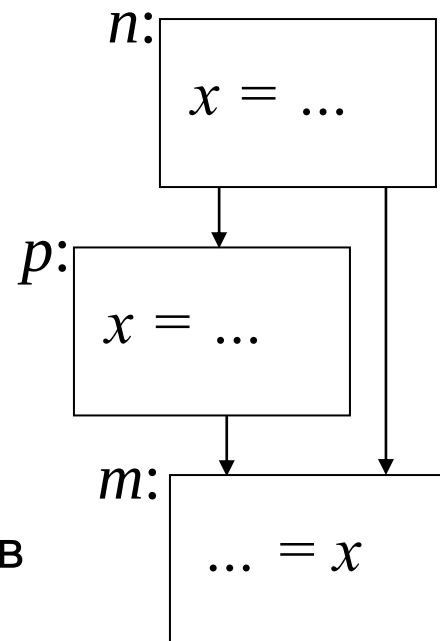


В лиловом блоке **нужна** φ -функция для **x** так как справа появился блок, в котором вычисляется еще одно значение **x**

7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи

- ◇ Пусть $Dom(m)$ – множество доминаторов узла m .
- ◇ Если $n \in Dom(m)$, определение x_0 в вершине n не требует φ -функции в узле m , так как каждый путь, который достигает m проходит через n .
- ◇ Если $n \in Dom(m)$, единственным вариантом, при котором определение x_0 не достигнет узла m , является вклинивание еще одного определения X в некотором узле $p \notin Dom(m)$, расположенном между n и m .
При этом φ -функцию будет требовать не определение X в узле n , а его переопределение в узле p .



7.3 Построение частично усеченной SSA-формы

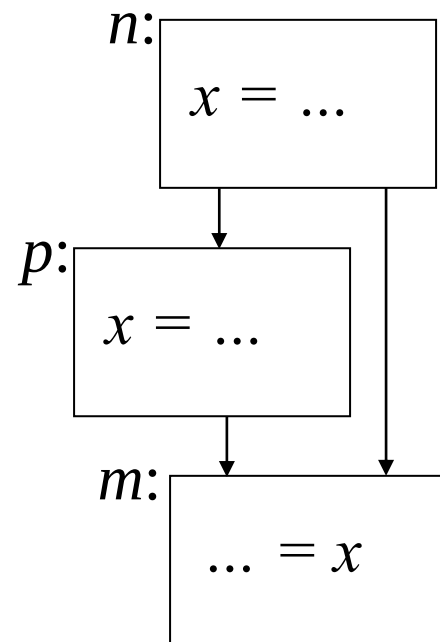
7.3.1. Постановка задачи

◇ Пусть $Dom(m)$ — множество доминаторов узла m .

◇ Нетрудно заметить, что блок с меткой m — граница доминирования блока с меткой p .
В узле m , так как каждый путь, который достигает m проходит через n .

◇ Если $n \in Dom(m)$, единственным вариантом, при котором определение x_0 не достигнет узла m , является вклинивание определения X в некотором узле $p \notin Dom(m)$, расположенном между n и m .

При этом φ -функцию будет требовать не определение X в узле n , а его переопределение в узле p .



7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи

◇ Пусть $Dom(m)$ — множество доминаторов узла m .

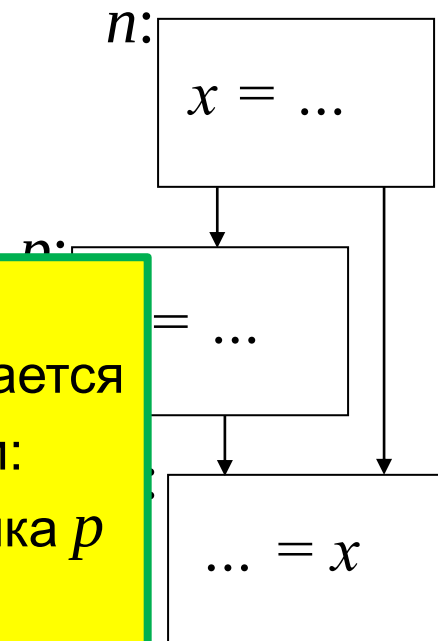
◇ Нетрудно заметить, что блок с меткой m — граница доминирования блока с меткой p .
В узле m , так как каждый путь, который достигает m проходит через n .

Напоминание.

Границей доминирования $DF(n)$ узла n называется множество узлов m , удовлетворяющих условиям:

- (1) n является доминатором предшественника p узла m : $p \in Pred(m) \ \& \ n \in Dom(p)$
- (2) n не является строгим доминатором m
$$n \notin (Dom(m) - \{m\})$$

не определение X в узле n , а его
переопределение в узле p .



7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

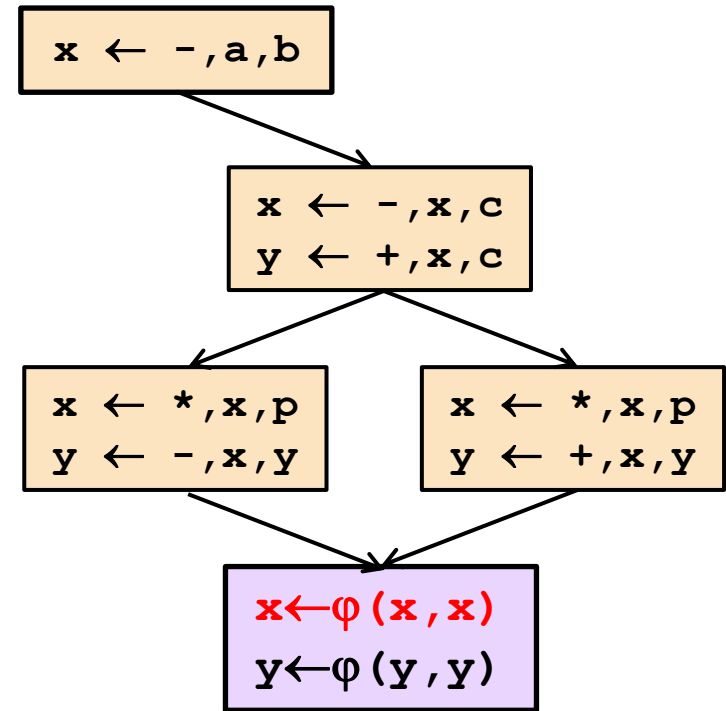
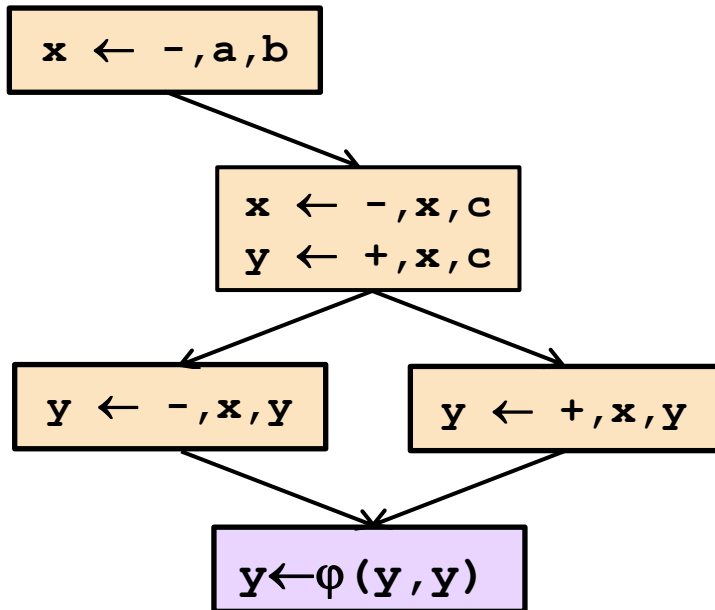
- ◇ Базовый алгоритм помещал по φ -функции для **каждой** переменной в начало **каждой** вершины сбора.
- ◇ Границы доминирования позволяют более точно определить, какие именно φ -функции необходимы в данной вершине.

Правило: Определение переменной x в базовом блоке B требует соответствующей φ -функции в каждой вершине из $DF(B)$.

При этом вставленная φ -функция становится новым определением x , так что могут потребоваться новые φ -функции.

7.3 Построение частично усеченной SSA-формы

7.3.1. Постановка задачи. Пример



Какие φ -функции нужны в лиловом блоке?

7.3 Построение частично усеченной SSA-формы

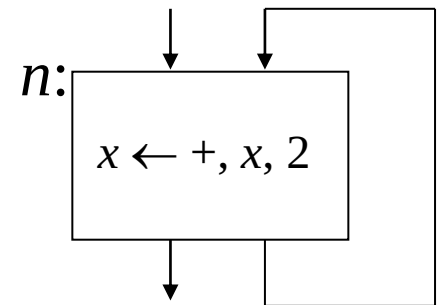
7.3.2. Размещение φ -функций

- ◇ Базовый алгоритм помещал по φ -функции для **каждой** переменной в начало **каждой** вершины сбора.
- ◇ Границы доминирования позволяют более точно определить, какие именно φ -функции необходимы в данной вершине.

Правило: Определение переменной x в базовом блоке B требует соответствующей φ -функции в каждой вершине из $DF(B)$.

При этом вставленная φ -функция становится новым определением x , так что могут потребоваться новые φ -функции.

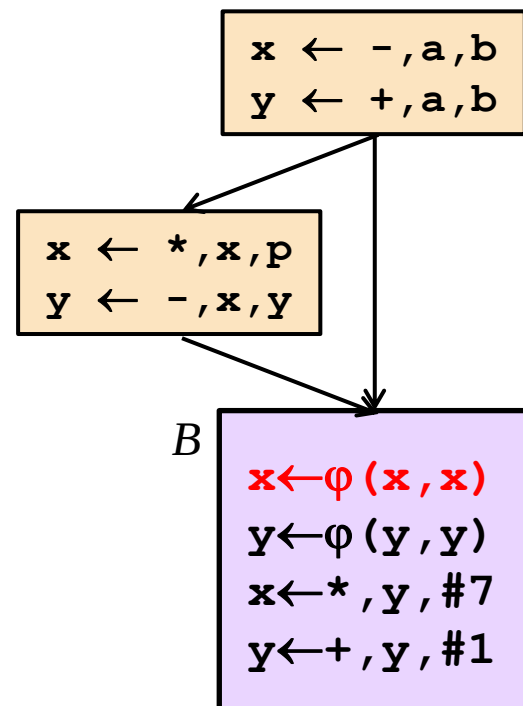
- ◇ **Пример.** $DF(n) = \{n\}$



7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение ϕ -функций

- ◇ ϕ -функция должна быть *живой*, т.е. живой должна быть переменная, которую эта ϕ -функция определяет. (пример на рисунке справа). Отметим, что при анализе живых переменных $x \in def_B$, а $y \in use_B$.
 - ◇ Переменная, которая используется только в одном блоке, не может иметь *живой* ϕ -функции.
- Поэтому ϕ -функции нужно вставлять только для *глобальных имен*, т.е. имен, используемых в нескольких базовых блоках.



7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

- ◇ Перед размещением φ -функций вычисляется множество *Globals* глобальных имен (имен, используемых более чем в одном блоке). При вычислении *Globals* используются результаты анализа живых переменных (множества def_B и use_B).
- ◇ Алгоритм вычисления множества *Globals* попутно вычисляет для каждого базового блока B множество def_B и для каждой переменной $x \in Globals$ – множество $Blocks(x)$ базовых блоков B , в которых определяется x .

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

- ◇ Итак, перед размещением φ -функций вычисляется множество глобальных имен *Globals*. При вычислении *Globals* используются результаты анализа живых переменных.
- ◇ Алгоритм вычисления множества *Globals* попутно вычисляет для каждого базового блока *B* множество def_B и для каждой переменной $x \in Globals$ – множество $Blocks(x)$ базовых блоков *B*, в которых определяется *x*.

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

Алгоритм построения множеств *Globals* и *Blocks(x)*

- ◇ **Вход:** ГПУ
- ◇ **Выход:** множество *Globals*,
множества *Blocks(x)* для каждой переменной *x*
множества def_B для каждого базового блока *B*.
- ◇ **Метод:** Выполнить следующие действия:

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

Алгоритм построения множеств *Globals* и *Blocks(x)*

```
Globals =  $\emptyset$ ;  
for each variable x do  
    Blocks(x) =  $\emptyset$ ;  
    for each block B do {  
        for each instruction i  $\in B$  do {  
            || пусть команда i имеет вид:  $x \leftarrow op, y, z$   
            if  $y \notin \text{def}_B$  then  $\text{Globals} = \text{Globals} \cup \{y\}$ ;  
            if  $z \notin \text{def}_B$  then  $\text{Globals} = \text{Globals} \cup \{z\}$ ;  
            if  $x \notin \text{use}_B$  then  $\text{def}_B = \text{def}_B \cup \{x\}$ ;  
             $\text{Blocks}(x) = \text{Blocks}(x) \cup \{B\}$   
        }  
    }  
}
```

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

Алгоритм построения множеств *Globals* и *Blocks(x)*

Напоминание

def_B – множество переменных, которые в блоке B сначала определяются, а потом используются; это мертвые переменные, так как определение в блоке B убивает их определения в других блоках;

use_B – множество переменных, которые в блоке B сначала используются, а потом определяются; это живые переменные, так как в блоке B используются их определения в других блоках.

```
if  $z \notin def_B$  then  $Globals = Globals \cup \{z\};$ 
```

```
if  $x \notin use_B$  then{
```

```
     $def_B = def_B \cup \{x\};$ 
```

```
     $Blocks(x) = Blocks(x) \cup \{B\}$ 
```

```
}
```

```
}
```

```
}
```

```
}
```

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

Алгоритм построения множеств *Globals* и *Blocks(x)*

Напоминание

def_B – множество переменных, которые в блоке B сначала определяются, а потом используются; это мертвые переменные, так как определение в блоке B убивает их определения в других блоках;

use_B – множество переменных, которые в блоке B сначала используются, а потом определяются; это живые переменные, так как в блоке B используются их определения в других блоках.

Напоминание: вычисление множеств use и def

// пусть блок B содержит k инструкций вида $x \leftarrow op, y, z$

```
for each block  $b$  {  
     $def_B = \emptyset$ ;  $use_B = \emptyset$ ;  
    for  $i = 1$  to  $k$  {  
        if ( $y \notin def_B$ )  $use_B = use_B \cup \{y\}$ ;  
        if ( $z \notin def_B$ )  $use_B = use_B \cup \{z\}$ ;  
        if ( $x \notin use_B$ )  $def_B = def_B \cup \{x\}$ ;  
    }  
}
```

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

Алгоритм размещения φ -функций

- ◇ **Вход:** исходный ГПУ
- ◇ **Выход:** преобразованный ГПУ
- ◇ **Метод:** Выполнить следующие действия

```
for each name  $x \in \text{Globals}$  do {  
     $\text{WorkList} = \text{WorkList} \cup \text{Blocks}(x)$ ;  
    for each block  $B \in \text{WorkList}$  do  
        for each block  $D \in \text{DF}(B)$  do {  
            if ( $D$  не содержит  $\varphi$ -функции для  $x$ )  
                вставить  $\varphi$ -функцию для  $x$  в  $D$ ;  
             $\text{WorkList} = \text{WorkList} \cup \{D\}$ ;  
        };  
    }
```


7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение ϕ -функций

Алгоритм размещения ϕ -функций

- ◇ **Вход:** исходный ГПУ
- ◇ **Выход:** преобразованный ГПУ
- ◇ **Метод:** Выполнить следующие действия

```
for each name  $x \in \text{Globals}$  do {  
     $\text{WorkList} = \text{WorkList} \cup \text{Blocks}(x)$ ;  
    for each block  $B \in \text{WorkList}$  do  
        for each block  $D \in \text{DF}(B)$  do {  
            if ( $D$  не содержит  $\phi$ -функции для  $x$  )  
                вставить  $\phi$ -функцию для  $x$  в  $D$ ;  
             $\text{WorkList} = \text{WorkList} \cup \{D\}$ ;
```

Чтобы учесть новое
определение x ,
реализуемое вставленной
 ϕ -функцией

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

- ◇ **Замечание 1.** Для каждого блока $B \in WorkList$ алгоритм вставляет φ -функцию в начало каждого блока $D \in DF(B)$.

Порядок φ -функций роли не играет.

После вставления φ -функции для x в блок D алгоритм добавляет D в *WorkList*, чтобы в дальнейшем учесть новое определение x в блоке D .

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ -функций

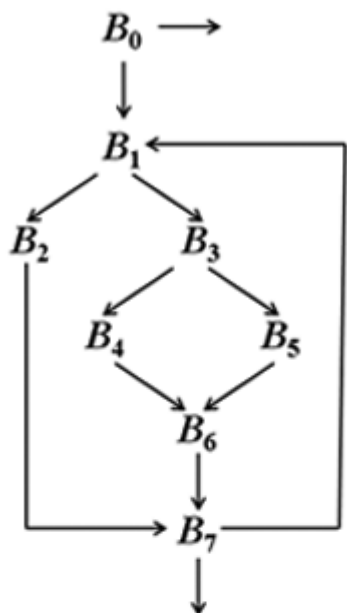
- ◇ **Замечание 2.** Для улучшения эффективности алгоритма необходимо избегать следующих двух видов дублирования:
- (1) помещение какого-либо блока в *WorkList* более одного раза для каждого глобального имени; для этого можно вести список уже обработанных блоков (например, не удалять B из *WorkList*, а пометить его как уже обработанный).
 - (2) рассматриваемый блок может входить в состав границ доминирования нескольких блоков, входящих в *WorkList*; чтобы избежать вставления дублирующих φ -функций для переменной x , можно использовать список блоков, которые уже содержат φ -функции для x , что быстрее, чем проверять каждый раз какие φ -функции включены.

7.3 Построение частично усеченной SSA-формы

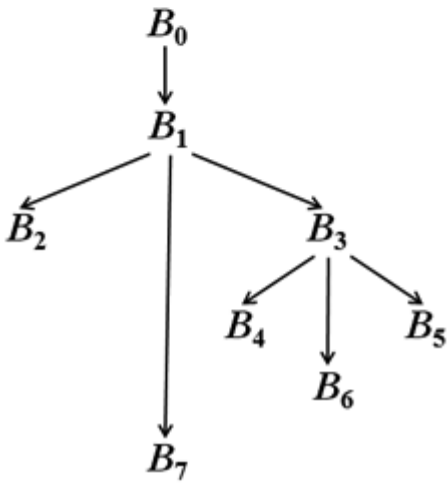
7.3.3. Размещение ϕ -функций. Пример.

B_0	$i \leftarrow 1$	B_5	$c \leftarrow$
B_1	$a \leftarrow$ $b \leftarrow$	B_3	$a \leftarrow$ $d \leftarrow$
B_2	$b \leftarrow$ $c \leftarrow$ $d \leftarrow$	B_7	$y \leftarrow +, a, b$ $z \leftarrow +, c, d$ $i \leftarrow +, i, 1$
B_6	$b \leftarrow$	B_4	$d \leftarrow$

Граф потока управления



Дерево доминаторов



7.3 Построение частично усеченной SSA-формы

7.3.3. Размещение φ -функций. Пример.

- ◇ Сначала вычисляются множества $Globals$ и $Blocks(x)$:
 $Globals = \{a, b, c, d, i\}$.

Множества $Blocks(x)$ представлены в таблице:

a	b	c	d	i
$\{B_1, B_3\}$	$\{B_3, B_6\}$	$\{B_1, B_2, B_5\}$	$\{B_2, B_3, B_4\}$	$\{B_0, B_7\}$

- ◇ Алгоритму размещения φ -функций потребуются также уже вычисленные границы доминирования

n	0	1	2	3	4	5	6	7
$DF(B_n)$	$\emptyset$	$\emptyset$	$\{B_7\}$	$\{B_7\}$	$\{B_6\}$	$\{B_6\}$	$\{B_7\}$	$\{B_1\}$

7.3 Построение частично усеченной SSA-формы

7.3.3. Размещение φ -функций. Пример.



Применяем алгоритм размещения φ -функций.

Берем первую переменную из $Globals = \{a, b, c, d, i\}$, т.е. a

$Blocks(a) = \{B_1, B_3\}$, так что алгоритм должен вставить

φ -функцию для a в каждый блок из границ доминирования

$DF(B_1) = \emptyset$ и $DF(B_3) = \{B_7\}$.

Вставив φ -функцию для a в B_7 , получим новое определение a

– φ -функцию, вставленную в B_7 . Следовательно, необходимо

включить B_7 в $WorkList$ и вставить φ -функцию для a в каждый

блок из $DF(B_7) = \{B_1\}$. Но B_1 уже имеется в $WorkList$, так что,

$DF(B_7)$ не добавляется к $WorkList$.

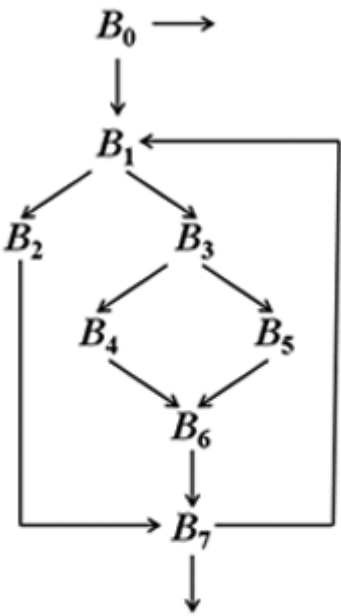
x	a	b	c	d	i
$Blocks(x)$	$\{B_7, B_1\}$	$\{B_7, B_1\}$	$\{B_7, B_1, B_6\}$	$\{B_7, B_1, B_6\}$	$\{B_1\}$

7.3 Построение частично усеченной SSA-формы

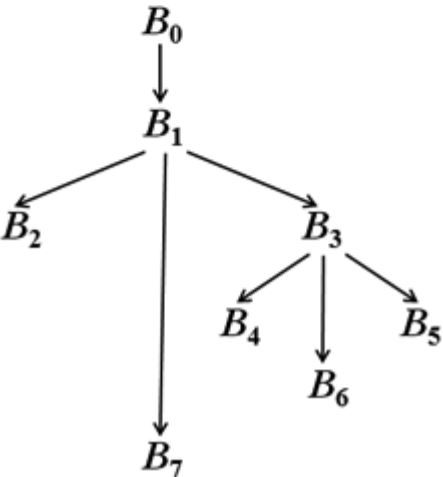
7.3.3. Размещение ϕ -функций. Пример.

B_0	$i \leftarrow 1$		
B_1	$a \leftarrow \phi(a, a)$ $b \leftarrow \phi(b, b)$ $c \leftarrow \phi(c, c)$ $d \leftarrow \phi(d, d)$ $i \leftarrow \phi(i, i)$ $a \leftarrow$ $b \leftarrow$	B_7	$a \leftarrow \phi(a, a)$ $b \leftarrow \phi(b, b)$ $c \leftarrow \phi(c, c)$ $d \leftarrow \phi(d, d)$ $y \leftarrow +, a, b$ $z \leftarrow +, c, d$ $i \leftarrow +, i, 1$
B_2	$b \leftarrow$ $c \leftarrow$ $d \leftarrow$	B_3	$a \leftarrow$ $d \leftarrow$
B_4	$d \leftarrow$	B_6	$c \leftarrow \phi(c, c)$ $d \leftarrow \phi(d, d)$ $b \leftarrow$
B_5	$c \leftarrow$		

Граф потока управления



Дерево доминаторов



7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Алгоритм

- ◇ **Вход:** программа с размещенными φ -функциями
- ◇ **Выход:** программа, в которой каждой переменной сопоставлено ее *SSA-имя*.
- ◇ **Метод:** Сначала (в основном алгоритме) инициализируются стеки и счетчики, после чего из корня дерева доминаторов n_0 вызывается рекурсивная функция *Rename*.
Rename обрабатывает блок, рекурсивно вызывая его последователей по дереву доминаторов.
Закончив обрабатывать очередной блок, *Rename* выталкивает из стеков все имена, помещенные в них во время обработки блока.
Функция *NewName*, манипулируя со счетчиками и стеками, в случае необходимости создает новые имена.

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Алгоритм

◇ Основной алгоритм:

```
for each  $i \in \text{Globals}$  do{  
    counter[i] = 0;  
    stack[i] =  $\emptyset$ ;  
};  
Rename ( $n_0$ ) ;
```

◇ Функция **newName** :

```
NewName (n) {  
    i = counter[n] ;  
    counter[n] += 1 ;  
    Push  $n_i$  onto stack[n] ;  
    return  $n_i$   
}
```

7.3 Построение частично усеченной SSA-формы

7.3.7. Переименование переменных.

Функция $Rename(B)$:

```
for each  $\varphi$ -function  $\in B$ :  $x = \varphi(\dots)$  do
    rename  $x$  as  $NewName(x)$  ;
for each instruction  $\in B$ :  $x \leftarrow op, y, z$  do{
    rewrite  $y$  as  $top(stack[y])$  ;
    rewrite  $z$  as  $top(stack[z])$  ;
    rewrite  $x$  as  $NewName(x)$  ;
for each successor of  $B$  in the flowgraph do
    fill in  $\varphi$ -function parameters;
for each successor  $S$  of  $B$  in the
    dominator tree do  $Rename(S)$ 
for each  $\varphi$ -function  $\in B$ :  $x = \varphi(\dots)$  do
     $Pop(stack[x])$  ;
for each instruction  $\in B$ :  $x \leftarrow op, y, z$  do
     $Pop(stack[x])$  ;
```

7.3 Построение частично усеченной SSA-

7.3.7. Переименование переменных.

Функция $Rename(B)$:

```
for each  $\phi$ -function  $\in B$ :  $x = \phi$   
    rename  $x$  as  $newName(x)$  ;  
for each instruction  $\in B$ :  $x \leftarrow op, y, z$  do {  
    rewrite  $y$  as  $top(stack[y])$  ;  
    rewrite  $z$  as  $top(stack[z])$  ;  
    rewrite  $x$  as  $newName(x)$  ;  
for each successor of  $B$  in the flowgraph do  
    fill in  $\phi$ -function parameters ;  
for each successor  $S$  of  $B$  in the  
    dominator tree do  $Rename(S)$   
for each  $\phi$ -function  $\in B$ :  $x = \phi(\dots)$  do  
     $Pop(stack[x])$  ;  
for each instruction  $\in B$ :  $x \leftarrow op, y, z$  do  
     $Pop(stack[x])$  ;
```

```
 $newName(n)$  {  
     $i = counter[n]$  ;  
     $counter[n] += 1$  ;  
    Push  $n_i$  onto  
     $stack[n]$  ;  
    return  $n_i$  }
```

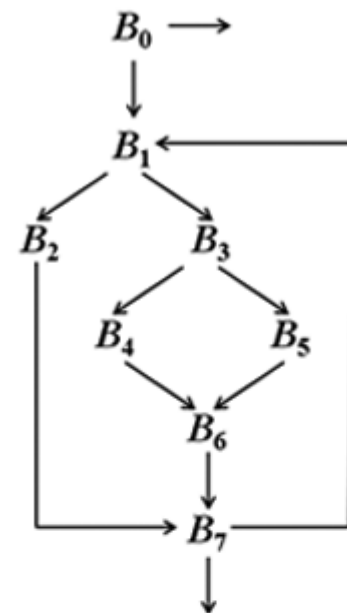
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

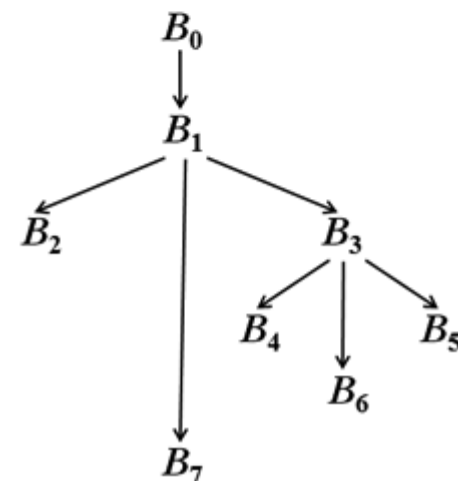
Применим алгоритм переименования к рассматриваемому примеру в предположении, что на входе в блок B_0 определены имена a_0, b_0, c_0, d_0 .

B_0	$i \leftarrow 1$		
B_1	$a \leftarrow \phi(a, a)$ $b \leftarrow \phi(b, b)$ $c \leftarrow \phi(c, c)$ $d \leftarrow \phi(d, d)$ $i \leftarrow \phi(i, i)$ $a \leftarrow$ $b \leftarrow$	B_7	$a \leftarrow \phi(a, a)$ $b \leftarrow \phi(b, b)$ $c \leftarrow \phi(c, c)$ $d \leftarrow \phi(d, d)$ $y \leftarrow +, a, b$ $z \leftarrow +, c, d$ $i \leftarrow +, i, 1$
B_2	$b \leftarrow$ $c \leftarrow$ $d \leftarrow$	B_3	$a \leftarrow$ $d \leftarrow$
B_4	$d \leftarrow$	B_6	$c \leftarrow \phi(c, c)$ $d \leftarrow \phi(d, d)$ $b \leftarrow$
B_5	$c \leftarrow$		

Граф потока управления



Дерево доминаторов



7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

◇ Корнем обрабатываемой части дерева доминаторов является блок B_0 . Поэтому «*Основной алгоритм*», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $Rename(B_0)$.

◇ Порядок обработки базовых блоков определяется деревом доминаторов

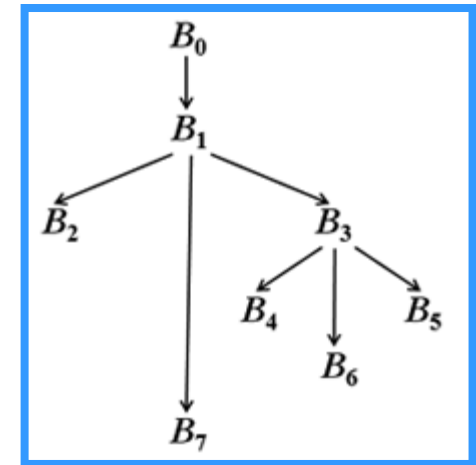
◇ первым обрабатывается блок B_0 .

$B_0 \quad i \leftarrow 1$

◇ во время обработки B_0 будет вызван $Rename(B_1)$,

◇ во время обработки B_1 будут вызваны $Rename(B_2)$, $Rename(B_7)$ и $Rename(B_3)$,

◇ во время обработки B_3 будет вызван $Rename(B_4)$, $Rename(B_6)$, $Rename(B_5)$



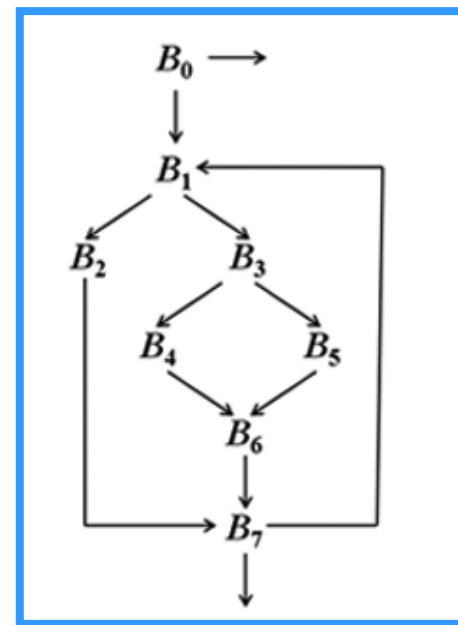
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

- ◇ Корнем обрабатываемой части дерева доминаторов является блок B_0 . Поэтому «**Основной алгоритм**», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов **Rename**(B_0).

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	

B_0 : $i \leftarrow 1$;



- ◇ **Rename**(B_0):

- ◇ Вызов **NewName** (i) $\rightarrow$ возвращает имя i_0
 - Замена $i \leftarrow 1$; на $i_0 \leftarrow 1$;
 - Занесение i_0 в стек для i
 - Увеличение счетчика для i на 1
- ◇ Заполнение параметров φ -функций в B_1
- ◇ Вызов **Rename**(B_1)

B_0 : $i_0 \leftarrow 1$;

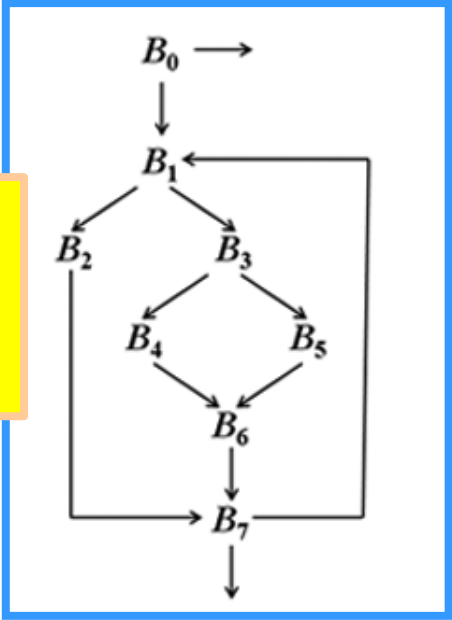
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

- Корнем обрабатываемой части дерева доминаторов является блок B_0 . Поэтому «**Основной алгоритм**», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $Rename(B_0)$.

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	

$B_0:$ $i \leftarrow 1;$



$B_0:$ $i_0 \leftarrow 1;$

Счетчики для переменных a, b, c, d имеют значение 1, а не 0, так как считается, что этим переменным были присвоены значения до входа в блок B_0

занесение i_0 в стек для i

Увеличение счетчика для i на 1

- Заполнение параметров φ -функций в B_1
- Вызов $Rename(B_1)$

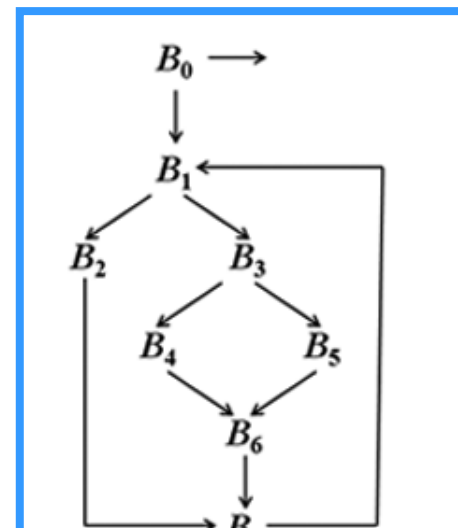
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

- ◇ Корнем обрабатываемой части дерева доминаторов является блок B_0 . Поэтому «**Основной алгоритм**», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов **Rename**(B_0).

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	

B_0 : $i \leftarrow 1$;



- ◇ **Rename**(B_0):

- ◇ Вызов **NewName** (i) $\rightarrow$ возвращает имя i_0
 - Замена $i \leftarrow 1$; на $i_0 \leftarrow 1$;
 - Занесение i_0 в стек для i
 - Увеличение счетчика для i на 1
- ◇ Заполнение параметров φ -функций в B_1
- ◇ Вызов **Rename**(B_1)

$B_1 = \text{Succ}(B_0)$ по ГПУ

B_0 : $i_0 \leftarrow 1$;

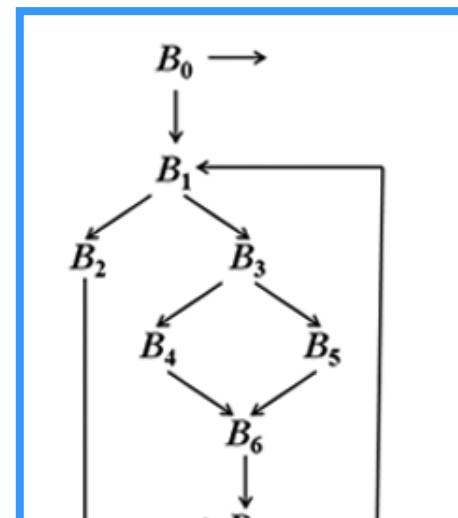
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

- ◇ Корнем обрабатываемой части дерева доминаторов является блок B_0 . Поэтому «**Основной алгоритм**», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов **$Rename(B_0)$** .

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	

$B_0: \quad i \leftarrow 1;$



- ◇ **$Rename(B_0)$:**

- ◇ Вызов **$NewName(i)$** $\rightarrow$ возвращает имя i_0

Замена $i \leftarrow 1;$ на $i_0 \leftarrow 1;$

Занесение i_0 в стек для i

Увеличение счетчика для i на 1

- ◇ Заполнение параметров φ -функций в B_1 **$B_1 = Succ(B_0)$ по ГПУ**

- ◇ Вызов **$Rename(B_1)$** **$B_1 = Succ(B_0)$ по дереву доминаторов**

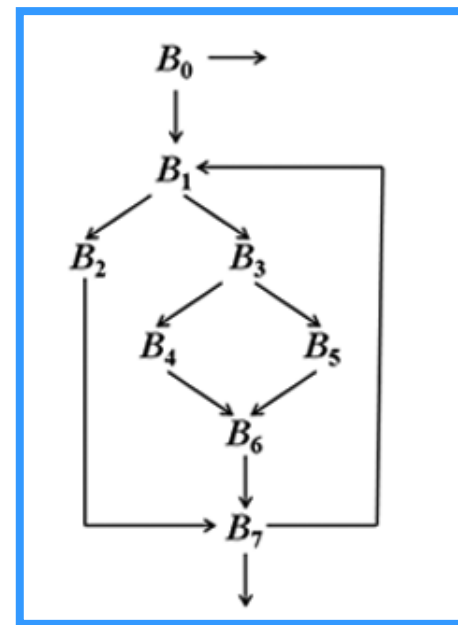
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

- ◇ Корнем обрабатываемой части дерева доминаторов является блок B_0 . Поэтому «**Основной алгоритм**», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов **$Rename(B_0)$** .

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	

$B_0:$ $i \leftarrow 1;$



- ◇ **$Rename(B_0)$:**

- ◇ Вызов **$NewName(i)$** $\rightarrow$ возвращает имя i_0
 - Замена $i \leftarrow 1;$ на $i_0 \leftarrow 1;$
 - Занесение i_0 в стек для i
 - Увеличение счетчика для i на 1
- ◇ Заполнение параметров φ -функций в B_1
- ◇ Вызов **$Rename(B_1)$**

$B_0:$ $i_0 \leftarrow 1;$

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_1

	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0

B_1 :

```
a ← φ(a0, a) ;  
b ← φ(b0, b) ;  
c ← φ(c0, c) ;  
d ← φ(d0, d) ;  
i ← φ(i0, i) ;  
a ← ... ;  
c ← ... ;  
(a < c) ;
```

Работа $Rename(B_1)$:

- Вызов $NewName(a) \rightarrow a_1$
- Вызов $NewName(b) \rightarrow b_1$
- Вызов $NewName(c) \rightarrow c_1$
- Вызов $NewName(d) \rightarrow d_1$
- Вызов $NewName(i) \rightarrow i_1$
- Вызов $NewName(a) \rightarrow a_2$
- Вызов $NewName(c) \rightarrow c_2$
- Заполнение параметров ϕ -функций в B_2 отсутствует, так как в B_2 нет ϕ -функций
- Вызов $Rename(B_2)$

$\downarrow Rename(B_1)$

B_1 :

```
a1 ← φ(a0, a) ;  
b1 ← φ(b0, b) ;  
c1 ← φ c0, c) ;  
d1 ← φ(d0, d) ;  
i1 ← φ(i0, i) ;  
a2 ← ... ;  
c2 ← ... ;  
(a2 < c2) ;
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_1	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0

B_1 :

```
a ← φ(a0, a) ;  
b ← φ(b0, b) ;  
c ← φ(c0, c) ;  
d ← φ(d0, d) ;  
i ← φ(i0, i) ;
```

Номера левых параметров ϕ -функций соответствуют номерам переменных, определяемым в блоке B_0

- Вызов *NewName* (**b**) → **b₁**
- Вызов *NewName* (**c**) → **c₁**
- Вызов *NewName* (**d**) → **d₁**
- Вызов *NewName* (**i**) → **i₁**
- Вызов *NewName* (**a**) → **a₀**

Номера присвоены функцией *NewName*

- Заполнение параметров ϕ -функций в B_2 отсутствует, так как в B_2 нет ϕ -функций
- Вызов *Rename*(B_2)

$\downarrow$ *Rename*(B_1)

B_1 :

```
a1 ← φ(a0, a) ;  
b1 ← φ(b0, b) ;  
c1 ← φ(c0, c) ;  
d1 ← φ(d0, d) ;  
i1 ← φ(i0, i) ;  
a2 ← ... ;  
c2 ← ... ;  
(a2 < c2) ;
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_1

	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0

```
 $B_1$ :  
   $a \leftarrow \phi(a_0, a);$   
   $b \leftarrow \phi(b_0, b);$   
   $c \leftarrow \phi(c_0, c);$   
   $d \leftarrow \phi(d_0, d);$   
   $i \leftarrow \phi(i_0, i);$   
   $a \leftarrow \dots;$   
   $c \leftarrow \dots;$ 
```

Работа $Rename(B_1)$:

- Вызов $NewName(a) \rightarrow a_1$
- Вызов $NewName(b) \rightarrow b_1$
- Вызов $NewName(c) \rightarrow c_1$
- Вызов $NewName(d) \rightarrow d_1$
- Вызов $NewName(i) \rightarrow i_1$
- Вызов $NewName(a) \rightarrow a_2$

Номера присвоены функцией $NewName$

- Заполнение параметров ϕ -функций в B_2 отсутствует, так как в B_2 нет ϕ -функций
- Вызов $Rename(B_2)$

```
NewName(n) {  
   $i = counter[n];$   
   $counter[n] += 1;$   
  Push  $n_i$  onto  $stack[n];$   
  return  $n_i$ ;  
}
```

$me(B_1)$

```
 $B_1$ :  
   $a_1 \leftarrow \phi(a_0, a);$   
   $b_1 \leftarrow \phi(b_0, b);$   
   $c_1 \leftarrow \phi(c_0, c);$   
   $d_1 \leftarrow \phi(d_0, d);$   
   $i_1 \leftarrow \phi(i_0, i);$   
   $a_2 \leftarrow \dots;$   
   $c_2 \leftarrow \dots;$   
   $(a_2 < c_2);$ 
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_2

	a	b	c	d	i
Счетчики	3	2	3	2	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

B_2 до чистки

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	
			c_3		

```

NewName (n) {
  i = counter[n];
  counter[n] += 1;
  Push  $n_i$  onto stack[n];
  return  $n_i$ }
    
```

B_2 :

```

b ← ...;
c ← ...;
d ← ...;
    
```

$\downarrow$ *Rename*(B_2)

B_2 :

```

b2 ← ...;
c3 ← ...;
d2 ← ...;
    
```

Работа *Rename*(B_2):

- ◇ Вызов *NewName* (**b**) → b_2
- ◇ Вызов *NewName* (**c**) → c_3
- ◇ Вызов *NewName* (**d**) → d_2
- ◇ Заполнение параметров
 ϕ -функций в B_7
- ◇ Чистка стеков
- ◇ Возврат в *Rename*(B_1)

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_2

	a	b	c	d	i
Счетчики	3	2	3	2	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	

B_2 до чистки

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	
			c_3		

B_2 :
 $b \leftarrow \dots;$
 $c \leftarrow \dots;$
 $d \leftarrow \dots;$

$\downarrow$ $Rename(B_2)$

B_2 :
 $b_2 \leftarrow \dots;$
 $c_3 \leftarrow \dots;$
 $d_2 \leftarrow \dots;$

Работа $Rename(B_2)$:

- ◇ Вызов $NewName(b) \rightarrow b_2$
- ◇ Вызов $NewName(c) \rightarrow c_3$
- ◇ Вызов $NewName(d) \rightarrow d_2$
- ◇ Заполнение параметров
 ϕ -функций в B_7
- ◇ Чистка стеков
- ◇ Возврат в $Rename(B_1)$

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

B_2 до чистки

Счетчики

Стеки ($\downarrow$)

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	

Работа $Rename(B_2)$:

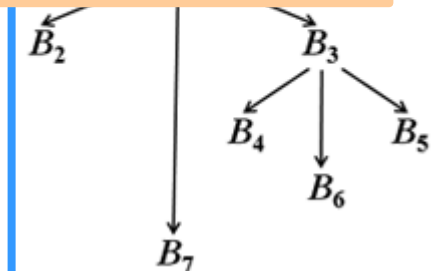
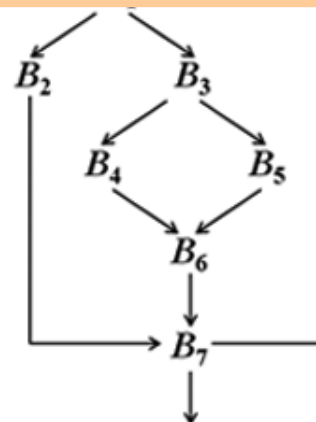
- ✦ Вызов $NewName(b) \rightarrow b_2$
- ✦ Вызов $NewName(c) \rightarrow c_3$
- ✦ Вызов $NewName(d) \rightarrow d_2$
- ✦ Заполнение параметров ϕ -функций в B_7
- ✦ **Чистка стеков**
- ✦ Возврат в $Rename(B_1)$

Как только имя переменной, определенное в B_i было использовано во всех блоках из $Succ(B_i)$, оно выбрасывается из стека, так как больше не понадобится (**Чистка стеков**)

Счетчики

Стеки ($\downarrow$)

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
			c_2		

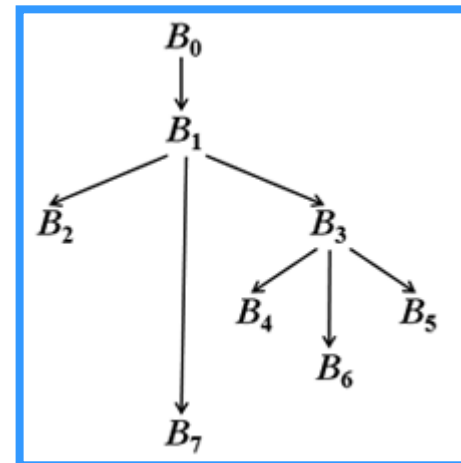
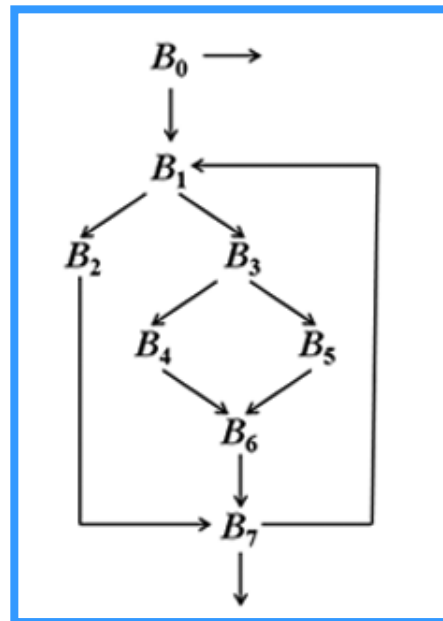


7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

После B_2

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
			c_2		



Работа $\text{Rename}(B_1)$:

- Заполнение параметров ϕ -функций B_7 проведено в B_2
- Вызов $\text{Rename}(B_7)$

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_7

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
			c_2		

B_7 :

```
a ← φ(a2, a);  
b ← φ(b2, b);  
c ← φ(c3, c);  
d ← φ(d2, d);  
x ← a + b;  
z ← c + d;  
i ← i + 1;  
(i ≤ 100) ...;
```

Работа $Rename(B_7)$:

- Вызов $NewName(a) \rightarrow a_3$;
- Вызов $NewName(b) \rightarrow b_3$;
- Вызов $NewName(c) \rightarrow c_4$;
- Вызов $NewName(d) \rightarrow d_3$;
- Вызов $NewName(i) \rightarrow i_2$;
- Переименование операндов;
- x и z не входят в $Globals$ и не переименовываются

$\downarrow$ $Rename(B_7)$

B_7 :

```
a3 ← φ(a2, a);  
b3 ← φ(b2, b);  
c4 ← φ(c3, c);  
d3 ← φ(d2, d);  
x ← a3 + b3;  
z ← c4 + d3;  
i2 ← i1 + 1;  
(i2 ≤ 100) ...;
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

B_7 до чистки

Счетчики

Стеки ($\downarrow$)

a	b	c	d	i
4	4	5	4	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_3	c_2	d_3	i_2
		c_4		

B_7 :

```
 $a_3 \leftarrow \phi(a_2, a);$   
 $b_3 \leftarrow \phi(b_2, b);$   
 $c_4 \leftarrow \phi(c_3, c);$   
 $d_3 \leftarrow \phi(d_2, d);$   
 $x \leftarrow a_3 + b_3;$   
 $z \leftarrow c_4 + d_3;$   
 $i_2 \leftarrow i_1 + 1;$   
 $(i_2 \leq 100) \dots;$ 
```

◇ Работа ***Rename***(B_7):

◇ Заполнение параметров

ϕ -функций в B_1



◇ **Чистка стеков**

◇ Возврат в ***Rename***(B_1);

Значения a_3 , b_3 , c_4 , d_3 и i_2 уже определены и (по определению SSA) больше определяться не будут, поэтому они удаляются из стеков

B_1 :

```
 $a_1 \leftarrow \phi(a_0, a_3);$   
 $b_1 \leftarrow \phi(b_0, b_3);$   
 $c_1 \leftarrow \phi(c_0, c_4);$   
 $d_1 \leftarrow \phi(d_0, d_3);$   
 $i_1 \leftarrow \phi(i_0, i_2);$   
 $a_2 \leftarrow \dots;$   
 $c_2 \leftarrow \dots;$   
 $(a_2 < c_2);$ 
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

B_7 до чистки

Счетчики

Стеки ($\downarrow$)

a	b	c	d	i
4	4	5	4	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_3	c_2	d_3	i_2
		c_4		

B_7 :

```

 $a_3 \leftarrow \phi(a_2, a);$ 
 $b_3 \leftarrow \phi(b_2, b);$ 
 $c_4 \leftarrow \phi(c_3, c);$ 
 $d_3 \leftarrow \phi(d_2, d);$ 
 $x \leftarrow a_3 + b_3;$ 
 $z \leftarrow c_4 + d_3;$ 
 $i_2 \leftarrow i_1 + 1;$ 
 $(i_2 \leq 100) \dots;$ 
    
```

◇ Работа ***Rename***(B_7):

◇ Заполнение параметров

ϕ -функций в B_1 **$B_1 \in Succ(B_7)$**

◇ **Чистка стеков**

◇ Возврат в ***Rename***(B_1);

Значения a_3 , b_3 , c_4 , d_3 и i_2 уже определены и (по определению SSA) больше определяться не будут, поэтому они удаляются из стеков

B_1 :

```

 $a_1 \leftarrow \phi(a_0, a_3);$ 
 $b_1 \leftarrow \phi(b_0, b_3);$ 
 $c_1 \leftarrow \phi(c_0, c_4);$ 
 $d_1 \leftarrow \phi(d_0, d_3);$ 
 $i_1 \leftarrow \phi(i_0, i_2);$ 
 $a_2 \leftarrow \dots;$ 
 $c_2 \leftarrow \dots;$ 
 $(a_2 < c_2);$ 
    
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

**B_7 после
ЧИСТКИ**

Счетчики

Стеки ($\downarrow$)

	a	b	c	d	i
Счетчики	4	4	5	4	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

B_7 :

```
 $a_3 \leftarrow \phi(a_2, a);$   
 $b_3 \leftarrow \phi(b_2, b);$   
 $c_4 \leftarrow \phi(c_3, c);$   
 $d_3 \leftarrow \phi(d_2, d);$   
 $x \leftarrow a_3 + b_3;$   
 $z \leftarrow c_4 + d_3;$   
 $i_2 \leftarrow i_1 + 1;$   
 $(i_2 \leq 100) \dots;$ 
```

- ◇ Работа $\text{Rename}(B_7)$:
 - ◇ Возврат в $\text{Rename}(B_1)$;
- ◇ Работа $\text{Rename}(B_1)$:
 - ◇ Заполнение параметров ϕ -функций в B_3 не производится, так как в B_3 нет ϕ -функций
 - ◇ Вызов $\text{Rename}(B_3)$;

B_1 :

```
 $a_1 \leftarrow \phi(a_0, a_3);$   
 $b_1 \leftarrow \phi(b_0, b_3);$   
 $c_1 \leftarrow \phi(c_0, c_4);$   
 $d_1 \leftarrow \phi(d_0, d_3);$   
 $i_1 \leftarrow \phi(i_0, i_2);$   
 $a_2 \leftarrow \dots;$   
 $c_2 \leftarrow \dots;$   
 $(a_2 < c_2);$ 
```

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_3

Счетчики

Стеки ($\downarrow$)

a	b	c	d	i
4	4	5	4	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2		

B_3 :
 $a \leftarrow \dots;$
 $d \leftarrow \dots;$
 $(a \leq d) \dots;$

$\downarrow$ $Rename(B_3)$

Вход в B_4

Счетчики

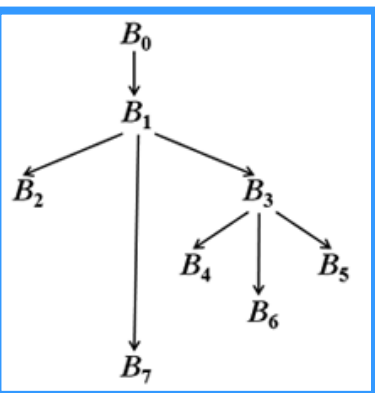
Стеки ($\downarrow$)

a	b	c	d	i
4	4	5	4	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_4	
a_4				

B_3 :
 $a_4 \leftarrow \dots;$
 $d_4 \leftarrow \dots;$
 $(a_4 \leq d_4) \dots;$

Работа $Rename(B_3)$:

- ◇ Вызов $NewName(a) \rightarrow a_4$;
- ◇ Вызов $NewName(d) \rightarrow d_4$;
- ◇ Переименование операндов;
- ◇ **Чистка стеков**
- ◇ Вызов $Rename(B_4)$;



7.3 Построение частично усеченной SSA-формы

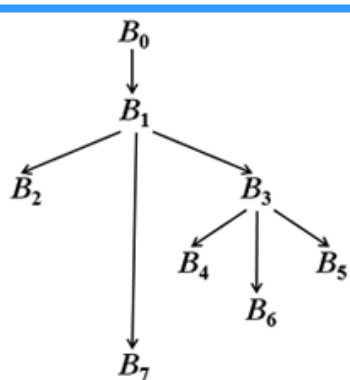
7.3.4. Переименование переменных

Вход в B_4

	a	b	c	d	i
Счетчики	5	4	5	5	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	

B_4 до чистки

	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	



$B_4:$ $d \leftarrow \dots;$

$\downarrow$ $Rename(B_4)$

$B_4:$ $d_5 \leftarrow \dots;$

Работа $Rename(B_4)$:

- ◇ Вызов $NewName(d) \rightarrow d_5$;
- ◇ Переименование операндов;
- ◇ Заполнение параметров ϕ -функций в B_6
- ◇ **Чистка стеков**
- ◇ Вызов $Rename(B_6)$;

$B_6:$

$$c_5 \leftarrow \phi(c_2, c);$$

$$d_6 \leftarrow \phi(d_4, d_5);$$

$$b_4 \leftarrow \dots;$$

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_6

Счетчики

Стеки ($\downarrow$)

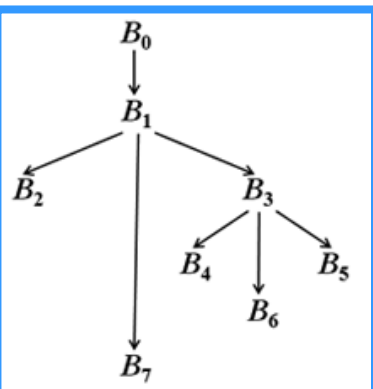
	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

B_6 до чистки

Счетчики

Стеки ($\downarrow$)

	a	b	c	d	i
Счетчики	5	5	6	7	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_4	c_2	d_4	
	a_4		c_5	d_6	



B_6 :

```

c ← φ(c, c);
d ← φ(d, d);
b ← ...;
  
```

$\downarrow$ *Rename*(B_6)

B_6 :

```

c5 ← φ(c2, c);
d6 ← φ(d4, d5);
b4 ← ...;
  
```

Работа *Rename*(B_6):

- ◇ Вызов *NewName*(c) → c_5 ;
- ◇ Вызов *NewName*(d) → d_6 ;
- ◇ Вызов *NewName*(b) → b_4 ;
- ◇ Переименование операндов;
- ◇ Заполнение параметров ϕ -функций в B_7
- ◇ **Чистка стеков**
- ◇ Возврат в *Rename*(B_3);

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Вход в B_5

Счетчики

Стеки ($\downarrow$)

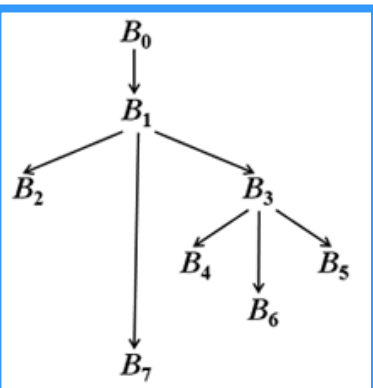
	a	b	c	d	i
Счетчики	5	4	5	5	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

B_5 ДО ЧИСТКИ

Счетчики

Стеки ($\downarrow$)

	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_5	
	a_4		c_5		



$B_5:$ $c \leftarrow \dots;$

$\downarrow$ $Rename(B_5)$

$B_5:$ $c_5 \leftarrow \dots;$

Работа $Rename(B_5)$:

- ♦ Вызов $NewName(d) \rightarrow d_5$;
- ♦ Переименование операндов;
- ♦ Заполнение параметров φ -функций в B_6
- ♦ **Чистка стеков**
- ♦ Возврат в $Rename(B_3)$;

$B_6:$

$$c_6 \leftarrow \varphi(c_2, c_5);$$

$$d_6 \leftarrow \varphi(d_4, d_5);$$

$$b_4 \leftarrow \dots;$$

7.3 Построение частично усеченной SSA-формы

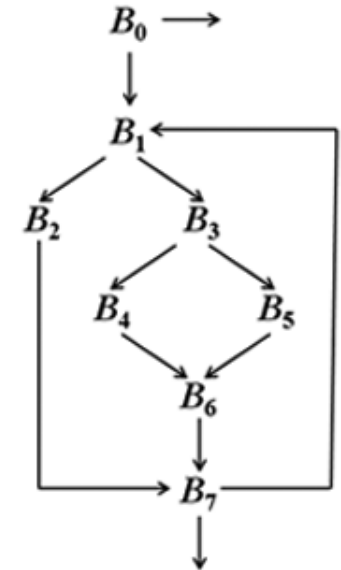
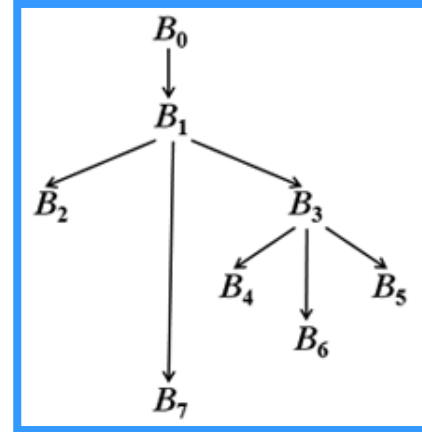
7.3.4. Переименование переменных

Вход в B_3

Счетчики

Стеки ($\downarrow$)

	a	b	c	d	i
Счетчики	5	5	6	7	3
Стеки ($\downarrow$)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_4	c_2	d_4	
	a_4		c_5	d_6	



Работа $Rename(B_3)$:

◆ **Чистка стеков**

◆ Возврат в $Rename(B_1)$;

Работа $Rename(B_1)$:

◆ Возврат в $Rename(B_0)$;

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

Блок B_6 (последний)

Rename(B_6) по рекурсии из блока B_1 .

- 1) заменяет имена, определяемые φ -функциями, создавая новые имена a_4 , b_4 , c_6 и d_6 .
- 2) заменяет использования глобальных имен в двух присваиваниях, не меняя определяемых ими имен, так как ни y , ни z не являются глобальными.
- 3) последнем присваивании заменяет используемое имя на i_1 , а затем создает новое имя i_2 для определения переменной i .
- 4) заменяет второй параметр каждой φ -функции в B_1 (единственный последователь по ГПУ) на соответствующее текущее имя (a_4 , b_4 , c_6 , d_6 и i_2).

Потом *Rename* освобождает стеки, возвращается в B_1 , в B_0 и прекращает работу.

7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

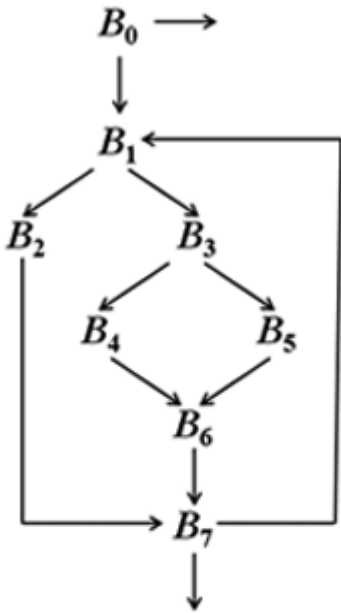
Блок B_0

Когда в конце обхода дерева доминаторов управление снова попадает в B_0 *Rename* выталкивает значение из стека для i и происходит выход из *Rename*.

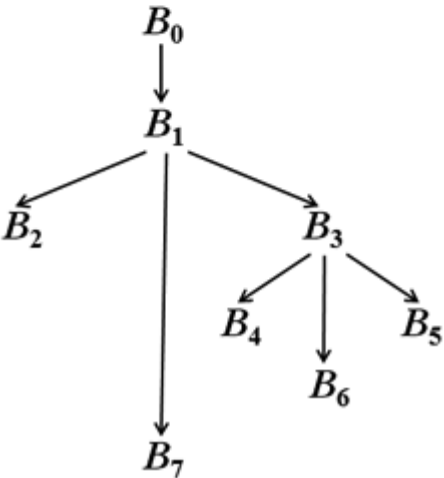
Состояние счетчиков и стеков в момент, когда *Rename*(B_0) уже кончила обрабатывать блок B_0 , но еще не удалила имена, связанные с B_0 из соответствующих стеков

Глобальные переменные	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки	a_0	b_0	c_0	d_0	i_0

Граф потока управления



Дерево доминаторов



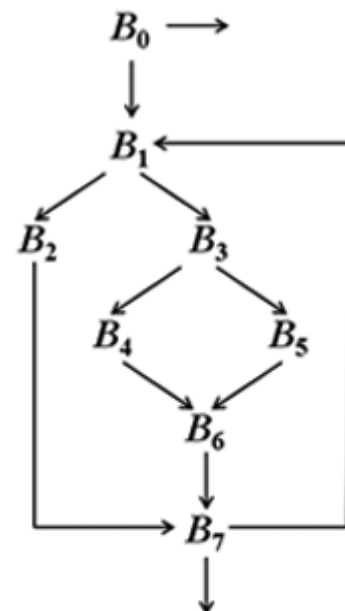
7.3 Построение частично усеченной SSA-формы

7.3.4. Переименование переменных

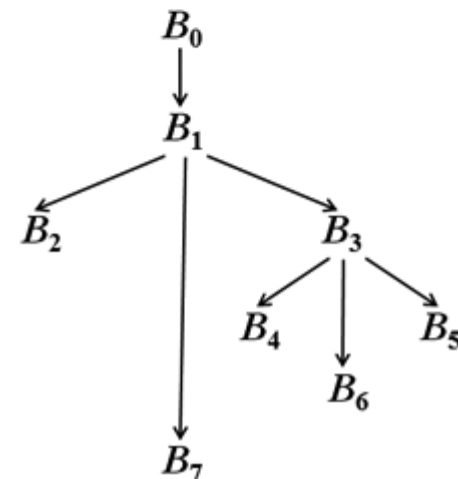
После окончания работы алгоритма получим

B_0	$i_0 \leftarrow 1$		
B_1	$a_1 \leftarrow \phi(a_0, a_4)$ $b_1 \leftarrow \phi(b_0, b_4)$ $c_1 \leftarrow \phi(c_0, c_6)$ $d_1 \leftarrow \phi(d_0, d_6)$ $i_1 \leftarrow \phi(i_0, i_2)$ $a_2 \leftarrow$ $b_2 \leftarrow$	B_7	$a_4 \leftarrow \phi(a_2, a_3)$ $b_4 \leftarrow \phi(b_2, b_3)$ $c_6 \leftarrow \phi(c_3, c_5)$ $d_6 \leftarrow \phi(d_2, d_5)$ $y \leftarrow +, a_4, b_4$ $z \leftarrow +, c_6, d_6$ $i_2 \leftarrow +, i_1, 1$
B_2	$b_2 \leftarrow$ $c_3 \leftarrow$ $d_2 \leftarrow$	B_3	$a_3 \leftarrow$ $d_3 \leftarrow$
B_4	$d_4 \leftarrow$	B_6	$c_5 \leftarrow \phi(c_2, c_4)$ $d_5 \leftarrow \phi(d_4, d_3)$ $b_3 \leftarrow$
B_5	$c_4 \leftarrow$		

Граф потока управления



Дерево доминаторов



7.4 Восстановление кода из SSA-формы

7.4.1. Постановка задачи

- ◇ Поскольку современные процессоры не выполняют ϕ -функций, компилятор должен перевести программу в SSA-форме в выполняемый код.
- ◇ Рассмотрение примеров наводит на мысль, что **компилятору достаточно попросту опустить индексы имен и удалить ϕ -функции.**
Такой подход был бы правильным, если бы при построении SSA-формы использовался простейший алгоритм.
- ◇ Однако, если переименования выполняются рассмотренным алгоритмом, простой процесс переименования **может привести к некорректному коду.**

7.4 Восстановление кода из SSA-формы

7.4.1. Постановка задачи

◇ **Пример.** Внизу слева показан базовый блок из четырех команд и его оптимизация с помощью ЛНЗ над исходными именами.

Справа показан тот же самый пример с использованием SSA-имен.

Вследствие того, что исходное имя a имеет разные SSA-имена a_0 и a_1 , удалось оптимизировать последнее присваивание.

Заметим однако, что если у имен опустить индексы, получится некорректный код: c будет присвоено значение 17.

Исходные имена		SSA-имена	
$a \leftarrow +, x, y$	$a \leftarrow +, x, y$	$a_0 \leftarrow +, x_0, y_0$	$a_0 \leftarrow +, x_0, y_0$
$b \leftarrow +, x, y$	$b \leftarrow a$	$b_0 \leftarrow +, x_0, y_0$	$b_0 \leftarrow a_0$
$a \leftarrow 17$	$a \leftarrow 17$	$a_0 \leftarrow 17$	$a_1 \leftarrow 17$
$c \leftarrow +, x, y$	$c \leftarrow +, x, y$	$c_0 \leftarrow +, x_0, y_0$	$c_0 \leftarrow a_0$

7.4 Восстановление кода из SSA-формы

7.4.2. Замена ϕ -функций группами инструкций копирования

- ◇ Можно оставить SSA-имена неизменными, заменив каждую ϕ -функцию группой команд копирования (по одной для каждого входного ребра), предоставив разбираться с именами оптимизирующему преобразованию «Распространение копий». В рассматриваемом примере при удалении ϕ -функций будет:

```
B7  a4 ←  $\phi(a_2, a_3)$ 
      b4 ←  $\phi(b_2, b_3)$ 
      c6 ←  $\phi(c_3, c_5)$ 
      d6 ←  $\phi(d_2, d_5)$ 
      y  ← +, a4, b4
      z  ← +, c6, d6
      i2 ← +, i1, 1
```

```
B6  c5 ←  $\phi(c_2, c_4)$ 
      d5 ←  $\phi(d_4, d_3)$ 
      b3 ←
```

```
B2  b2 ←
      c3 ←
      d2 ←
```

```
B7  y  ← +, a4, b4
      z  ← +, c6, d6
      i2 ← +, i1, 1
```

```
B6  b3 ←
      a4 ← a2
      b4 ← b2
      c6 ← c3
      d6 ← d2
```

```
B2  b2 ←
      c3 ←
      d2 ←
      a4 ← a3
      b4 ← b3
      c6 ← c5
      d6 ← d5
```


7.4 Восстановление кода из SSA-формы

7.4.2. Замена ϕ -функций группами инструкций копирования

B_7 $a_4 \leftarrow \phi(a_2, a_3)$
 $b_4 \leftarrow \phi(b_2, b_3)$
 $c_6 \leftarrow \phi(c_3, c_5)$
 $d_6 \leftarrow \phi(d_2, d_5)$
 $y \leftarrow +, a_4, b_4$
 $z \leftarrow +, c_6, d_6$
 $i_2 \leftarrow +, i_1, 1$

B_6 $c_5 \leftarrow \phi(c_2, c_4)$
 $d_5 \leftarrow \phi(d_4, d_3)$
 $b_3 \leftarrow$

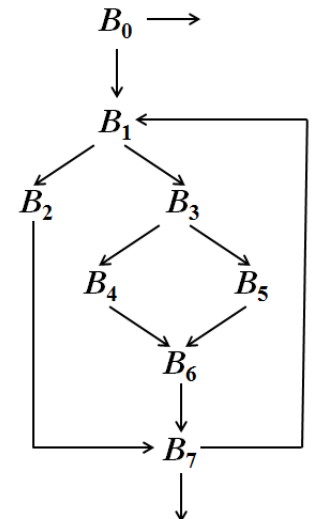
B_2 $b_2 \leftarrow$
 $c_3 \leftarrow$
 $d_2 \leftarrow$

B_7 $y \leftarrow +, a_4, b_4$
 $z \leftarrow +, c_6, d_6$
 $i_2 \leftarrow +, i_1, 1$

B_6 $b_3 \leftarrow$
 $a_4 \leftarrow a_2$
 $b_4 \leftarrow b_2$
 $c_6 \leftarrow c_3$
 $d_6 \leftarrow d_2$

B_2 $b_2 \leftarrow$
 $c_3 \leftarrow$
 $d_2 \leftarrow$
 $a_4 \leftarrow a_3$
 $b_4 \leftarrow b_3$
 $c_6 \leftarrow c_5$
 $d_6 \leftarrow d_5$

- ◇ Удаление ϕ -функций из блока B_6 породит инструкции копирования в блоках B_4 и B_8 .
- ◇ Удаление ϕ -функций из блока B_1 должно породить инструкции копирования в блоках B_0 и B_7 . Но этого делать нельзя!



7.4 Восстановление кода из SSA-формы

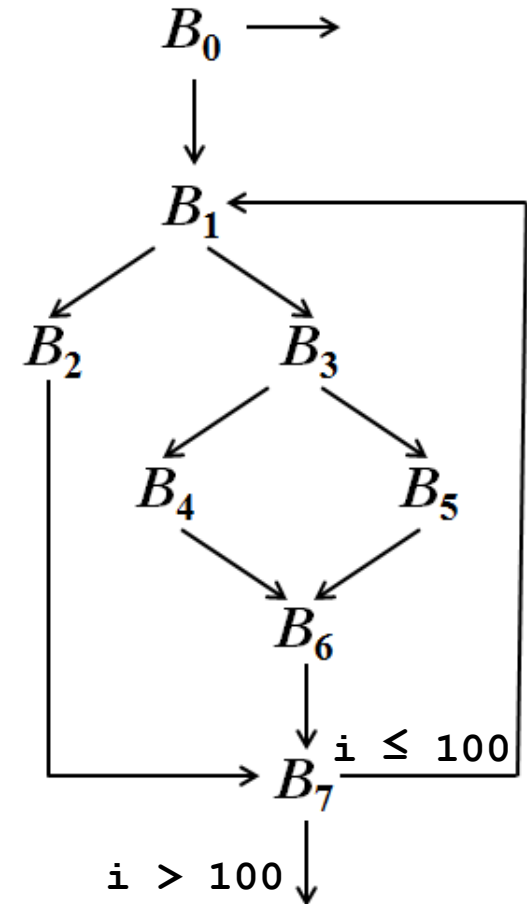
7.4.2. Замена ϕ -функций группами инструкций копирования

- Удаление ϕ -функций из блока B_1 должно породить инструкции копирования в блоках B_0 и B_7 .

Но этого делать нельзя! Почему?

- У блоков B_0 и B_7 по два последующих блока (вторые блоки не попали на схему, так как они находятся вне анализируемого цикла).

Если вставить копирования в блоки B_0 и B_7 , они будут выполняться не только на ребрах, внутри рассматриваемого цикла, но и на ребрах выводящих из цикла.



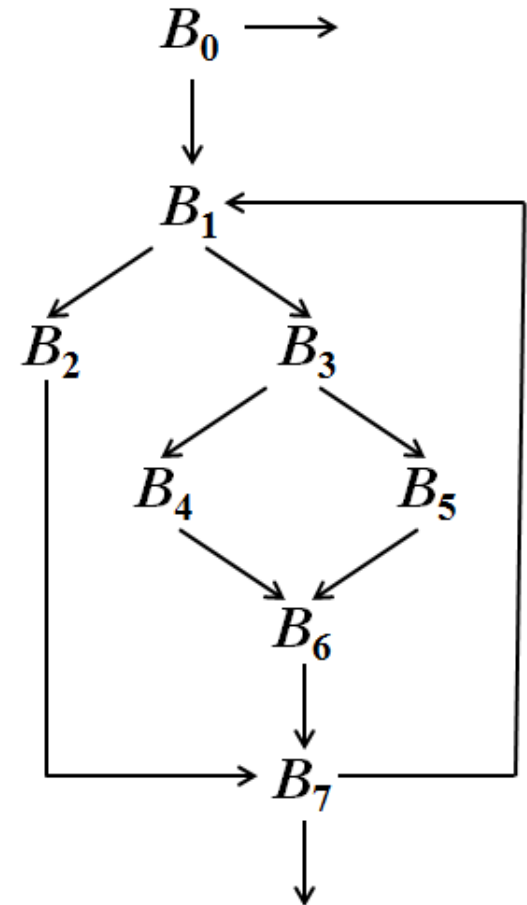
7.4 Восстановление кода из SSA-формы

7.4.2. Замена Φ -функций группами инструкций копирования

- По определению ребро, выходящее из вершины с несколькими потомками, и входящее в вершину с несколькими предками, называется *критическим*.

Ребра $(B_0 B_1)$ и $(B_7 B_1)$ – критические.

- Критические ребра обычно исключают, разрывая их и вставляя в разрыв дополнительные вершины.
- Разорвем критические ребра и добавим два новых блока B_8 и B_9 , поместив в них требуемые инструкции копирования.



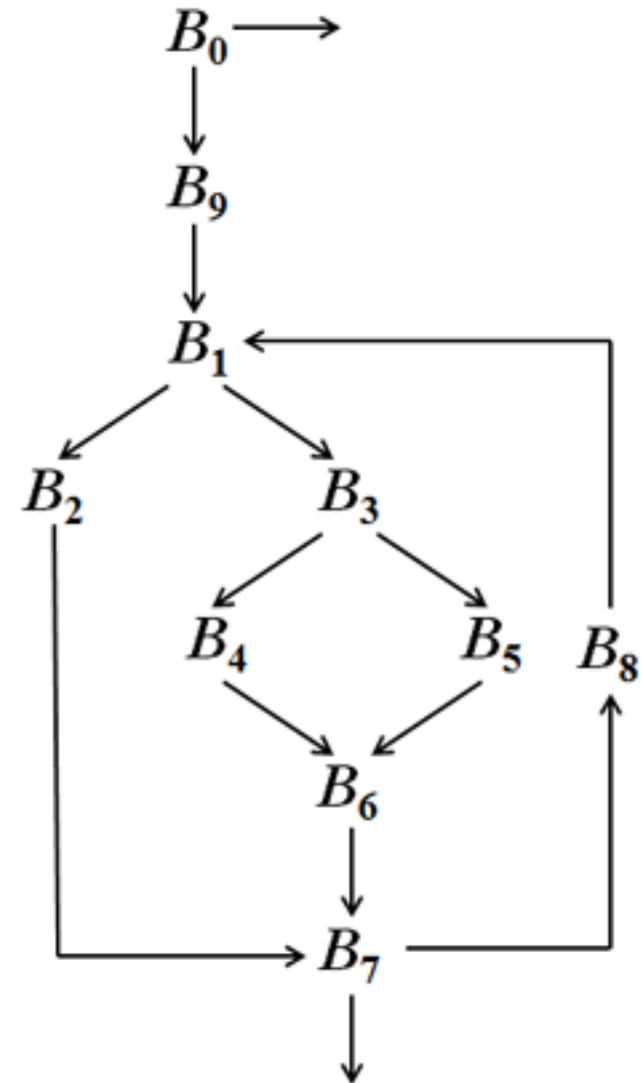
7.4 Восстановление кода из SSA-формы

7.4.2. Замена Φ -функций группами инструкций копирования

- По определению ребро, выходящее из вершины с несколькими потомками, и входящее в вершину с несколькими предками, называется *критическим*.

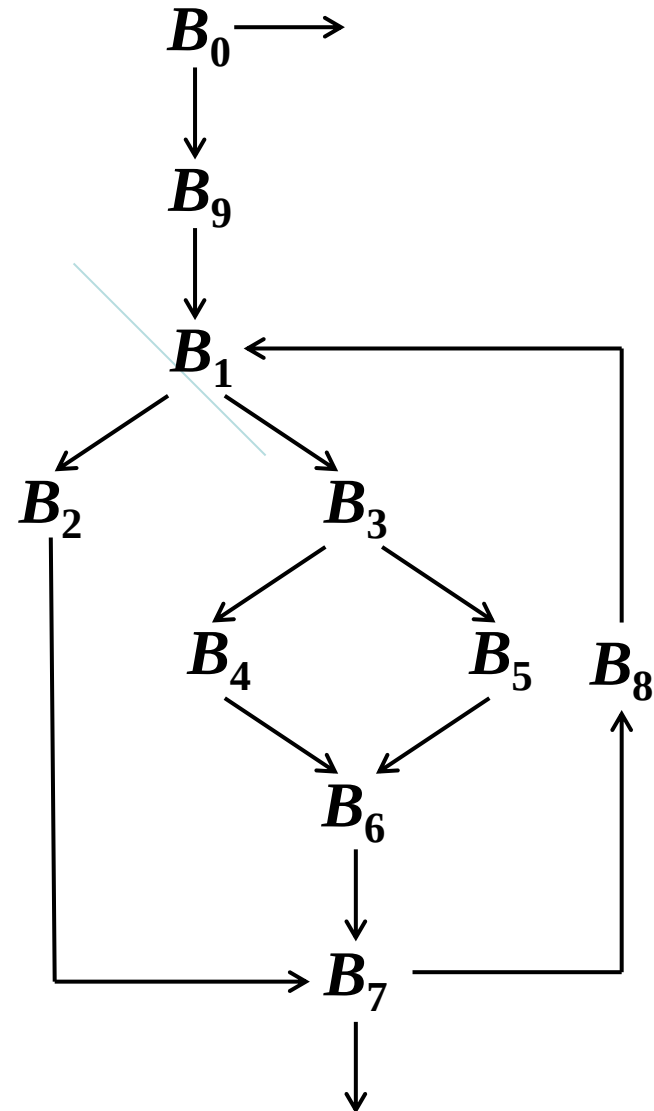
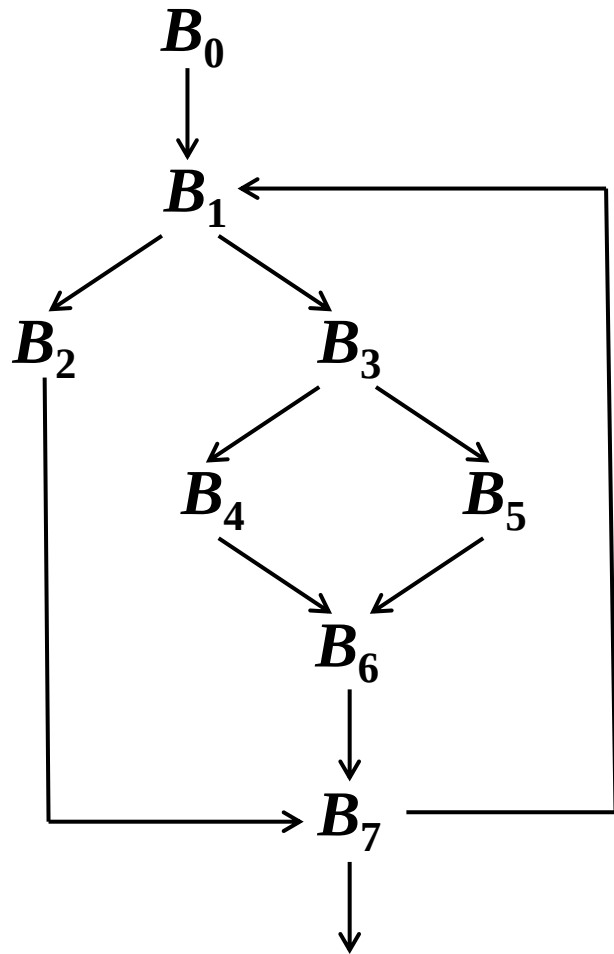
Ребра $(B_0 B_1)$ и $(B_7 B_1)$ – критические.

- Критические ребра обычно исключают, разрывая их а вставляя в разрыв дополнительные вершины.
- Разорвем критические ребра и добавим два новых блока B_8 и B_9 , поместив в них требуемые инструкции копирования.



7.4 Восстановление кода из SSA-формы

7.4.2. Замена Φ -функций группами инструкций копирования



8. Распространение констант

8.1 Глобальное распространение констант

8.1.1 Постановка задачи

- ◇ **Распространение констант** — это оптимизирующее преобразование, заменяющее **выражения**, которые при выполнении всякий раз вычисляют одну и ту же константу, **самой этой константой**.
- ◇ Преобразование удобно выполнять над процедурой в SSA-форме.

8.1 Глобальное распространение констант

8.1.1 Постановка задачи

- ◇ **Распространение констант** — это оптимизирующее преобразование, заменяющее **выражения**, которые при выполнении всякий раз вычисляют одну и ту же константу, самой этой **константой**.
- ◇ Преобразование удобно выполнять над процедурой (программой) в SSA-форме.
- ◇ Распространение констант является прямой задачей потока данных.

8.1 Глобальное распространение констант

8.1.1 Постановка задачи

- ◇ **Распространение констант** – это оптимизирующее преобразование, заменяющее **выражения**, которые при выполнении всякий раз вычисляют одну и ту же константу, самой этой **константой**.
- ◇ Преобразование удобно выполнять над процедурой (или файлом, содержащим несколько процедур) в *SSA*-форме.
- ◇ Распространение констант является прямой задачей потока данных.
- ◇ Выполняя глобальный анализ потока данных (методом итераций) рассматриваем в процедуре все *SSA*-имена v и каждому из них *сопоставляем константу*; это может быть:
 - ◇ специальная константа *Undef* – значение переменной не определено;
 - ◇ специальная константа *NAC* (*not-a-constant*) – переменная не может быть константой;
 - ◇ одна из множества рассматриваемых констант.

8.1 Глобальное распространение констант

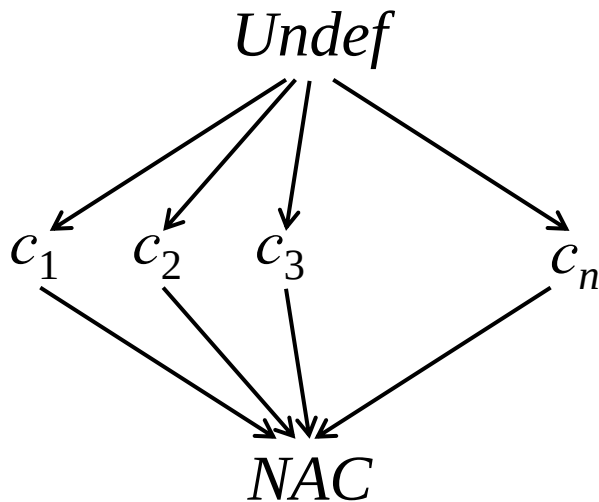
8.1.2 Константы *NAC* и *Undef*

- ◇ ***SSA*-имени v сопоставляется константа *NAC* если установлено, что:**
 - ◇ v может быть присвоено входное значение, т.е.
 - ◆ v — имя формального параметра процедуры или
 - ◆ v — имя целевой переменной функции ввода;
 - ◇ при вычислении v используется *NAC*
 - ◇ на различных путях вычисления v могут быть присвоены разные значения.
- ◇ ***SSA*-имени v сопоставляется константа *Undef*, если не найдено ни одного определения v , достигающего рассматриваемой точки. На самом деле всем *SSA*-именам сначала сопоставляется константа *Undef*.**

8.1 Глобальное распространение констант

8.1.2 Полурешетка значений констант

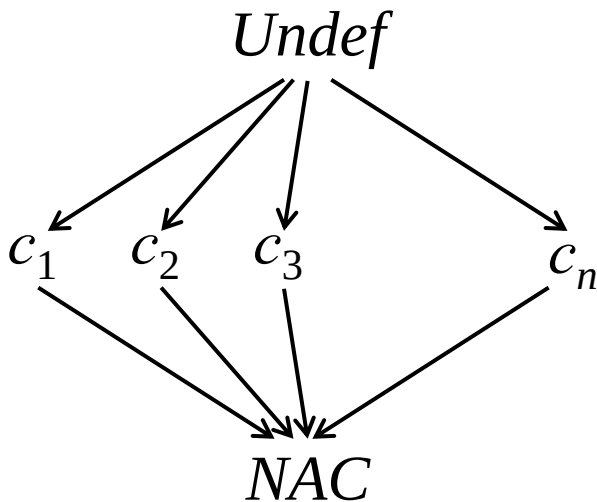
- ◇ Множество значений, сопоставляемых каждой переменной, (включая специальные значения *Undef* и *NAC*) можно представить как полурешетку (см. рисунок).



8.1 Глобальное распространение констант

8.1.2 Полурешетка значений констант

- ◇ Множество значений, сопоставляемых каждой переменной, (включая специальные значения *Undef* и *NAC*) можно представить как полурешетку (см. рисунок).



- ◇ верхний элемент $\top = Undef$
- ◇ нижний элемент $\perp = NAC$
- ◇ Значения констант (c_i) не сравнимы между собой, но все они «меньше» *Undef* и «больше» *NAC*:
 $\forall i: NAC \leq c_i \leq Undef$

8.1 Глобальное распространение констант

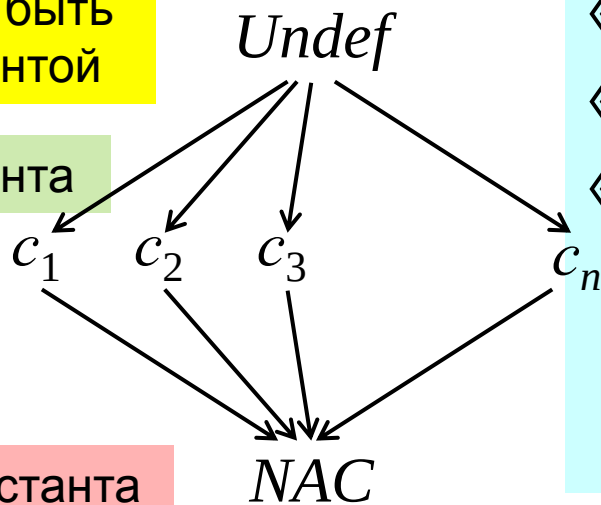
8.1.2 Полурешетка значений констант

- ◇ Множество значений, сопоставляемых каждой переменной, (включая специальные значения *Undef* и *NAC*) можно представить как полурешетку (см. рисунок).

может быть
константой

константа

не константа



◇

верхний элемент $\top = Undef$

◇

нижний элемент $\perp = NAC$

◇

Значения констант (c_i) не сравнимы между собой, но все они

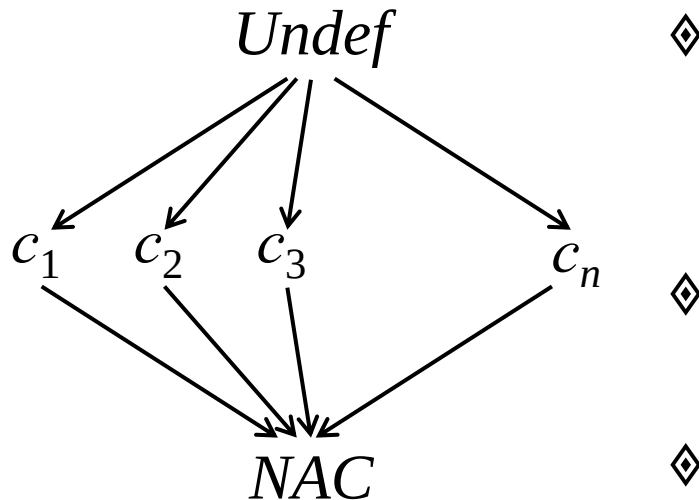
«меньше» *Undef* и «больше» *NAC*:

$$\forall i: NAC \leq c_i \leq Undef$$

8.1 Глобальное распространение констант

8.1.2 Полурешетка значений констант

- ◇ Множество значений, сопоставляемых каждой переменной, (включая специальные значения *Undef* и *NAC*) можно представить как полурешетку (см. рисунок).



для всех значений v :

$$Undef \wedge v = v$$

$$NAC \wedge v = NAC$$

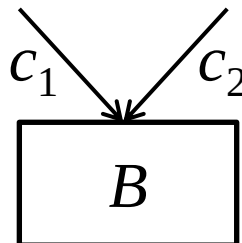
в частности

$$NAC \wedge Undef = NAC$$

по определению

для любых двух констант c_1 и c_2

$$c_1 \wedge c_2 = \begin{cases} c_1 & \text{если } c_1 = c_2 \\ NAC & \text{если } c_1 \neq c_2 \end{cases}$$



8.1 Глобальное распространение констант

8.1.3 Операция сбора в структуре распространения констант

- ◇ Значение потока данных в структуре распространения констант представляет собой отображение

$$v: \mathcal{N} \rightarrow C,$$

где $\mathcal{N}$ – множество *SSA*-имен

C – полурешетка констант

Значение, соответствующее *SSA*-имени m в отображении v , обозначается как $v(m)$

- ◇ Полурешетка значений потока данных представляет собой декартово произведение k полурешеток C , где k – количество переменных в анализируемой процедуре:

$$v \wedge v' = v'' \quad \Leftrightarrow \quad \forall m: v''(m) = v(m) \wedge v'(m).$$

$$v \leq v' \quad \Leftrightarrow \quad \forall m: v(m) \leq v'(m)$$

8.1 Глобальное распространение констант

8.1.4 Передаточные функции структуры распространения констант

- ◇ Множество передаточных функций F_{Const} состоит из функций вида

$$f: \mathcal{V} \rightarrow \mathcal{V},$$

где $\mathcal{V}$ – множество отображений $v: \mathcal{N} \rightarrow \mathcal{C}$.

- ◇ F_{Const} содержит тождественную функцию i :

$$\forall v \in \mathcal{V} \quad i(v) = v$$

- ◇ F_{Const} содержит константную передаточную функцию c :

$$\forall v \in \mathcal{V} \quad c(v) = v_0$$

где

$$\forall v \in \mathcal{V} \quad v_0(v) = \text{Undef}$$

В частности, $c(\text{Entry}) = v_0$ означает, что перед началом выполнения программы не определена ни одна переменная.

8.1 Глобальное распространение констант

8.1.4 Передаточные функции структуры распространения констант

- ◇ Пусть $f_s \in \mathcal{F}_{Const}$ – передаточная функция для инструкции s и пусть v и $v' \in \mathcal{V}$ – значения потока данных в точках до и после s : $v' = f_s(v)$.
- ◇ Опишем f_s через v и v' .
 - (1) Если s – не содержит присваивания какому-либо SSA -имени, то f_s – тождественная функция i , т.е. для всех v $v'(m) = v(m)$
 - (2) Если s – содержит присваивание SSA -имени x , то для $m \neq x$ $v'(m) = v(m)$,
для $m = x$ $v'(m) = v(x)$

8.1 Глобальное распространение констант

8.1.4 Передаточные функции структуры распространения констант

◇ Описание f_s через m и m'

(2) Пусть s – присваивание переменной x и пусть $v = x$
 $m' = m'(x)$ определяется следующим образом:

(a) Если s – присваивание константы c : $x = c$, то
 $m'(x) = c$

(b) Если s – присваивание выражения: $x = y + z$, то

$$m'(x) = \begin{cases} m(y) + m(z), & \text{если } m(y) \text{ и } m(z) \text{ константы} \\ NAC, & \text{если } m(y) = NAC \text{ или } m(z) = NAC \\ Undef & \text{в остальных случаях} \end{cases}$$

(c) Если s – присваивание любого другого выражения (например, вызова функции или указателя), то
 $m'(x) = NAC$.

8.1 Глобальное распространение констант

8.1.5 Монотонность структуры распространения констант

◇ **Утверждение.** Структура распространения констант **МОНОТОННА**:

если для m_1, m_2, m'_1, m'_2 выполняются соотношения:

$$m_1 \leq m_2, \quad m'_1 = f_s(m_1) \quad m'_2 = f_s(m_2)$$

то $m'_1 \leq m'_2$

◇ **Доказательство.** Рассмотрим определение f_s через m и m' .

Случай 1): $f_s \equiv i \Rightarrow m'_1 = m_1, m'_2 = m_2 \Rightarrow m'_1 \leq m'_2$

Случай 2а): $\forall m: f_s(m) = c \Rightarrow m'_1 = m'_2 = c \Rightarrow m'_1 \leq m'_2$

Случай 2b): на следующем слайде

Случай 2с): $\forall m: f_s(m) = NAC \Rightarrow m'_1 = m'_2 = NAC \Rightarrow m'_1 \leq m'_2$

8.1 Глобальное распространение констант

8.1.5 Монотонность структуры распространения констант

◇ Доказательство утверждения (случай 2b)

Для присваивания $x = y + z$ передаточная функция f_s описывается формулой

$$m'(x) = \begin{cases} m(y) + m(z), & \text{если } m(y) \text{ и } m(z) \text{ константы} \\ NAC, & \text{если } m(y) = NAC \text{ или } m(z) = NAC \\ Undef & \text{в остальных случаях} \end{cases}$$

которой соответствует таблица справа.

Из таблицы видно, что во всех случаях

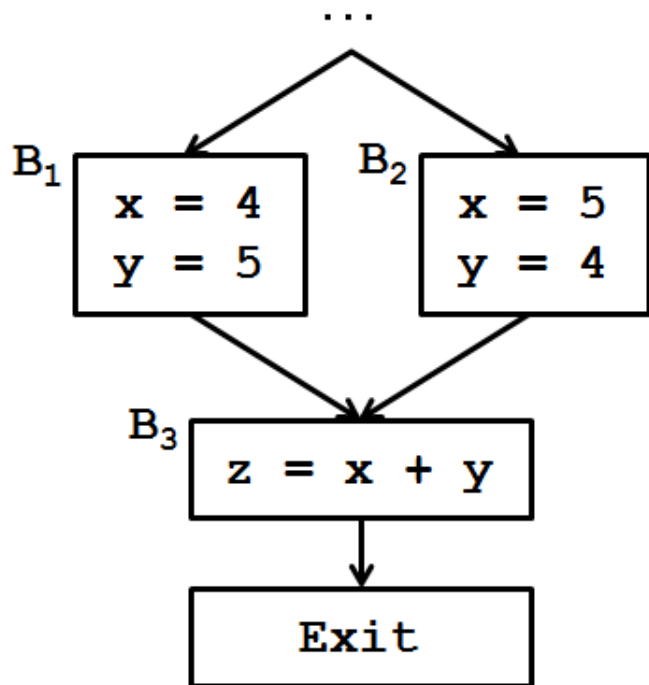
$$m'(x) \leq m(x)$$

$m(y)$	$m(z)$	$m'(x)$
<i>Undef</i>	<i>Undef</i>	<i>Undef</i>
	c_2	<i>Undef</i>
	<i>NAC</i>	<i>NAC</i>
c_1	<i>Undef</i>	<i>Undef</i>
	c_2	$c_1 + c_2$
	<i>NAC</i>	<i>NAC</i>
<i>NAC</i>	<i>Undef</i>	<i>NAC</i>
	c_2	<i>NAC</i>
	<i>NAC</i>	<i>NAC</i>

8.1 Глобальное распространение констант

8.1.6 Недистрибутивность структуры распространения констант

◇ Для установления отсутствия дистрибутивности у передаточной функции структуры распространения констант рассмотрим пример



Пример.

Итеративный алгоритм не может обнаружить, что в программе на рисунке слева значение $x + y$ на входе в блок B_3 всегда равно 9

И x , и y на входе в B_3 имеют значение *NAC*

Этот результат безопасен (консервативен), но неточен: не удастся отследить зависимость:

когда x равно 4, y равно 5,

а когда x равно 5, y равно 4.

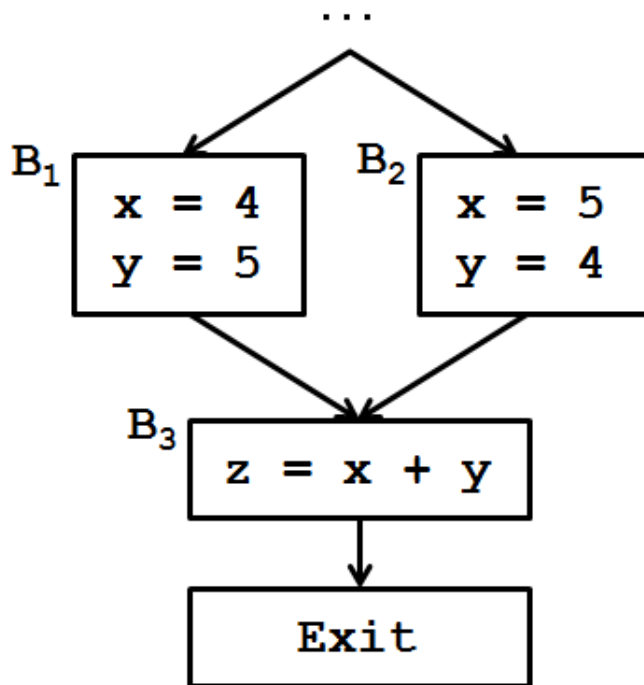
8.1 Глобальное распространение констант

8.1.6 Недистрибутивность структуры распространения констант

- ◇ Таким образом, структура распространения констант не дистрибутивна, т.е. *MFP*-решение безопасно, но необязательно равно *MOP*-решению и может оказаться «меньше» (грубее) него.

Пример.

Итеративный алгоритм не сможет обнаружить, что в программе на рисунке слева значение $\mathbf{x} + \mathbf{y}$ на входе в блок B_3 всегда равно **9**, так как и $\mathbf{x}$, и $\mathbf{y}$ на входе в B_3 имеют значение **НАС**



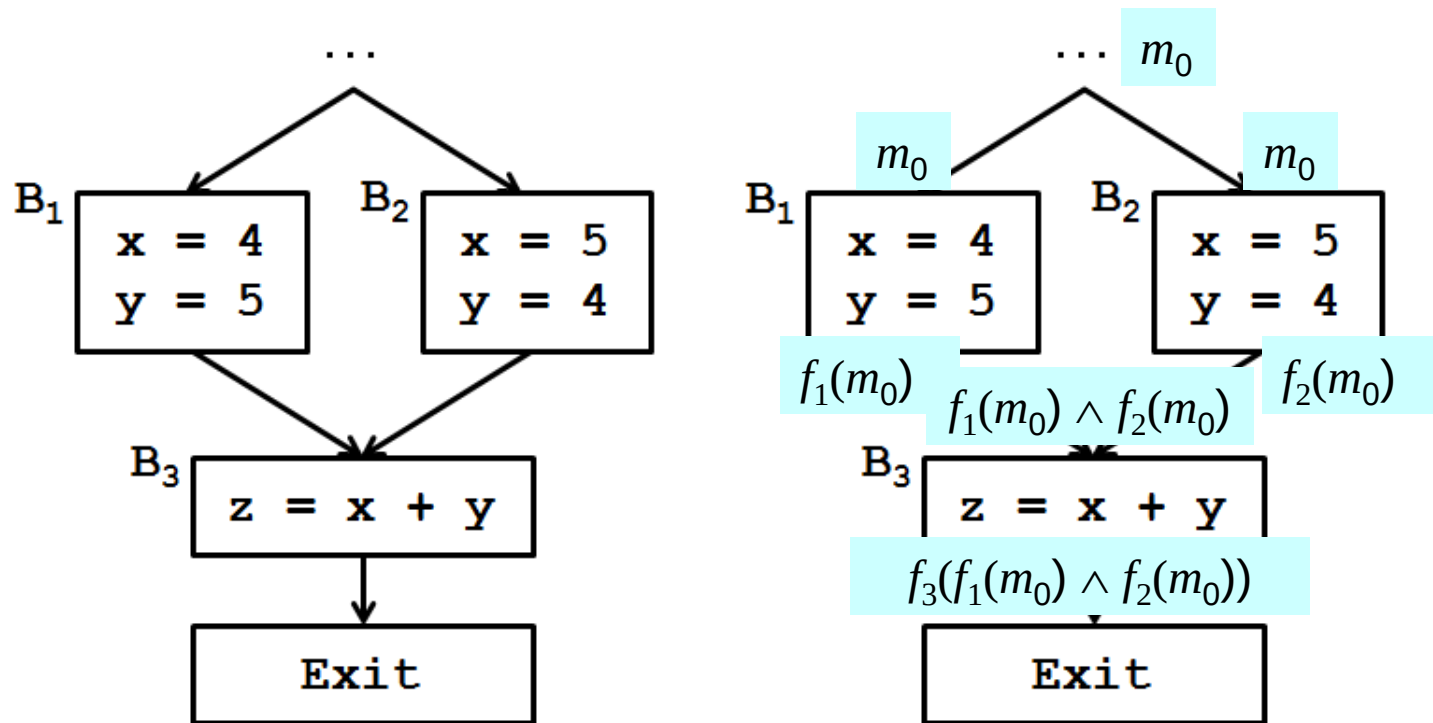
Этот результат безопасен (консервативен), но неточен: из-за недистрибутивности структуры распространения констант итеративный алгоритм не может отследить зависимость: когда $\mathbf{x}$ равно 4, $\mathbf{y}$ равно 5, а когда $\mathbf{x}$ равно 5, $\mathbf{y}$ равно 4.

8.1 Глобальное распространение констант

8.1.6 Недистрибутивность структуры распространения констант

◇ Окончание примера.

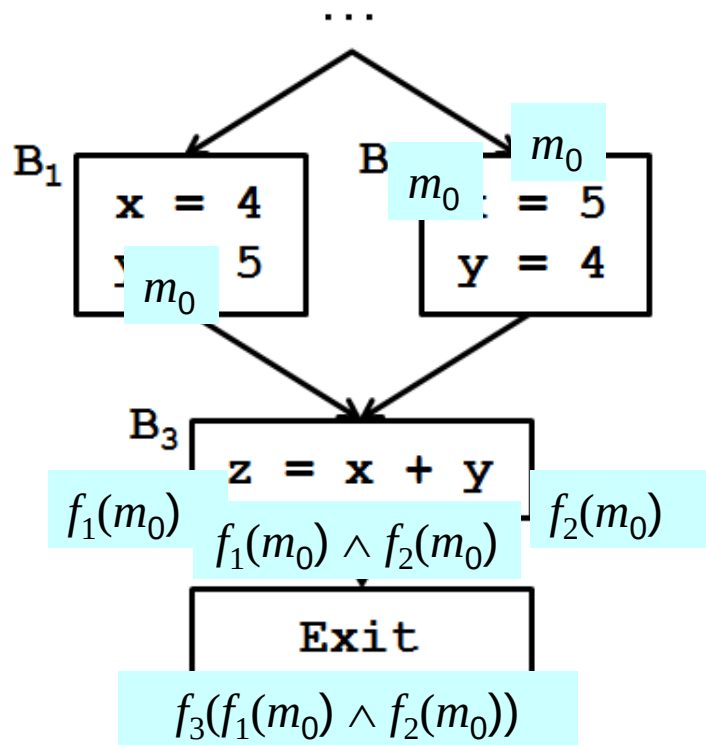
Пусть f_1 , f_2 и f_3 – передаточные функции блоков B_1 , B_2 и B_3 соответственно, а m_0 – состояние на входе в блоки B_1 и B_2 . Тогда



8.1 Глобальное распространение констант

8.1.6 Недистрибутивность структуры распространения констант

◇ **Пример.** Таким образом, как видно из таблицы



m	$m(x)$	$m(y)$	$m(z)$
m_0	<i>Undef</i>	<i>Undef</i>	<i>Undef</i>
$f_1(m_0)$	4	5	<i>Undef</i>
$f_2(m_0)$	5	4	<i>Undef</i>
$f_1(m_0) \wedge f_2(m_0)$	<i>NAC</i>	<i>NAC</i>	<i>Undef</i>
$f_3(f_1(m_0) \wedge f_2(m_0))$	<i>NAC</i>	<i>NAC</i>	<i>NAC</i>
$f_3(f_1(m_0))$	4	5	9
$f_3(f_2(m_0))$	5	4	9
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	<i>NAC</i>	<i>NAC</i>	9

$f_3(f_1(m_0) \wedge f_2(m_0)) = \text{NAC}$, а $f_3(f_1(m_0)) \wedge f_3(f_2(m_0)) = 9$,

откуда следует **недистрибутивность** структуры:

$$f_3(f_1(m_0) \wedge f_2(m_0)) \neq f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

8.1 Глобальное распространение констант

8.1.7 Интерпретация значения *Undef*

- ◇ Значение *Undef* используется в итеративном алгоритме:
 - (1) для инициализации входного узла (граничное условие).
 - (2) для инициализации внутренних точек программы на нулевой итерации
- ◇ Смысл случаев (1) и (2):
 - ◇ **Случай (1):** в начале выполнения программы значения ее переменных не определены.
В конце итеративного процесса переменные на выходе из *Entry* сохраняют значение *Undef*, поскольку *Out[Entry]* не изменяется.

8.1 Глобальное распространение констант

8.1.7 Интерпретация значения *Undef*

◇ Смысл случаев (1) и (2):

◇ **Случай (2):** из-за недостатка информации в начале итерационного процесса решение аппроксимируется верхним элементом *Undef*.

В конце итерационного процесса эти *Undef* сохранятся только в тех точках, для которых не найдется определений соответствующих переменных ни на одном пути, ведущем к этим точкам.

Если же найдется хотя бы один путь, содержащий определение переменной, достигающее рассматриваемой точки программы, то эта переменная не будет иметь значение *Undef*.

8.1 Глобальное распространение констант

8.1.7 Интерпретация значения *Undef*

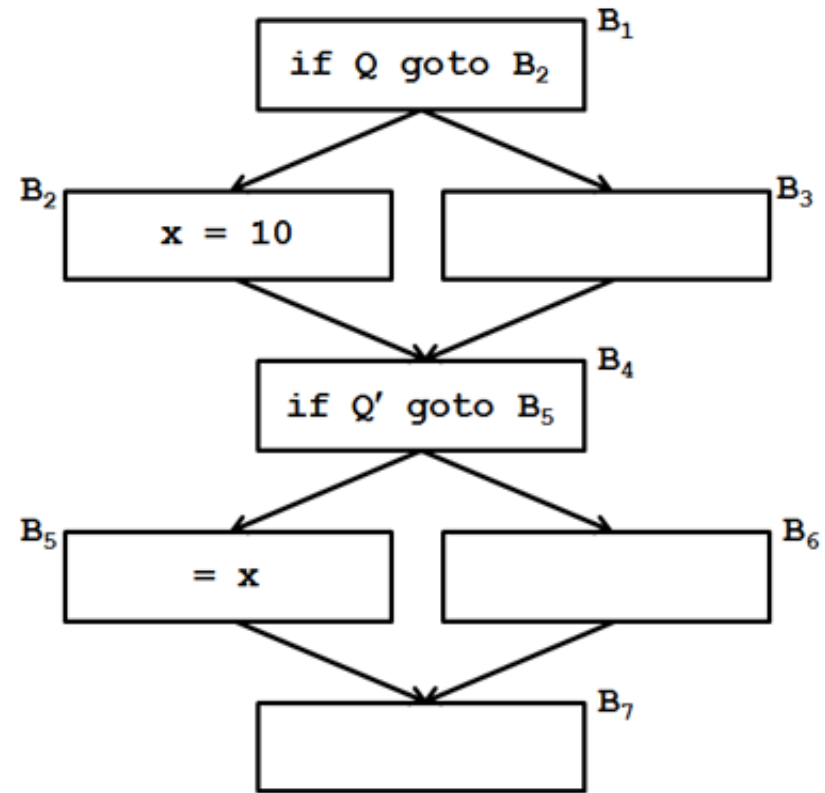
- ◇ Если все определения какой-либо переменной, достигающие некоторой точки программы, имеют одно и то же константное значение (но не *Undef* и не *NAC*), то эта переменная рассматривается как константа, **даже если может оказаться, что она не определена на одном из путей.**
- ◇ Если предположить, что оптимизируемая программа корректна, то итеративный алгоритм может найти большее количество констант, т.к. он может считать константами и те переменные, которые на части путей имеют значения *Undef*.
Такой подход правомерен для языков программирования, которые исходят из того, что неопределенная (*Undef*) переменная может иметь любое значение.
Если семантика языка программирования требует, чтобы все неопределенные переменные принимали некоторое конкретное значение, необходимо это ограничение включить в формулировку задачи анализа потока данных.

8.1 Глобальное распространение констант

8.1.7 Интерпретация значения *Undef*

◇ На рисунке справа значения x на выходе из блоков B_2 и B_3 – соответственно 10 и *Undef*. Из $Undef \wedge 10 = 10$ следует, что значение x на входе в B_4 равно 10, и в B_5 можно заменить x на 10.

◇ Если выполнение пойдет по пути $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5$, то блока B_5 будет достигать значение $x = Undef$, и замена x на 10 будет корректной не для всех исходных языков.



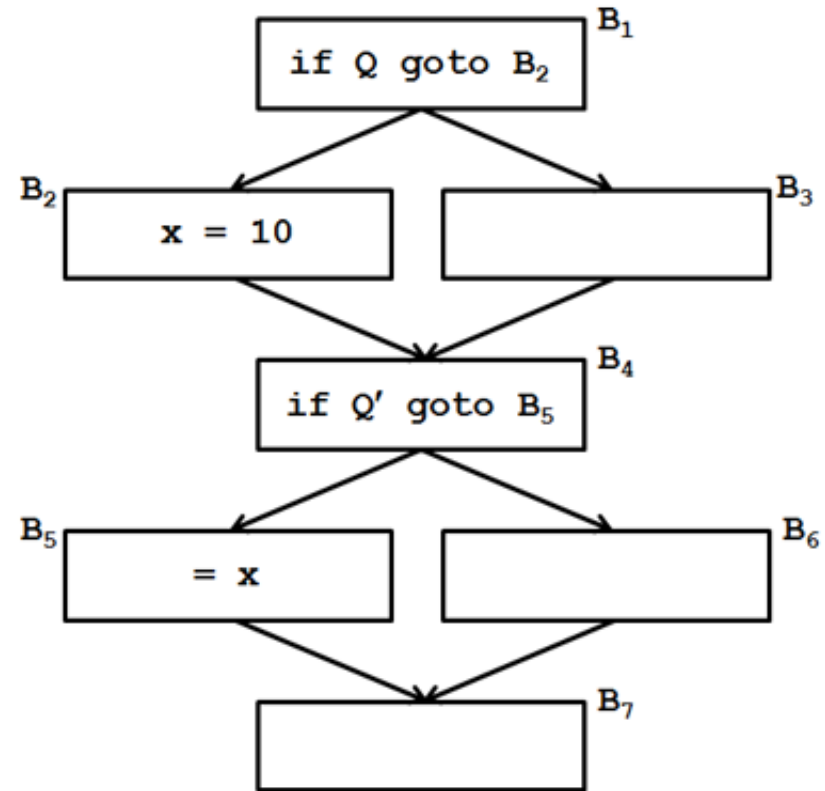
8.1 Глобальное распространение констант

8.1.7 Интерпретация значения *Undef*

Пример.

- ◇ Даже если можно доказать, что предикат Q не может иметь значение *false*, когда предикат Q' имеет значение *true*, и поэтому путь $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5$ никогда не будет пройден, следует понимать, что **такой вывод находится за пределами возможностей анализа потоков данных.**

- ◇ Таким образом, если считать, что исходная программа корректна и что все ее переменные определяются до их использования, то корректно считать, что x при входе в B_5 определена, и ее значение равно 10.



8.2 Алгоритм глобального распространения констант

8.2.1 Фаза инициализации

// Initialization Phase

WorkList = $\emptyset$;

for each SSA-name **v**

initialize **Value(v)** by rules specified in the text;

if **Value(v)** $\neq$ **Undef** then

WorkList = **WorkList** $\cup$ {**v**};

На фазе инициализации устанавливаются начальные значения *SSA*-имен и формируется начальное состояние **WorkList**.

Для каждого *SSA*-имени **v** анализируется операция, определяющая **v** и устанавливающая значение **Value(v)** в соответствии со следующими простыми правилами (следующий слайд).

8.2 Алгоритм глобального распространения констант

8.2.1 Фаза инициализации

```
// Initialization Phase
WorkList =  $\emptyset$ ;
for each SSA-name  $v$ 
  initialize Value( $v$ ) by rules specified in the
  text;
if Value( $v$ )  $\neq$  Undef then
  WorkList = WorkList  $\cup$  { $v$ };
```

The rules specified in the text

1. v определяется ϕ -функцией, Value(v) = Undef.
2. v равно одной из констант c_i , Value(v) = c_i .
3. v может быть присвоено входное значение, Value(v) = NAC.
4. Значение v неизвестно, Value(v) = Undef.

Если Value(v) $\neq$ Undef, алгоритм добавляет v в WorkList 26

8.2 Алгоритм глобального распространения констант

8.2.2 Фаза распространения

```
// Propagation Phase
// Выполнять итерации (до неподвижной точки)
while (WorkList  $\neq \emptyset$ ;)
  remove some w from WorkList      //Выбрать
                                    очередное имя
  for each operation op that uses w
    let w be the SSA-name that op defines
    if Value(w)  $\neq$  NAC then        //Вычислить снова
                                    и проверить не
                                    изменилось ли
    t  $\leftarrow$  Value(w)
    Value(w)  $\leftarrow$  result of interpreting op over
                        lattice values
    if Value(w)  $\neq$  t
    then WorkList  $\leftarrow$  WorkList  $\cup$  {w};
```


8.2 Алгоритм глобального распространения констант

8.2.2 Фаза распространения

Фаза распространения проста:

Из **WorkList** извлекается (с удалением) очередное SSA-имя **w**.

Алгоритм анализирует каждую операцию **op**, которая, используя **w**, определяет SSA-имя **w'**.

Если **Value(w') = NAC**, дальнейший анализ **w** не имеет смысла: мы уже скатились в **NAC** и больше не поднимемся из-за монотонности функции.

В противном случае, моделируется выполнение **op** с помощью интерпретации этой операции над полурешетками значений ее операндов.

Если результат отличен от **Value(w)**, он рассматривается как новое значение **Value(w)**, причем **w** добавляется в **WorkList**.

Алгоритм завершается, когда **WorkList** пуст.

8.2 Алгоритм глобального распространения констант

8.2.2 Фаза распространения

Интерпретация операции над значениями полурешетки требует некоторой осторожности.

Для ϕ -функции результаты операции получаются применением операции «сбор» ко всем аргументам ϕ -функции; правила вычисления результатов операции «сбор» на рисунке справа (в порядке предпочтения)

Правила для операции $\wedge$

$$T \wedge x = x \quad \forall x$$

$$\perp \wedge x = \perp \quad \forall x$$

$$c_i \wedge c_j = c_i \quad c_i = c_j$$

$$c_i \wedge c_j = \perp \quad c_i \neq c_j$$

Что касается других операций компилятор должен «знать» особенности их выполнения. Если любой операнд имеет значение T , необходимо вернуть результат T . Если ни один из операндов не имеет значения T , необходимо вернуть соответствующее значение.

Для каждой операции, вычисляющей значение, алгоритму необходимо иметь набор правил, моделирующих поведение операндов.

Рассмотрим, например операцию $a \times b$. Если $a = 4$, а $b = 17$, необходимо выдать в качестве результата $a \times b$ значение 68. Однако, если $a = \perp$, необходимо вернуть результат $\perp$ для всех значений b , кроме $b = 0$, потому что $a \times 0 = 0$ независимо от значения a .

8.2 Алгоритм глобального распространения констант

8.2.3 Заключительные замечания



Сложность.

Фаза распространения – это классическая схема нахождения фиксированной точки. Доказательство сходимости и завершимости, а также оценка сложности алгоритма опирается на факт конечности нисходящих цепочек.

Переменная, связанная с каждым SSA-именем может иметь одно из трех начальных значений: какая-нибудь константа c_i , отличная от *Undef*, или *Undef*, или *NAC*.

Фаза распространения может только «уменьшить» ее значение. Для данного SSA-имени это может произойти не более, чем дважды: от *Undef* к c_i и от c_i к *NAC*. Следовательно, каждое SSA-имя может встретиться в *Worklist* не более, чем два раза: алгоритм интерпретирует операцию, когда один из операндов удаляется из *Worklist*. Так что общее число интерпретаций не больше, чем удвоенное число операций.

8.2 Алгоритм глобального распространения констант

8.2.3 Заключительные замечания

◇ Ценность SSA-формы

SSA-форма позволила разработать простой, эффективный и понятный алгоритм глобального распространения констант.

Рассмотрим для сравнения классический подход к проблеме распространения констант методом анализа потока данных. Для его разработки потребуется рассмотреть множество Constants в каждом базовом блоке B_i , составить систему уравнений относительно неизвестных $In[B_i]$, определить передаточную функцию для вычисления $Out[B_i]$. Несомненно представленный выше алгоритм гораздо проще. Наибольшую сложность вызывает разработка традиционного механизма интерпретации операций, но в остальном это простой итерационный алгоритм нахождения неподвижной точки над чрезвычайно разреженной решеткой.

Алгоритм работает гораздо быстрее традиционного анализа потока данных.

9. Глобальная нумерация значений

9.1. Базовый алгоритм локальной нумерации значений

9.1.1. Вводные замечания

- ◇ *Нумерация значений* – классический метод анализа и преобразования промежуточного представления (трехадресного кода) в *ОАГ* – ориентированный ациклический граф (лекция 1, п. 1.6.2)

Идея: алгоритм присваивает индивидуальный номер каждому значению, которое будет вычислено во время выполнения компилируемой программы, при этом два выражения e_i и e_j получают один и тот же номер тогда и только тогда, когда удастся доказать, что значения выражений e_i и e_j равны для всех возможных значений их операндов.

9.1. Базовый алгоритм локальной нумерации значений

9.1.1. Вводные замечания

- ◇ При построении ОАГ удобно представить его в виде таблицы значений (*TЗ*), каждая строка ТЗ (массива структур) представляет один узел ОАГ.
- ◇ Терминальные узлы ОАГ – идентификаторы (**id**) переменных, определенных вне рассматриваемого базового блока, и константы (**nm**) представляются в ТЗ структурами из двух полей **<op, ref_ST>** (ссылка **ref_ST** отсылает к таблице символов).
- ◇ Нетерминальные узлы ОАГ – выражения представляются своими сигнатурами: выражению **<op, left, right>**, где **op** – код операции, а **left** и **right** – левый и правый операнды, соответствует *сигнатура* **<op, #left, #right>**, где **#left** и **#right** – номера значений левого и правого операндов (если **op** – унарная операция, то **#right** = 0).
- ◇ Номер значения – это номер строки ТЗ, в которой находится сигнатура операции, определяющей это значение

9.1. Базовый алгоритм локальной нумерации значений

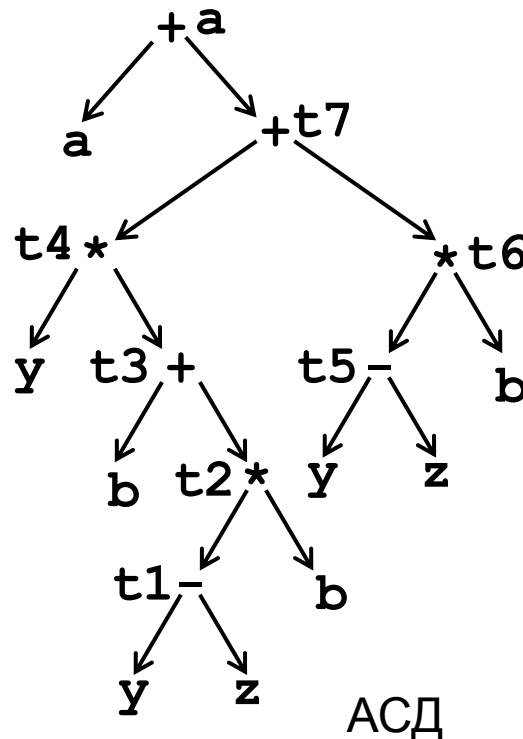
9.1.2 Представление базового блока в виде ориентированного ациклического графа

◇ Пример. Выражение в исходном коде:

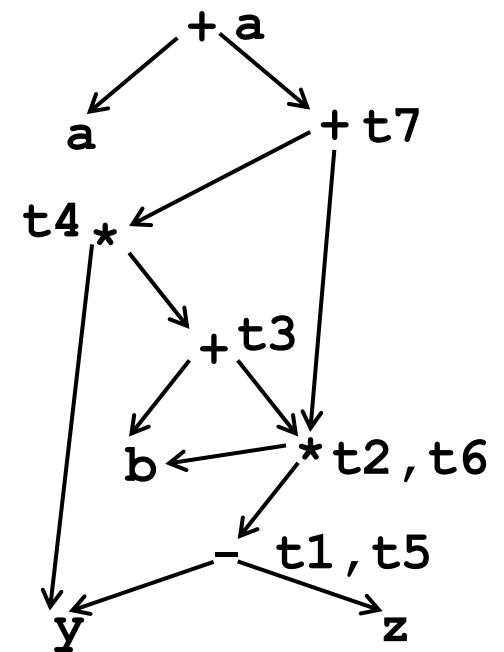
$$a = a + y * (b + (y - z) * b) + (y - z) * b$$

```
t1 ← -, y, z
t2 ← *, t1, b
t3 ← +, b, t2
t4 ← *, y, t3
t5 ← -, y, z
t6 ← *, t5, b
t7 ← +, t4, t6
a ← +, a, t7
```

Выражение в
промежуточном
представлении



АСД



ОАГ

9.1. Базовый алгоритм локальной нумерации значений

9.1.2 Представление ОАГ в виде таблицы значений

◇ Таблица значений рассматриваемого примера имеет вид:

$t1^5 \leftarrow -, y^3, z^4$
 $t2^6 \leftarrow *, t1^5, b^2$
 $t3^7 \leftarrow +, b^2, t2^6$
 $t4^8 \leftarrow *, y^3, t3^7$
 $t5^5 \leftarrow -, y^3, z^4$
 $t6^6 \leftarrow *, t5^5, b^2$
 $t7^9 \leftarrow +, t4^8, t6^6$
 $a^{10} \leftarrow +, a^1, t7^9$

Верхние индексы у идентификаторов – номера соответствующих значений

1	id	ссылка в ТС		a
2	id	ссылка в ТС		b
3	id	ссылка в ТС		y
4	id	ссылка в ТС		z
5	-	3	4	t1, t5
6	*	5	2	t2, t6
7	+	2	6	t3
8	*	3	7	t4
9	+	8	6	t7
10	+	1	9	a
# значения	КОП	# операнда	# операнда	Присоединенные переменные
	Определение значения (сигнатура)			

9.1. Базовый алгоритм локальной нумерации значений

9.1.3 Алгоритм построения ТЗ

Алгоритм (на псевдокоде) построения ТЗ для базового блока B , содержащего n инструкций вида $t_i \leftarrow op_i, l_i, r_i$.

Функция $\#val(s)$ определяет номер значения, соответствующего сигнатуре $s = (op, \#val(l), \#val(r))$:

```
for each " $t_i \leftarrow op_i, l_i, r_i$ " do
     $s_i = (op_i, \#val(l_i), \#val(r_i))$ 
    if (ТЗ содержит  $s_j == s_i$ )
        then
            вернуть  $j$  в качестве значения  $\#val(s_i)$ 
    else
        завести в ТЗ новую строку  $ТЗ_k$ 
        записать сигнатуру  $s_i$  в строку  $ТЗ_k$ 
        вернуть  $k$  в качестве значения  $\#val(s_i)$ 
```

9.1. Базовый алгоритм локальной нумерации значений

9.1.3. Модификация подхода

- ◇ Чтобы ускорить алгоритм нумерации значений для отображения переменных, констант и вычисляемых значений (выражений) на их номера значений используется хэш-таблица.
- ◇ Для переменных и констант в качестве аргумента функции $\#Val(s)$ используются их имена по таблице символов.
- ◇ Для выражения вида $op, opnd_1, opnd_2$ сигнатура (аргумент функции $\#Val(s)$) имеет вид:

$$op, \#Val(opnd_1), \#Val(opnd_2)$$

где $\#Val(opnd_i)$ – номер значения операнда $opnd_i$,
а op – знак операции (например, +).

- ◇ В инструкциях присваивания и копирования
номер значения правой части становится номером значения левой части.

9.1. Базовый алгоритм локальной нумерации значений

9.1.3 Построение таблицы значений для базового блока

- ◇ Как уже упоминалось, базовый алгоритм локальной нумерации значений позволяет построить хеш-таблицу значений для каждого базового блока.

9.2. Нумерация значений в суперблоках

9.2.1 Постановка задачи глобальной нумерации значений

- ◇ Известно несколько подходов к построению алгоритма глобальной нумерации значений.
- ◇ Мы хотим получить **алгоритм глобальной нумерации значений**, распространив базовый алгоритм на всю анализируемую процедуру.
- ◇ Первым шагом будет построение таблицы нумерации значений в суперблоке.

9.2. Нумерация значений в суперблоках

9.2.1. Суперблоки (расширенные базовые блоки).

- ◇ **Определение.** *Суперблок* (или *расширенный базовый блок*) E определяется как множество базовых блоков $B_1, B_2, \dots, B_n$, где только у блока B_1 может быть несколько предшественников, а каждый из блоков $B_i, 2 \leq i \leq n$ имеет в суперблоке единственного предшественника.

Блоки $B_i \in E$ формируют дерево с корнем B_1 .

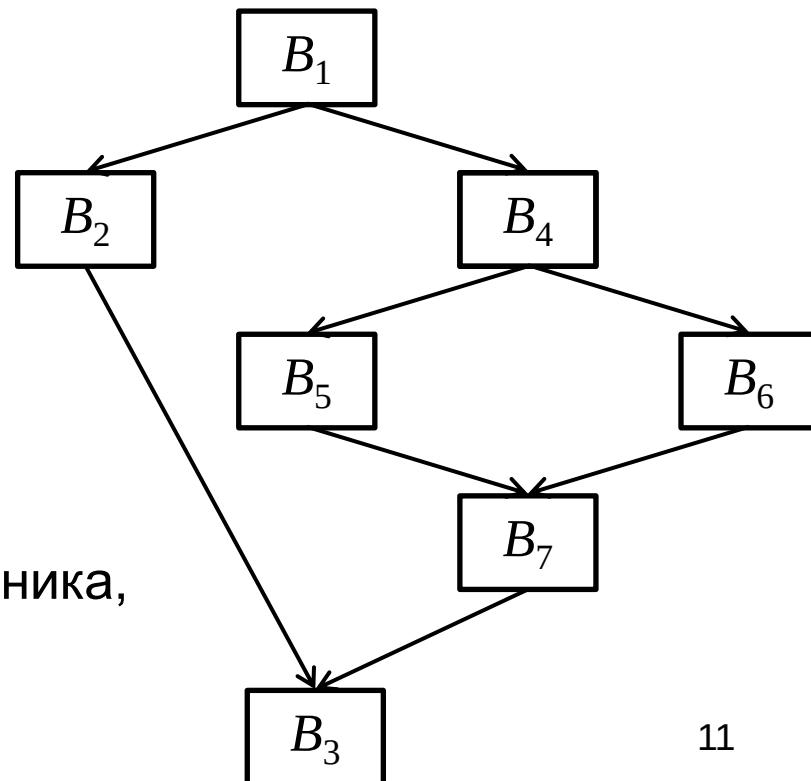
У суперблока E может быть несколько выходов на блоки (или суперблоки), не входящие в состав E .

- ◇ Алгоритм нумерации значений в суперблоках будем называть *расширенным алгоритмом локальной нумерации значений*.

9.2. Нумерация значений в суперблоках

9.2.1 Суперблоки. Пример.

B_1 $m \leftarrow +, a, b$ $n \leftarrow +, a, b$	B_2 $p \leftarrow +, c, d$ $r \leftarrow +, c, d$	B_3 $y \leftarrow +, a, b$ $z \leftarrow +, c, d$
B_4 $q \leftarrow +, a, b$ $r \leftarrow +, c, d$	B_5 $e \leftarrow +, b, 18$ $s \leftarrow +, a, b$ $u \leftarrow +, e, f$	B_6 $e \leftarrow +, b, 17$ $t \leftarrow +, c, d$ $u \leftarrow +, e, f$
B_7 $v \leftarrow +, a, b$ $w \leftarrow +, c, d$ $x \leftarrow +, e, f$		



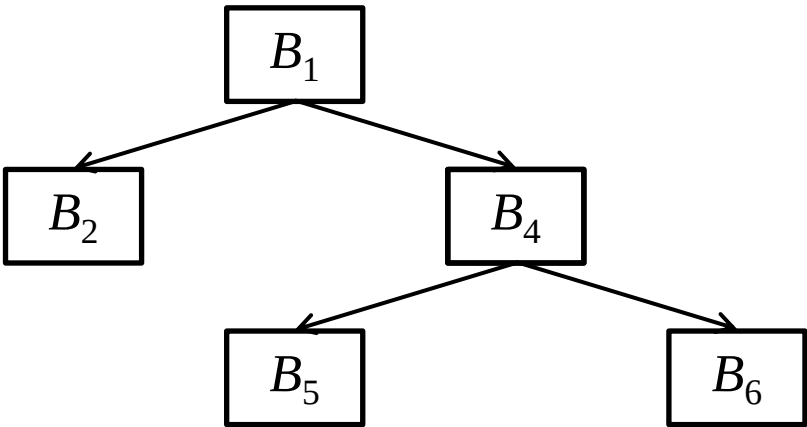
В коде, показанном на рисунке можно выделить три суперблока:

$\{B_1, B_2, B_4, B_5, B_6\}$, $\{B_7\}$ и $\{B_3\}$.

Блоки B_7 и B_3 имеют по два предшественника, поэтому их нельзя включить в суперблок больших размеров

9.2. Нумерация значений в суперблоках

9.3.1 Суперблоки. Пример.



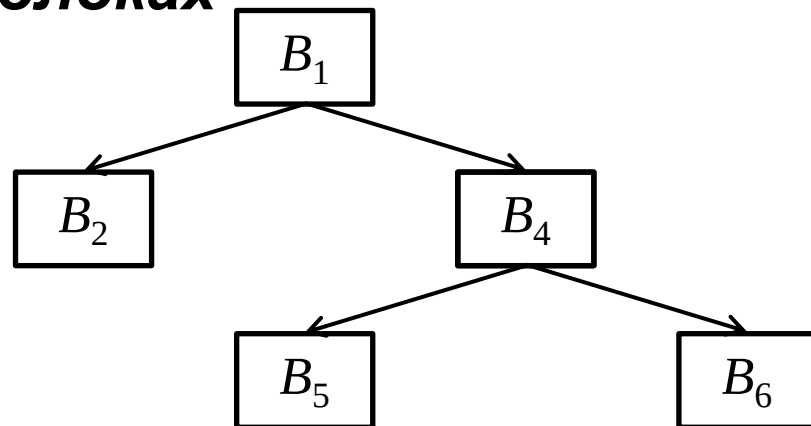
B_1 $m \leftarrow +, a, b$ $n \leftarrow +, a, b$	B_2 $p \leftarrow +, c, d$ $r \leftarrow +, c, d$	
B_4 $q \leftarrow +, a, b$ $r \leftarrow +, c, d$	B_5 $e \leftarrow +, b, 18$ $s \leftarrow +, a, b$ $u \leftarrow +, e, f$	B_6 $e \leftarrow +, b, 17$ $t \leftarrow +, c, d$ $u \leftarrow +, e, f$

В суперблоке $\{B_1, B_2, B_4, B_5, B_6\}$ можно выделить три пути:
 $\{B_1, B_2\}$, $\{B_1, B_4, B_5\}$ и $\{B_1, B_4, B_6\}$.

Пронумеруем значения вдоль каждого из этих путей.

9.3. Нумерация значений в суперблоках

9.3.1 Суперблоки. Пример.



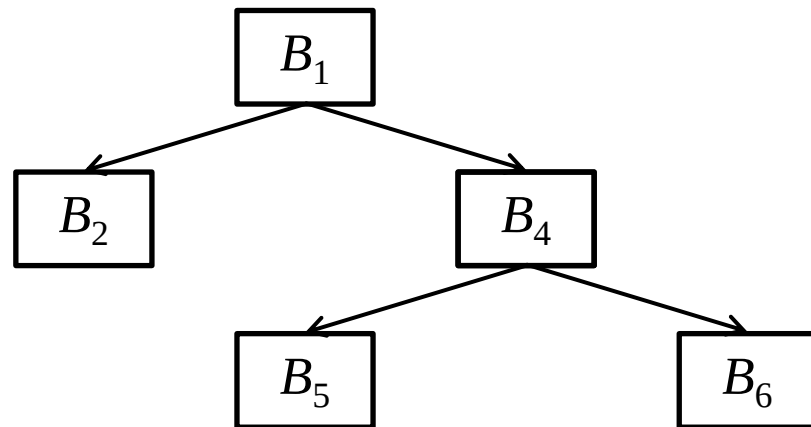
B_1 $m \leftarrow +, a, b$ $n \leftarrow +, a, b$	B_2 $p \leftarrow +, c, d$ $r \leftarrow +, c, d$	
B_4 $q \leftarrow +, a, b$ $r \leftarrow +, c, d$	B_5 $e \leftarrow +, b, 18$ $s \leftarrow +, a, b$ $u \leftarrow +, e, f$	B_6 $e \leftarrow +, b, 17$ $t \leftarrow +, c, d$ $u \leftarrow +, e, f$

а) Путь $\{B_1, B_4, B_5\}$:

- ◇ строится (хэш) таблица нумерации значений для блока B_1
- ◇ таблица нумерации значений для пути $\{B_1, B_4\}$ строится как продолжение таблицы для блока B_1
- ◇ таблица нумерации значений для пути $\{B_1, B_4, B_5\}$ строится как продолжение таблицы для пути $\{B_1, B_4\}$.
- ◇ путь $\{B_1, B_4, B_5\}$ рассматривается как единый базовый блок

9.2. Нумерация значений в суперблоках

9.2.1 Суперблоки. Пример.



B_1 $m \leftarrow +, a, b$ $n \leftarrow +, a, b$	B_2 $p \leftarrow +, c, d$ $r \leftarrow +, c, d$	
B_4 $q \leftarrow +, a, b$ $r \leftarrow +, c, d$	B_5 $e \leftarrow +, b, 18$ $s \leftarrow +, a, b$ $u \leftarrow +, e, f$	B_6 $e \leftarrow +, b, 17$ $t \leftarrow +, c, d$ $u \leftarrow +, e, f$

б) Путь $\{B_1, B_4, B_6\}$:

- ◇ ТЗ для пути $\{B_1, B_4, B_6\}$ строится как продолжение ТЗ для пути $\{B_1, B_4\}$. Путь $\{B_1, B_4, B_6\}$ можно рассматривать как единый базовый блок

в) Путь $\{B_1, B_2\}$:

- ◇ ТЗ для пути $\{B_1, B_2\}$ строится как продолжение ТЗ для блока B_1 .

9.2. Нумерация значений в суперблоках

9.2.2 Контекстные таблицы значений (КТЗ)

- ◇ ТЗ, строящиеся внутри суперблоков, являются *контекстными* в том смысле, что они учитывают уже построенные ТЗ своих доминаторов.
- ◇ **Оценка размера ТЗ для каждого контекста** (очевидная):
в рассматриваемом промежуточном представлении число имен не может более чем в три раза превышать число инструкций, так как в инструкции может использоваться не более трех имен. Это соображение позволяет сразу выделить достаточно памяти под каждую ТЗ, исключив возможность переполнения ТЗ.

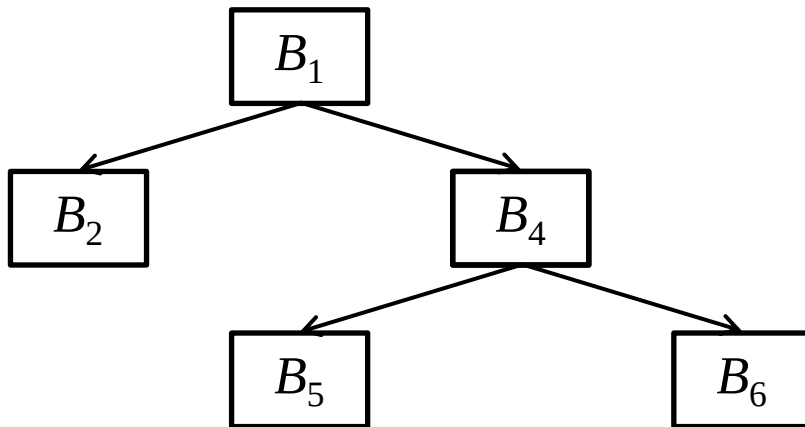
9.2. Нумерация значений в суперблоках

9.2.2 Контекстные таблицы значений (КТЗ)

◇ Обеспечивается это, естественно, с помощью стека:

В нашем примере сначала в стеке строится ТЗ для B_1 , потом – КТЗ для B_2 , как продолжение ТЗ для B_1 , потом выталкивается КТЗ для B_2 , и на ее месте строится КТЗ для $\{B_1, B_4\}$, которая потом превращается в КТЗ для $\{B_1, B_4, B_5\}$ и т.д., пока не будут обойдены все блоки.

Объединенные КТЗ образуют *текущий контекст*.



После того как будут обработаны все пути из суперблока $\{B_1, B_2, B_4, B_5, B_6\}$ будут обработаны блоки B_7 и B_3 . При их обработке **не удастся использовать** ТЗ других блоков, так как каждый из них достижим по двум путям, что приведет к путанице при объединении ТЗ.

9.2. Нумерация значений в суперблоках

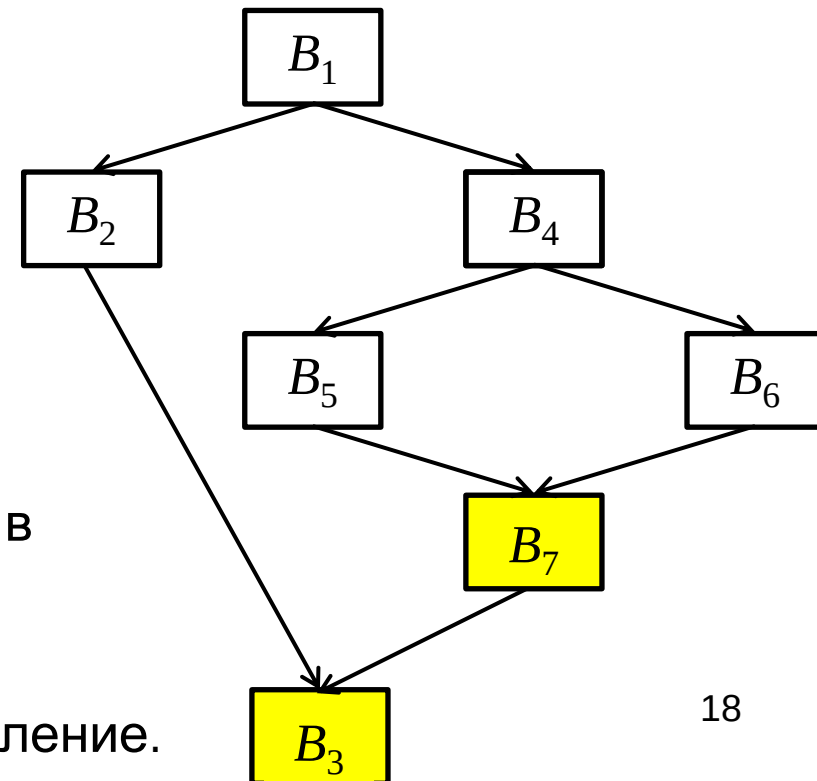
9.2.3 Затруднение

- ◇ Если в нескольких базовых блоках, входящих в состав суперблока (например, в блоках B_1 и B_4), определяются номера значений одной и той же переменной (используется одно и то же имя переменной) результат его определения в одном блоке (B_4) может попасть в контекст, связанный с другим блоком (B_1).
- ◇ В этом случае при удалении ТЗ блока B_4 из стэка в стэке могут сохраниться записи об этой переменной в ТЗ блока B_1 . Так как такую возможность необходимо учесть в алгоритме нумерации значений, алгоритм усложнится.
- ◇ Описанного затруднения легко избежать, если выполнять нумерацию значений для суперблоков в SSA-форме.

9.2. Нумерация значений в суперблоках

9.2.4 Контекстные таблицы значений и SSA-форма

B_1 $m_0 \leftarrow +, a_0, b_0$ $n_0 \leftarrow +, a_0, b_0$	B_2 $p_0 \leftarrow +, c_0, d_0$ $r_0 \leftarrow +, c_0, d_0$	B_3 $r_2 \leftarrow \phi(r_0, r_1)$ $y_0 \leftarrow +, a_0, b_0$ $z_0 \leftarrow +, c_0, d_0$
B_4 $q_0 \leftarrow +, a_0, b_0$ $r_1 \leftarrow +, c_0, d_0$	B_5 $e_0 \leftarrow +, b_0, 18$ $s_0 \leftarrow +, a_0, b_0$ $u_0 \leftarrow +, e_0, f_0$	B_6 $e_1 \leftarrow +, b_0, 17$ $t_0 \leftarrow +, c_0, d_0$ $u_1 \leftarrow +, e_1, f_0$
B_7 $e_2 \leftarrow \phi(e_0, e_1)$ $u_2 \leftarrow \phi(u_0, u_1)$ $v_0 \leftarrow +, a_0, b_0$ $w_0 \leftarrow +, c_0, d_0$ $x_0 \leftarrow +, e_2, f_0$		



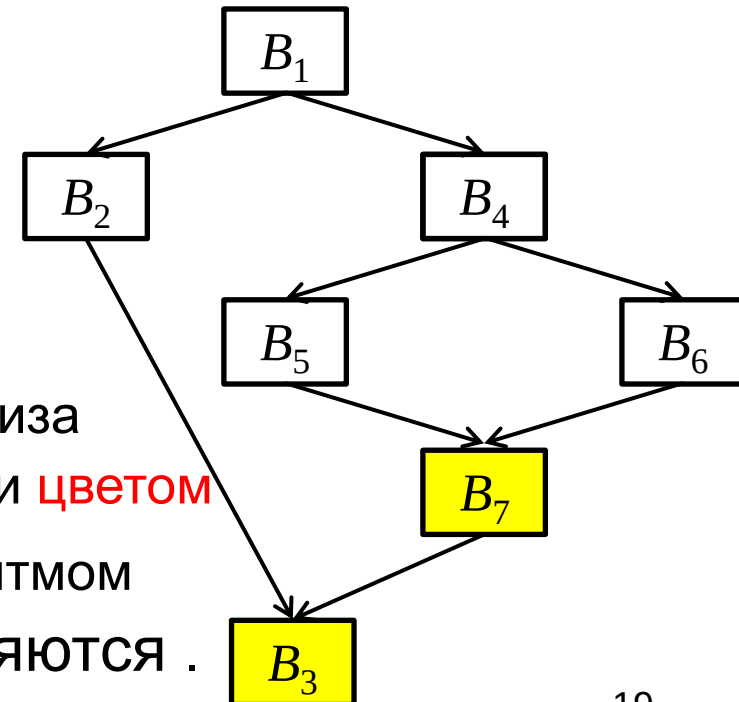
◇ SSA-форма обладает двумя важными свойствами:

- (1) каждое имя определяется только в одной инструкции
- (2) каждое использование значения ссылается только на одно определение.

9.2. Нумерация значений в суперблоках

9.2.4 Контекстные таблицы значений и SSA-форма

B_1 $m_0 \leftarrow +, a_0, b_0$ <u>$n_0 \leftarrow +, a_0, b_0$</u>	B_2 $p_0 \leftarrow +, c_0, d_0$ <u>$r_0 \leftarrow +, c_0, d_0$</u>	B_3 $r_2 \leftarrow \varphi(r_0, r_1)$ $y_0 \leftarrow +, a_0, b_0$ $z_0 \leftarrow +, c_0, d_0$
B_4 <u>$q_0 \leftarrow +, a_0, b_0$</u> $r_1 \leftarrow +, c_0, d_0$	B_5 $e_0 \leftarrow +, b_0, 18$ <u>$s_0 \leftarrow +, a_0, b_0 \cdot$</u> $u_0 \leftarrow +, e_0, f_0$	B_6 $e_1 \leftarrow +, b_0, 17$ <u>$t_0 \leftarrow +, c_0, d_0 \cdot$</u> $u_1 \leftarrow +, e_1, f_0$
B_7 $e_2 \leftarrow \varphi(e_0, e_1)$ $u_2 \leftarrow \varphi(u_0, u_1)$ $v_0 \leftarrow +, a_0, b_0$ $w_0 \leftarrow +, c_0, d_0$ $x_0 \leftarrow +, e_2, f_0$		

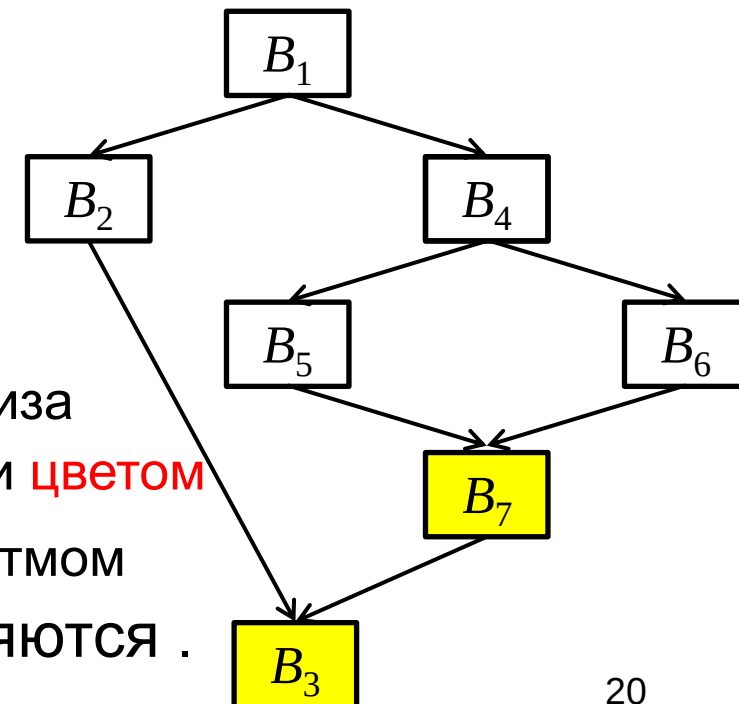


- ◇ Инструкции, удаляемые алгоритмом анализа суперблоков, выделены подчеркиванием и **цветом**
- ◇ Инструкции, помеченные знаком $\cdot$, алгоритмом локальной нумерации значений не удаляются.

9.2. Нумерация значений в суперблоках

9.2.4 Контекстные таблицы значений и SSA-форма

B_1 $m_0 \leftarrow +, a_0, b_0$ $n_0 \leftarrow m_0$	B_2 $p_0 \leftarrow +, c_0, d_0$ $r_0 \leftarrow p_0$	B_3 $r_2 \leftarrow \varphi(r_0, r_1)$ $y_0 \leftarrow +, a_0, b_0$ $z_0 \leftarrow +, c_0, d_0$
B_4 $q_0 \leftarrow m_0$ $r_1 \leftarrow +, c_0, d_0$	B_5 $e_0 \leftarrow +, b_0, 18$ $s_0 \leftarrow m_0 \bullet$ $u_0 \leftarrow +, e_0, f_0$	B_6 $e_1 \leftarrow +, b_0, 17$ $t_0 \leftarrow r_1 \bullet$ $u_1 \leftarrow +, e_1, f_0$
B_7 $e_2 \leftarrow \varphi(e_0, e_1)$ $u_2 \leftarrow \varphi(u_0, u_1)$ $v_0 \leftarrow +, a_0, b_0$ $w_0 \leftarrow +, c_0, d_0$ $x_0 \leftarrow +, e_2, f_0$		



- ◇ Инструкции, удаляемые алгоритмом анализа суперблоков, выделены подчеркиванием и **цветом**
- ◇ Инструкции, помеченные знаком $\bullet$, алгоритмом локальной нумерации значений не удаляются.

9.2. Нумерация значений в суперблоках

9.2.5 Оценка расширенного алгоритма локальной нумерации значений

- ◇ Расширенный алгоритм локальной нумерации значений отлично работает внутри суперблока. Однако, при переходе от одного суперблока к другому он пропускает некоторые избыточности.
- ◇ В рассматриваемом примере:
 - ◇ B_3 и B_7 составляют отдельные суперблоки, поэтому вычисления $a_0 + b_0$ и $c_0 + d_0$ в блоках B_7 и B_3 не рассматриваются алгоритмом вместе с вычислениями в других блоках и не распознаются как избыточные, хотя на самом деле они избыточны.
 - ◇ Выражение $e + f$ является доступным в блоке B_7 :
 $e + f$ вычисляется на любом пути, достигающем блока B_7 , и ни e , ни f не переопределяются до вычисления в B_7 .
Потому оно избыточно и будет исключено, хотя это неверно.
Причина: расширенный алгоритм не может установить, что e имеет разные номера значений в блоках D и E .

9.2. Нумерация значений в суперблоках

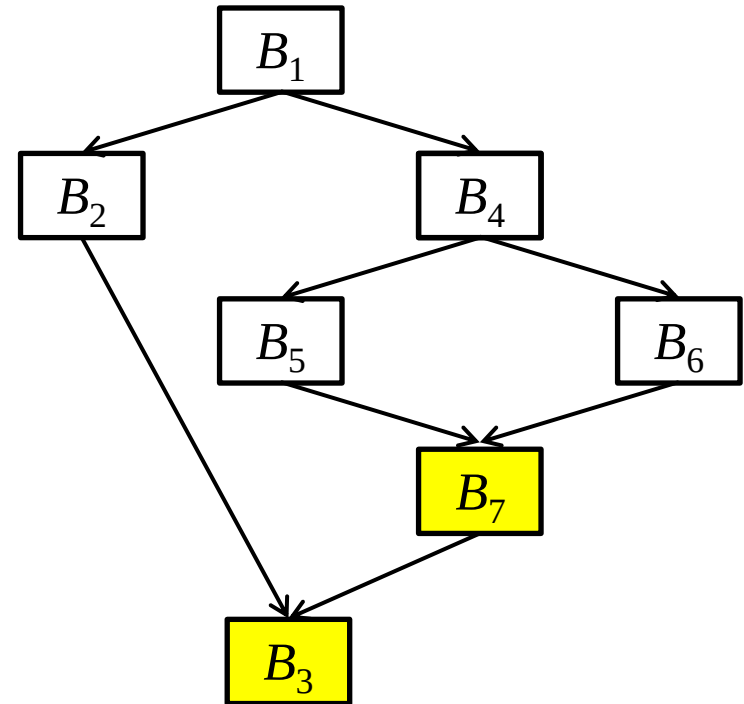
9.2.5 Оценка расширенного алгоритма локальной нумерации значений

- ◇ Несмотря на перечисленные недостатки, расширенный алгоритм заслуживает применения:
он позволяет найти намного больше избыточностей, чем локальный алгоритм, за минимальные дополнительные накладные расходы.
- ◇ Использование механизма контекстно-ориентированных ТЗ позволяет существенно сократить накладные расходы на работу с суперблоками.

9.3 Глобальная нумерация значений

9.3.1 Необходимость обработки точек сбора

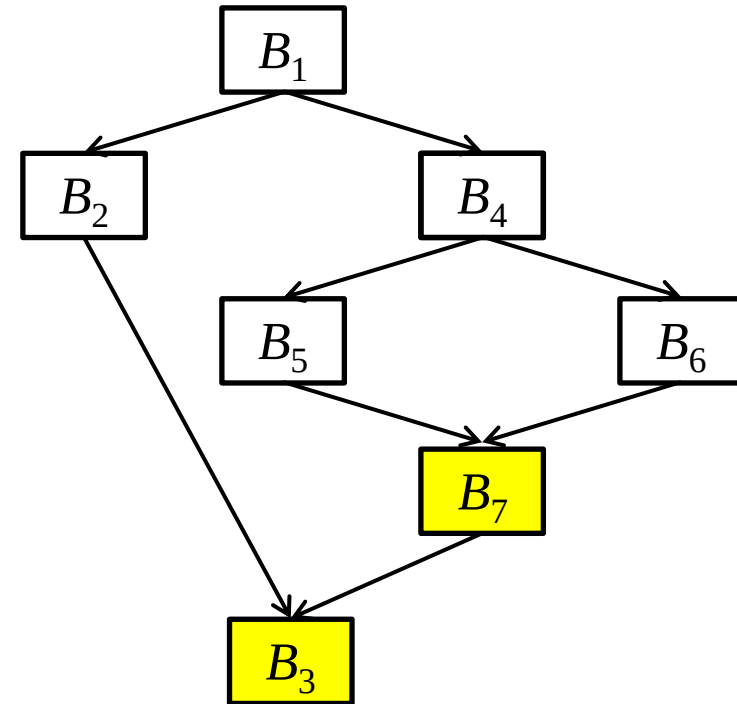
- ◇ Расширенный алгоритм локальной нумерации значений пропускает некоторые избыточности, так как он удаляет **всю** таблицу значений, когда достигает блока, имеющего в ГПУ более одного предшественника (в нашем примере это блоки B_7 и B_3).
- ◇ Нужен алгоритм, который может распространять информацию через точки сбора графа потока (в нашем примере это переходы от B_5 и B_6 к B_7 , и от B_2 и B_7 к B_3).



9.3 Глобальная нумерация значений

9.3.2 Проблема точек сбора

- ◇ Для нумерации значений в блоке B_7 , расширенный алгоритм не может использовать ТЗ для пути $\{B_1, B_4, B_5\}$, так как при этом не учитывается путь от B_6 к B_7 .
- ◇ Алгоритм не может использовать ТЗ для пути $\{B_1, B_4, B_6\}$, так как при этом не учитывается путь от B_5 к B_7 .
- ◇ Алгоритм не может слить ТЗ для B_5 и B_6 , так как операция слияния унифицирует номера значений, полученные вдоль непересекающихся путей. При унификации не учитывается, что, например, вычисления $e + f$ в B_5 и B_6 могут иметь разные номера значений.



9.3 Глобальная нумерация значений

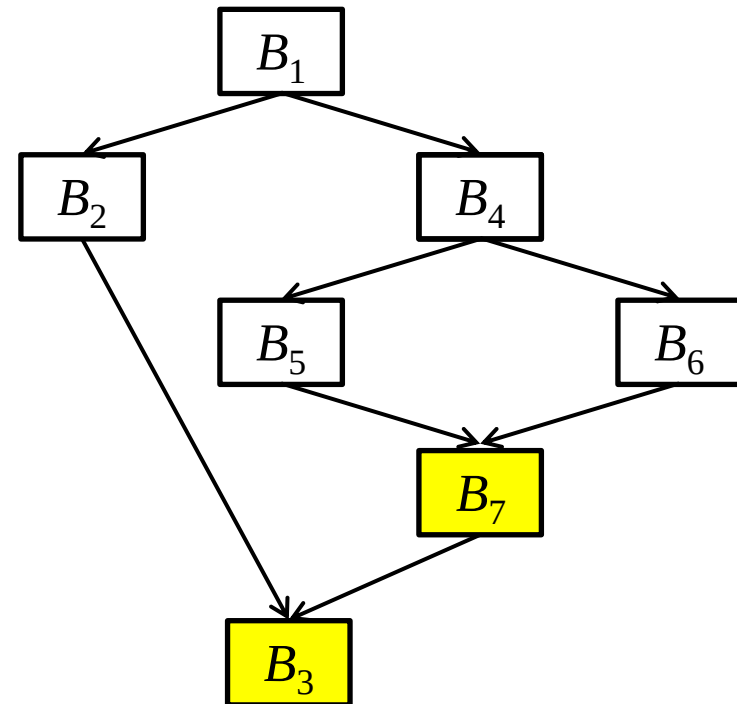
9.3.3 Обработка точек сбора

- ◇ ТЗ, которую алгоритм может использовать для блока B_7 , существует: оба пути, достигающие B_7 ($\{B_1, B_4, B_5\}$ и $\{B_1, B_4, B_6\}$), имеют общее начало $\{B_1, B_4\}$.

Следовательно, алгоритм может использовать ТЗ для B_4 (она содержит в себе таблицу для B_1) в качестве начального состояния ТЗ для B_7 .

Легко видеть, что $B_4 = \text{Idom}(B_7)$.

Следовательно для глобальной нумерации значений можно использовать **дерево доминаторов**, визуализирующее отношение Idom .



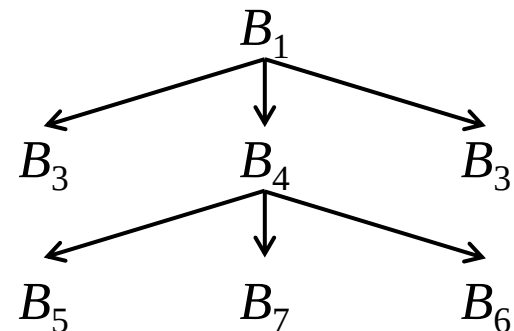
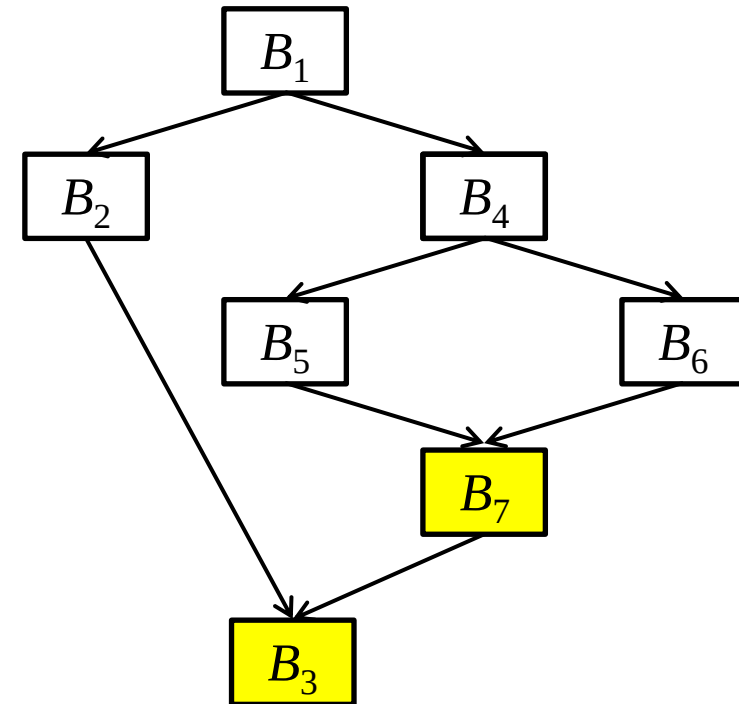
9.3 Глобальная нумерация значений

9.3.3 Обработка точек сбора

- ◇ ТЗ, которую алгоритм может использовать для блока B_7 , существует: оба пути, достигающие B_7 ($\{B_1, B_4, B_5\}$ и $\{B_1, B_4, B_6\}$), имеют общее начало $\{B_1, B_4\}$.

Следовательно, алгоритм может использовать ТЗ для B_4 (она содержит в себе таблицу для B_1) в качестве начального состояния ТЗ для B_7 .

Легко видеть, что $B_4 = \text{Idom}(B_7)$. Следовательно для глобальной нумерации значений можно использовать **дерево доминаторов**, визуализирующее отношение *Idom*.



Дерево доминаторов

9.4 Алгоритм глобальной нумерации значений

9.4.1 Вводные замечания

- ◇ Алгоритм *DBGVN* (*Dominator Based Global Value Numbering*) выполняет нумерацию значений процедуры с помощью рекурсивного обхода ее дерева доминаторов.
- ◇ ТЗ каждого базового блока инициализируется информацией, полученной при нумерации значений его родителя по дереву доминаторов.
- ◇ Для упрощения реализации алгоритма, *SSA*-имя первого вхождения выражения (на данном пути по дереву доминаторов) становится его номером значения. Это позволяет обойтись без массива имен, так как каждый номер значения – это аналог *SSA*-имени.
- ◇ Когда обнаруживается избыточное вычисление выражения, компилятор удаляет операцию, и заменяет все использования указанного *SSA*-имени на номер значения этого выражения.

9.4 Алгоритм глобальной нумерации значений

9.4.1 Вводные замечания

- ◇ Замена *SSA*-имени на номер значения вычисляющего его выражения допустима в следующих двух ситуациях:
 - ◇ Номер значения может заместить избыточное вычисление выражения в любом блоке, доминатором которого является блок, в котором в первый раз вычисляется это выражение.
 - ◇ Номер значения может заместить избыточное вычисление, результат которого является параметром φ -узла, входящего в состав границы доминирования блока, в котором в первый раз вычисляется это выражение.
- ◇ **Обработка φ -функций.** Прежде, чем компилятор сможет анализировать φ -функцию в блоке, он должен присвоить номера значений всем ее входам. Это возможно не всегда.
В частности, любой вход φ -функции, значение которого приходит по обратному ребру (относительно дерева доминаторов) не может иметь номера значения.

9.4 Алгоритм глобальной нумерации значений

9.4.2 Обработка φ -функций

- ◇ Следующие два условия гарантируют, что при попадании алгоритма в блок всем параметрам φ -функций этого блока уже присвоены номера значений:
 1. Рекурсивная процедура *DBGVN* обходит дерево доминаторов по ширине. Это гарантирует, что все предшественники блока будут обработаны до самого блока.
 2. Блок не имеет обратных входных ребер.
- ◇ Если φ -функция бессмысленна или избыточна, она должна быть удалена:
 - ◇ φ -функция *бессмысленна* если все ее входы имеют одинаковый номер значения. При удалении бессмысленной φ -функции ссылки на ее результат заменяются номером значения ее входов.
 - ◇ φ -функция *избыточна*, если она вычисляет то же самое значение, что и другая φ -функция в том же блоке

9.4 Алгоритм глобальной нумерации значений

9.4.3 Рекурсивный алгоритм глобальной нумерации значений

- ◇ **Вход:** (1) граф потока управления $\langle N, E \rangle$, дерево доминаторов DT
(2) множество Val значений переменных, констант и выражений
- ◇ **Выход:** отображение $VN: Val \rightarrow N \cup \{0\}$ (N – множество натуральных чисел), ставящее в соответствие каждому значению его номер: натуральное число или 0.
- ◇ **Метод:** Применить к корню DT рекурсивную процедуру $DBGVN$ (*Dominator Based Global Value Numbering*)

9.4 Алгоритм глобальной нумерации значений

9.4.4 Рекурсивная процедура *DBGVN* (на псевдокоде)

(1) Обработка φ -функций

procedure *DBGVN* (**Block** *B*)

 Отметить начало новой области имен

 || Обработка φ -функций

for each *p* $\in$ *B*, где *p* – φ -функция вида “ $n \leftarrow \varphi(\dots)$ ”

if *p* бессмысленна или избыточна

 поместить номер значения *p* в *VN*[*n*]

 удалить *p*

else

VN[*n*] $\leftarrow$ *n*;

 добавить *p* в ТЗ

9.4 Алгоритм глобальной нумерации значений

9.4.4 Рекурсивная процедура *DBGVN* (на псевдокоде)

(2) Обработка остальных инструкций

procedure *DBGVN*(**Block** *B*)

for each $a \in B$ где a – присваивание вида “ $x \leftarrow op, y, z$ ”

заменить y на $VN[y]$ и z на $VN[z]$

$expr \leftarrow op, y, z$ || $expr$ – вход в ТЗ

if $expr$ может быть упрощено до $expr'$

 Заменить a на “ $x \leftarrow expr'$ ”

$expr \leftarrow expr'$

if $expr$ имеется в ТЗ с номером v

$VN[x] \leftarrow v$

 Удалить a

else Добавить $expr$ в ТЗ с номером x $VN[x] \leftarrow x$ 32

9.4 Алгоритм глобальной нумерации значений

9.4.4 Рекурсивная процедура *DBGVN* (на псевдокоде)

(3) Окончание обработки блока B и переход к обработке его дочерних блоков (по дереву доминаторов)

procedure *DBGVN*(Block B)

for each $s \in Succ(B)$

 скорректировать входы φ -функций в S

for each дочернего блока c узла B по дереву доминаторов

$DBGVN(c)$

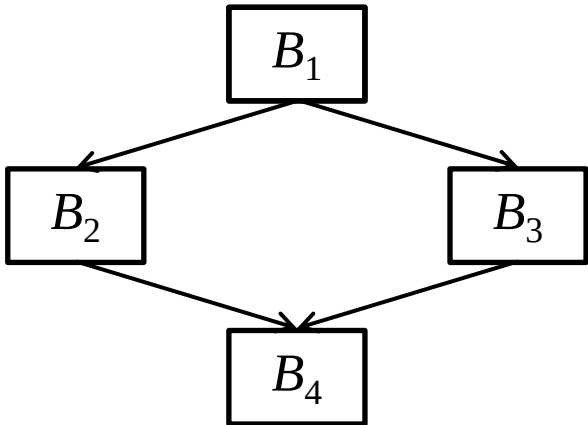
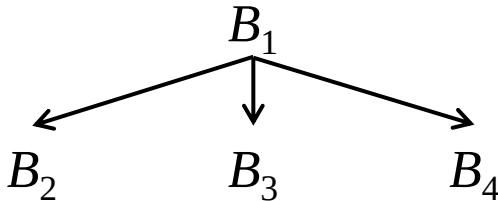
 || рекурсивный вызов

 Очистить ТЗ при выходе из области (стэк)

9.4 Алгоритм глобальной нумерации значений

9.4.5 Пример применения алгоритма

- ◇ Применим алгоритм к фрагменту кода на рисунке. Порядок обработки определяется деревом доминаторов



- ◇ Обработка **блока B_1** .
Переменным u_0 , v_0 , и w_0 в качестве номеров значений будут присвоены их *SSA*-имена $\langle u_0 \rangle$, $\langle v_0 \rangle$, и $\langle w_0 \rangle$

v	$\#val(v)$
u_0	$\langle u_0 \rangle$
v_0	$\langle v_0 \rangle$
w_0	$\langle w_0 \rangle$
x_0	
y_0	
u_1	
x_1	
y_1	
u_2	
x_2	
y_2	
z_0	
u_3	

B_1 $u_0 \leftarrow a_0 + b_0$ $v_0 \leftarrow c_0 + d_0$ $w_0 \leftarrow e_0 + f_0$	B_2 $x_0 \leftarrow c_0 + d_0$ $y_0 \leftarrow c_0 + d_0$
B_3 $u_1 \leftarrow a_0 + b_0$ $x_1 \leftarrow e_0 + f_0$ $y_1 \leftarrow e_0 + f_0$	B_4 $u_2 \leftarrow \phi(u_0, u_1)$ $x_2 \leftarrow \phi(x_0, x_1)$ $y_2 \leftarrow \phi(y_0, y_1)$ $z_0 \leftarrow u_2 + y_2$ $u_3 \leftarrow a_0 + b_0$

После обработки B_1

9.4 Алгоритм глобальной нумерации значений

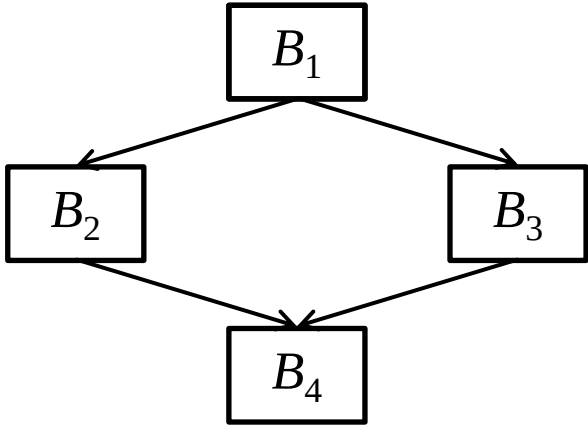
9.4.5 Пример применения алгоритма

◇ Обработка блока B_2 .

Выражение $c_0 + d_0$ определено в блоке B_1 , доминаторе B_2 .

Поэтому в B_2 можно удалить оба присваивания, присвоив x_0 и y_0 номер значения $\langle v_0 \rangle$.
Необходимо также подготовиться к обработке блока B_4 , заменив параметры ϕ -функций u_0 , x_0 , и y_0 номерами их значений $\langle u_0 \rangle$, $\langle v_0 \rangle$, и $\langle v_0 \rangle$.

v	$\#val(v)$
u_0	$\langle u_0 \rangle$
v_0	$\langle v_0 \rangle$
w_0	$\langle w_0 \rangle$
x_0	$\langle v_0 \rangle$
y_0	$\langle v_0 \rangle$
u_1	
x_1	
y_1	
u_2	
x_2	
y_2	
z_0	
u_3	



B_1 $u_0 \leftarrow a_0 + b_0$ $v_0 \leftarrow c_0 + d_0$ $w_0 \leftarrow e_0 + f_0$	B_2
B_3 $u_1 \leftarrow a_0 + b_0$ $x_1 \leftarrow e_0 + f_0$ $y_1 \leftarrow e_0 + f_0$	B_4 $u_2 \leftarrow \phi(\langle u_0 \rangle, u_1)$ $x_2 \leftarrow \phi(\langle v_0 \rangle, x_1)$ $y_2 \leftarrow \phi(\langle v_0 \rangle, y_1)$ $z_0 \leftarrow u_2 + y_2$ $u_3 \leftarrow a_0 + b_0$

После обработки B_2

9.4 Алгоритм глобальной нумерации значений

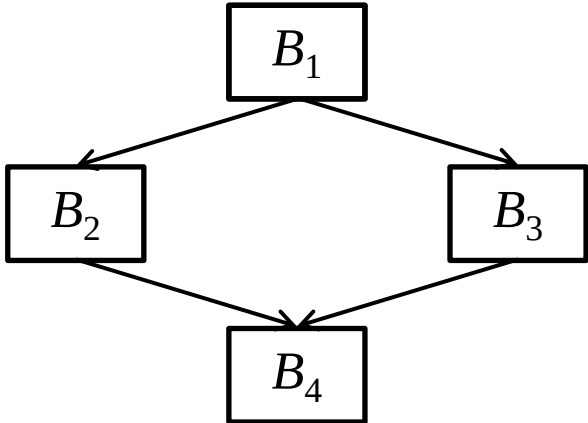
9.4.5 Пример применения алгоритма

◇ Обработка блока B_3 .

Все три выражения в правых частях уже вычислены в B_1 , доминаторе B_3

Поэтому переменным u_1 , x_1 и y_1 присваиваются номера значений $\langle u_0 \rangle$, $\langle w_0 \rangle$ и $\langle w_0 \rangle$ соответственно и удалить присваивания.

В заключение обработки B_3 , необходимо заполнить вторые параметры ϕ -функций из B_4 номерами значений $\langle u_0 \rangle$, $\langle w_0 \rangle$ и $\langle w_0 \rangle$.



v	$\#val(v)$
u_0	$\langle u_0 \rangle$
v_0	$\langle v_0 \rangle$
w_0	$\langle w_0 \rangle$
x_0	$\langle v_0 \rangle$
y_0	$\langle v_0 \rangle$
u_1	$\langle u_0 \rangle$
x_1	$\langle w_0 \rangle$
y_1	$\langle w_0 \rangle$
u_2	
x_2	
y_2	
z_0	
u_3	

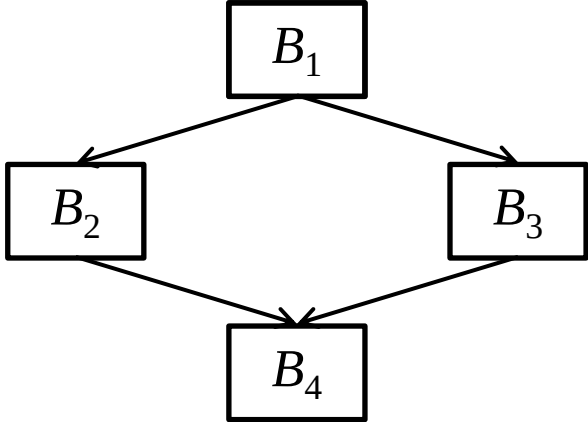
B_1 $u_0 \leftarrow a_0 + b_0$ $v_0 \leftarrow c_0 + d_0$ $w_0 \leftarrow e_0 + f_0$	B_2
B_3	B_4 $u_2 \leftarrow \phi(\langle u_0 \rangle, \langle u_0 \rangle)$ $x_2 \leftarrow \phi(\langle v_0 \rangle, \langle w_0 \rangle)$ $y_2 \leftarrow \phi(\langle v_0 \rangle, \langle w_0 \rangle)$ $z_0 \leftarrow u_2 + y_2$ $u_3 \leftarrow a_0 + b_0$

После обработки B_3

9.4 Алгоритм глобальной нумерации значений

9.4.5 Пример применения алгоритма

- ◇ Обработка блока B_4 .
- ◇ Сначала исследуются ϕ -функции. Это возможно, так как выполнены условия :
 - (1) все дети B_1 по дереву доминаторов (B_2, B_3) уже обработаны;
 - (2) в B_4 не входит обратных дуг.
- ◇ ϕ -функция, определяющая u_2 , **бессмысленна**: оба ее параметра равны $\langle u_0 \rangle$.
 ϕ -функция исключается, а u_2 присваивается номер значения $\langle u_0 \rangle$.



v	$\#val(v)$
u_0	$\langle u_0 \rangle$
v_0	$\langle v_0 \rangle$
w_0	$\langle w_0 \rangle$
x_0	$\langle v_0 \rangle$
y_0	$\langle v_0 \rangle$
u_1	$\langle u_0 \rangle$
x_1	$\langle w_0 \rangle$
y_1	$\langle w_0 \rangle$
u_2	$\langle u_0 \rangle$
x_2	
y_2	
z_0	
u_3	

B_1 $u_0 \leftarrow a_0 + b_0$ $v_0 \leftarrow c_0 + d_0$ $w_0 \leftarrow e_0 + f_0$	B_2
B_3	B_4 $x_2 \leftarrow \phi(\langle v_0 \rangle, \langle w_0 \rangle)$ $y_2 \leftarrow \phi(\langle v_0 \rangle, \langle w_0 \rangle)$ $z_0 \leftarrow u_2 + y_2$ $u_3 \leftarrow a_0 + b_0$

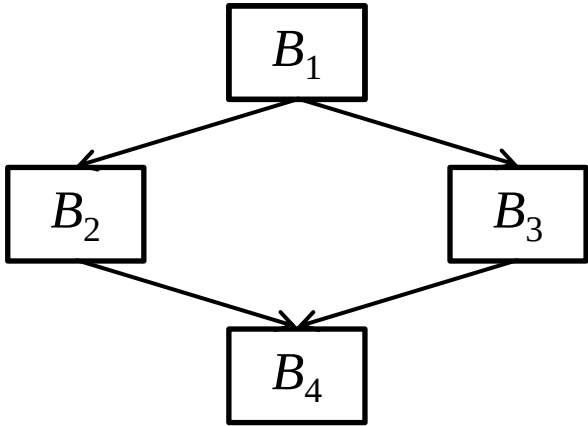
После обработки первой ϕ -функции

9.4 Алгоритм глобальной нумерации значений

9.4.5 Пример применения алгоритма

- ◇ Обработка блока B_4 .
- ◇ Вторая φ -функция имеет параметры v_0 и w_0 . Это первое вхождение φ -функции с такими параметрами. Ее значению x_2 в качестве номера значения присваивается ее *SSA-имя*.
- ◇ φ -функция, определяющая y_2 **избыточна**, так как y_2 равно x_2 . Поэтому эта φ -функция исключается, а y_2 присваивается номер значения $\langle x_2 \rangle$.

v	$\#val(v)$
u_0	$\langle u_0 \rangle$
v_0	$\langle v_0 \rangle$
w_0	$\langle w_0 \rangle$
x_0	$\langle v_0 \rangle$
y_0	$\langle v_0 \rangle$
u_1	$\langle u_0 \rangle$
x_1	$\langle w_0 \rangle$
y_1	$\langle w_0 \rangle$
u_2	$\langle u_0 \rangle$
x_2	$\langle u_0 \rangle$
y_2	$\langle x_2 \rangle$
z_0	
u_3	



B_1 $u_0 \leftarrow a_0 + b_0$ $v_0 \leftarrow c_0 + d_0$ $w_0 \leftarrow e_0 + f_0$	B_2
B_3	B_4 $x_2 \leftarrow \varphi(\langle v_0 \rangle, \langle w_0 \rangle)$ $z_0 \leftarrow u_2 + y_2$ $u_3 \leftarrow a_0 + b_0$

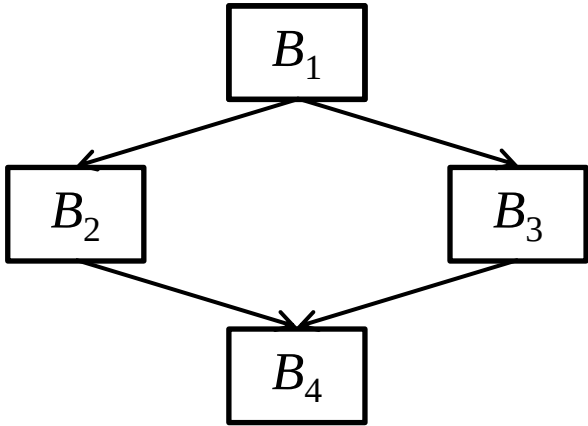
После обработки φ -функций

9.4 Алгоритм глобальной нумерации значений

9.4.5 Пример применения алгоритма

- ◇ Обработка блока B_4 .
- ◇ Обработка присваиваний.
- ◇ После подстановки вместо каждого операнда номера его значения для правой части присваивания z_0 получится номер значения $\# = \langle u_0 \rangle + \langle x_2 \rangle$.
- ◇ Присваивание u_3 избыточно (номер значения его правой части такой же, как у правой части присваивания u_0 в блоке B_1 . Поэтому это присваивание исключается, а u_3 присваивается номер значения $\langle u_0 \rangle$.

v	$\#val(v)$
u_0	$\langle u_0 \rangle$
v_0	$\langle v_0 \rangle$
w_0	$\langle w_0 \rangle$
x_0	$\langle v_0 \rangle$
y_0	$\langle v_0 \rangle$
u_1	$\langle u_0 \rangle$
x_1	$\langle w_0 \rangle$
y_1	$\langle w_0 \rangle$
u_2	$\langle u_0 \rangle$
x_2	$\langle u_0 \rangle$
y_2	$\langle x_2 \rangle$
z_0	$\#$
u_3	$\langle u_0 \rangle$



B_1 $u_0 \leftarrow a_0 + b_0$ $v_0 \leftarrow c_0 + d_0$ $w_0 \leftarrow e_0 + f_0$	B_2
B_3	B_4 $x_2 \leftarrow \phi(x_0, x_1)$ $z_0 \leftarrow u_2 + y_2$

После обработки B_4

9.4 Алгоритм глобальной нумерации значений

9.4.5 Заключительные замечания

- ◇ Алгоритм глобальной нумерации значений был описан для программ, уже переведенных в *SSA*-форму. Однако, **можно включить нумерацию значений в процесс конструирования *SSA*.**
- ◇ В результате включения нумерации значений в процесс конструирования *SSA* повышается производительность оптимизатора, так как
 - ◇ сокращается объем выполняемой работы
 - ◇ уменьшается размер *SSA*-формы программы.
- ◇ Алгоритм глобальной нумерации значений объединяется с алгоритмом переименования переменных при построении *SSA*-формы.
В данном курсе этот объединенный алгоритм не рассматривается.

10. Анализ потока данных на основе областей

10.1 Области графа потока управления

10.1.1. Определение области

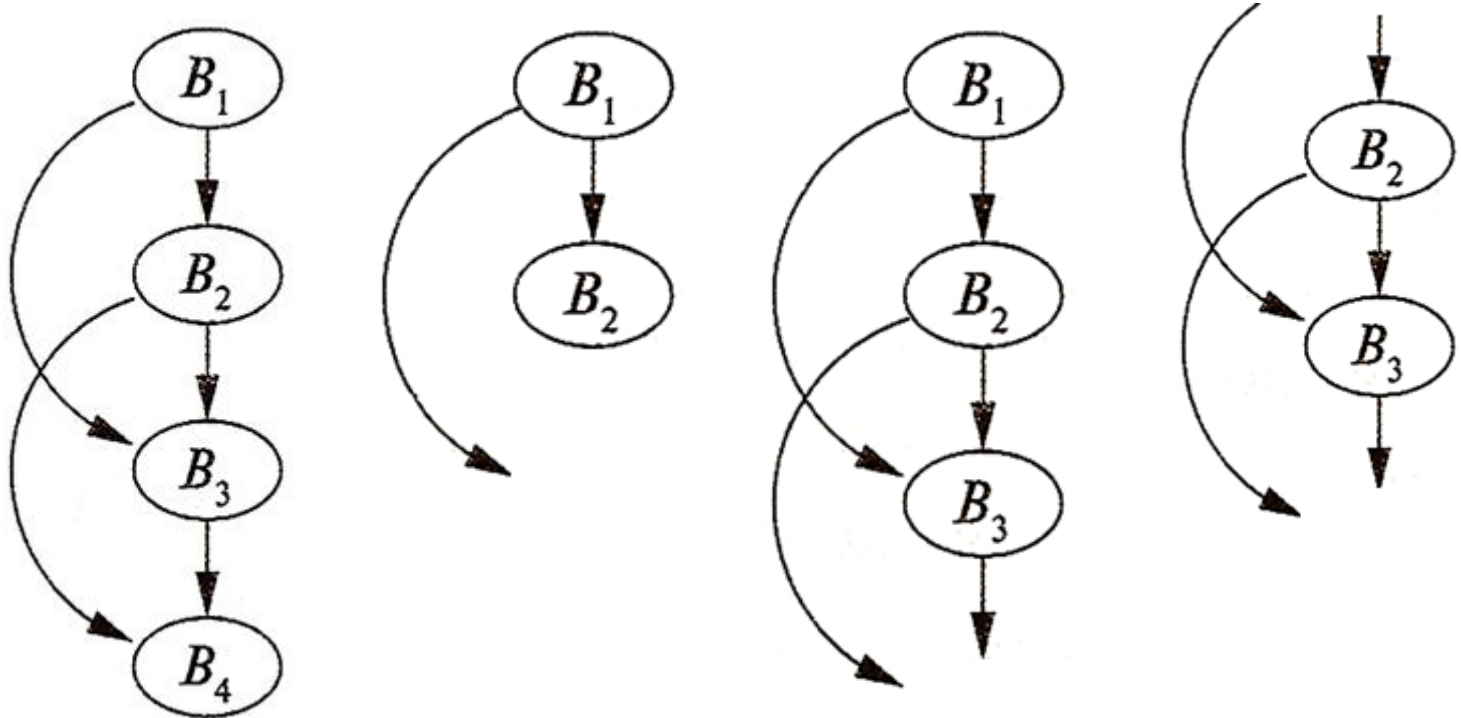
◇ **Определение.** *Областью* графа потока называется его подграф $R = \langle N_R, E_R \rangle$ такой, что

1. существует узел $h \in N_R$, доминирующий над всеми узлами в R ; этот узел называется *заголовком области* R ;
2. если из некоторого узла r_2 графа потока управления можно достичь узла $r_1 \in R$, минуя заголовок h , то и $r_2 \in R$;
3. множество E_R включает все ребра графа потока управления между любыми узлами $r_1, r_2 \in R$, за исключением некоторых ребер, входящих в заголовок h ;
4. **единственным входом** в область является ее заголовок.

10.1 Области графа потока управления

10.1.1. Определение области

◇ Пример 1.



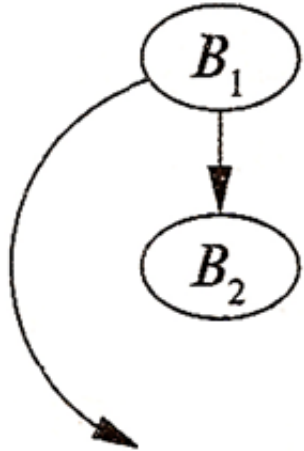
10.1 Области графа потока управления

10.1.1. Определение области

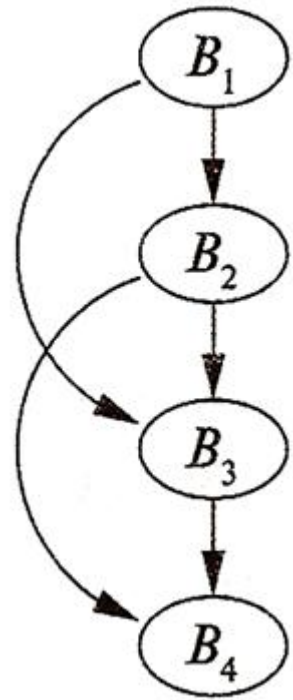
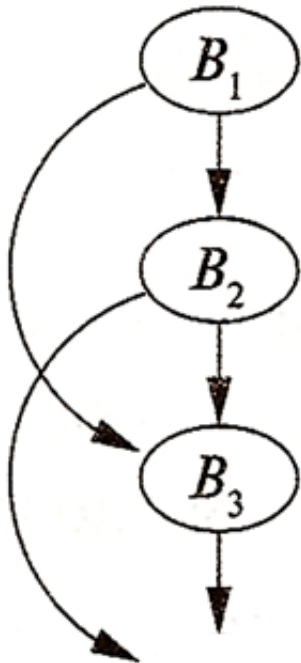
◇ Примеры.

7. Рассмотрим граф на рисунке справа

1) узлы B_1 и B_2 вместе с ребром $B_1 \rightarrow B_2$ образуют область с заголовком B_1



2) узлы B_1 , B_2 и B_3 и ребра $B_1 \rightarrow B_2$, $B_2 \rightarrow B_3$, $B_1 \rightarrow B_3$ образуют область с заголовком B_1



10.1 Области графа потока управления

10.1.1. Определение области

◇ Примеры.

1. Рассмотрим граф на верхнем рисунке

3) подграф $R = \langle \{B_2, B_3\}, \{B_2 \rightarrow B_3\} \rangle$ **область не образует**, так как управление может попасть в него и через B_2 , и через B_3 :

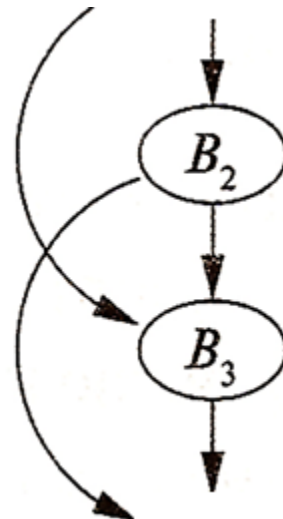
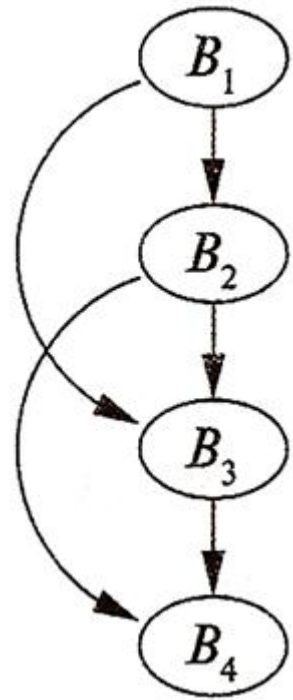
B_2 не является доминатором B_3 ,

B_3 не является доминатором B_2

и, следовательно, условие 1 определения 10.1.1

(**существует узел $h \in N_R$, доминирующий над всеми узлами в R ;**)

не выполняется (нижний рисунок)



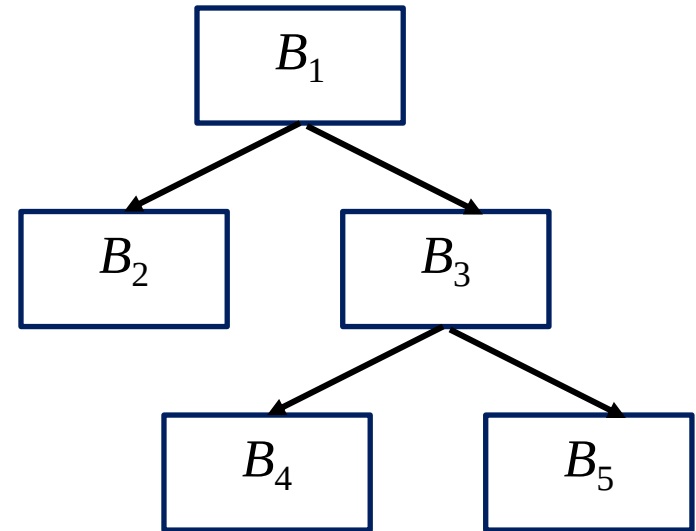
10.1 Области графа потока управления

10.1.1. Определение области

◇ Пример 2.

Суперблок (рассматривался, когда изучался метод глобальной нумерации значений) – пример области. В частности граф на рисунке – область с заголовком B_1 .

Узел B_1 доминирует над остальными узлами

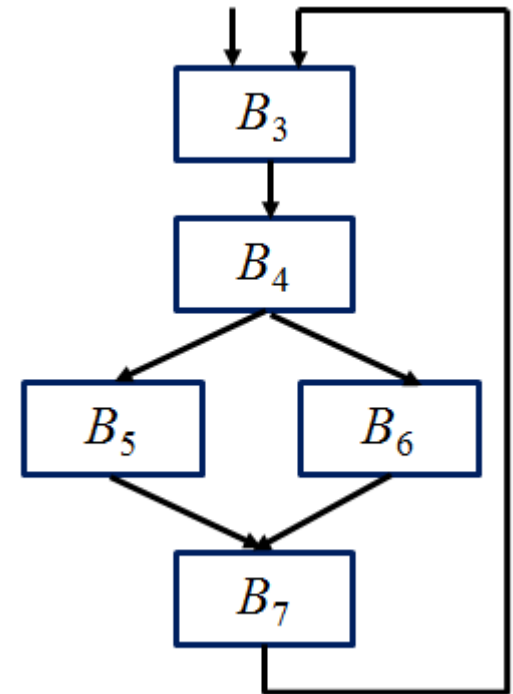


10.1 Области графа потока управления

10.1.1. Определение области

◇ Пример 3.

Граф на рисунке – область с заголовком B_3
(естественный цикл).



10.1 Структурный анализ графа потока управления

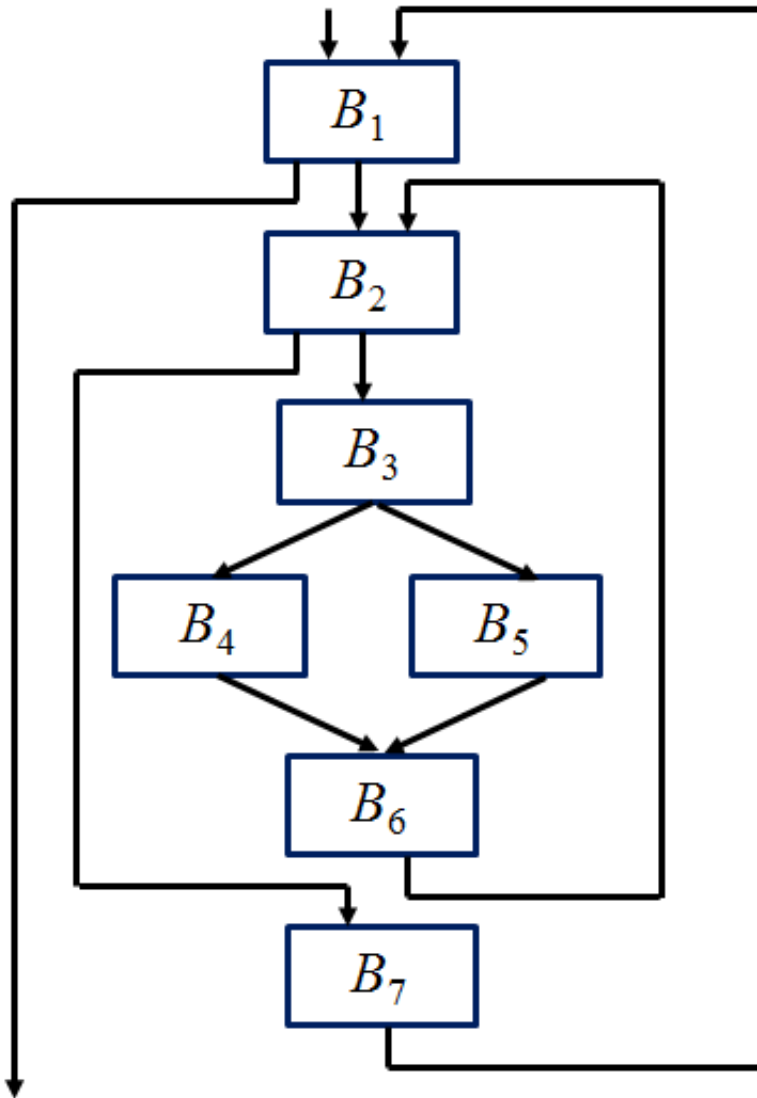
10.1.2. Классификация областей

◇ Определения:

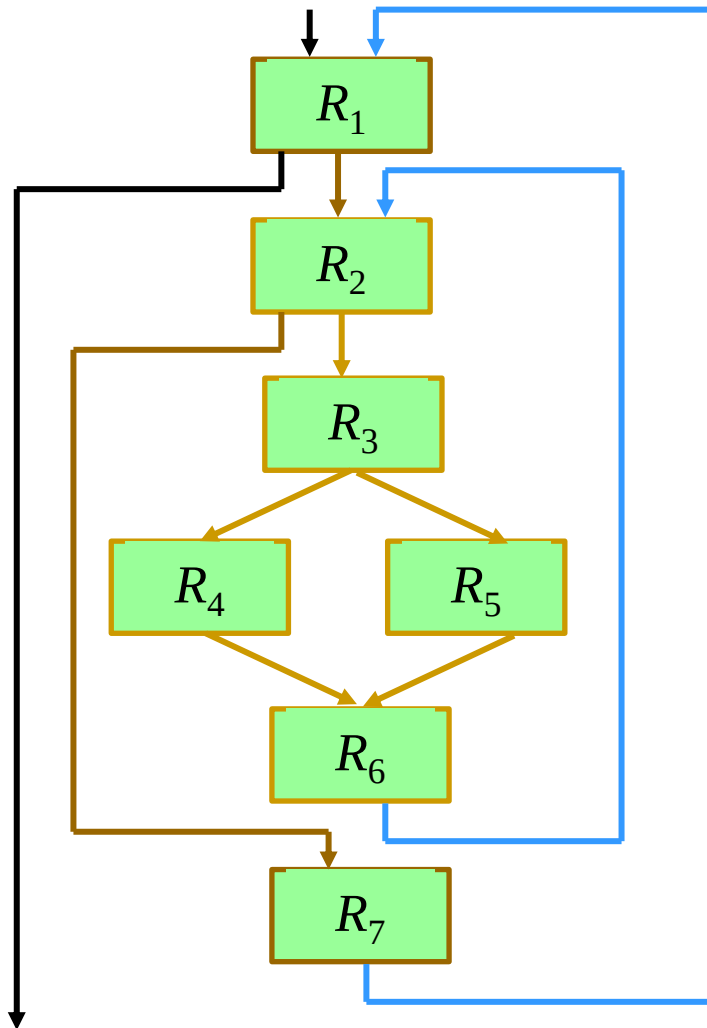
- (1) Каждый базовый блок B может рассматриваться как область $R = \langle \{B\}, \emptyset \rangle$. Такая область называется **область-лист**.
- (2) Пусть L – самый внутренний цикл гнезда циклов.
Тело цикла L (все узлы и ребра, за исключением обратных ребер к заголовку цикла) можно заменить узлом, представляющим область R . Такая область называется **область-тело**.
- (3) Если к области-телу R , соответствующей телу цикла L присоединить обратное ребро к заголовку цикла L , получится новая область Q . Такая область называется **область-цикл**.

10.1 Структурный анализ графа потока управления

10.1.2. Виды областей. Пример



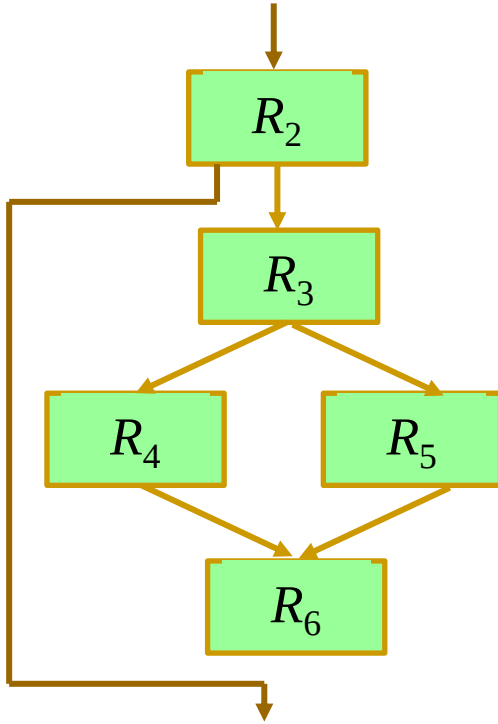
Фрагмент процедуры



Базовые блоки – области-листья

10.1 Структурный анализ графа потока управления

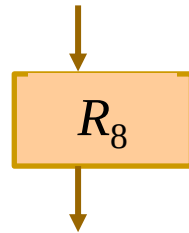
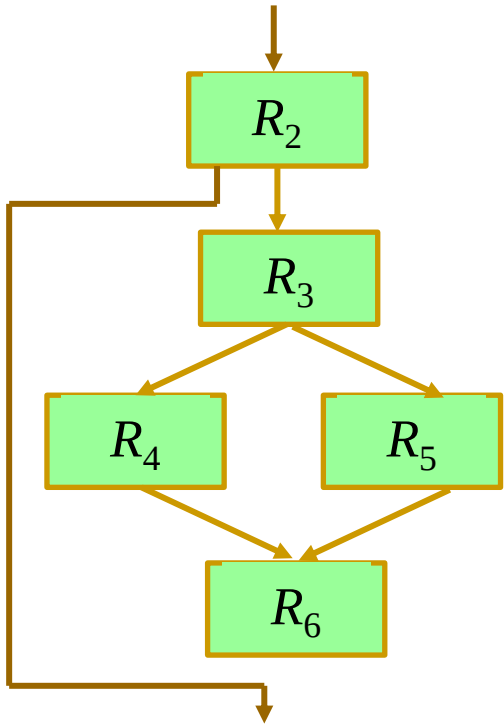
10.1.2. Виды областей. Пример



Тело внутреннего цикла

10.1 Структурный анализ графа потока управления

10.1.2. Виды областей. Пример

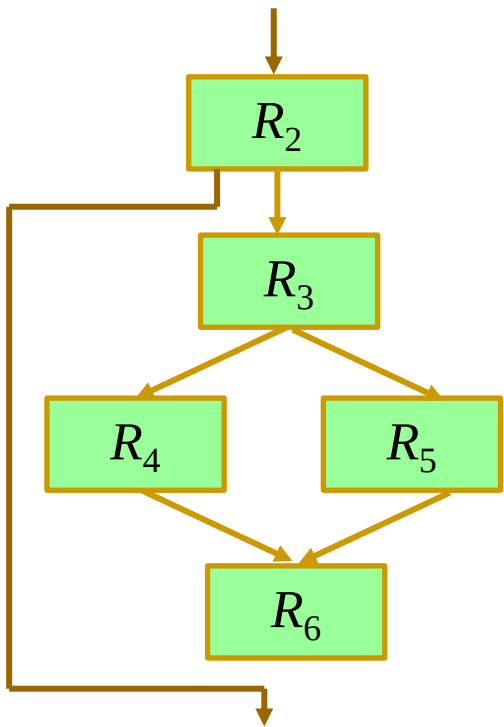


можно заменить на
область-тело R_8

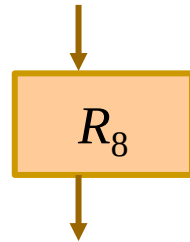
Тело внутреннего цикла

10.1 Структурный анализ графа потока управления

10.1.2. Виды областей. Пример

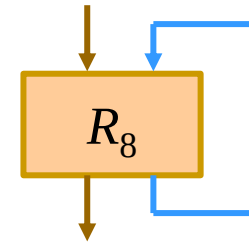


Тело внутреннего цикла

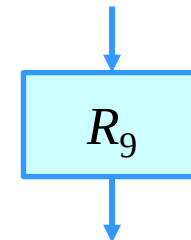


можно заменить на
область-тело R_8

добавив обратную
дугу,



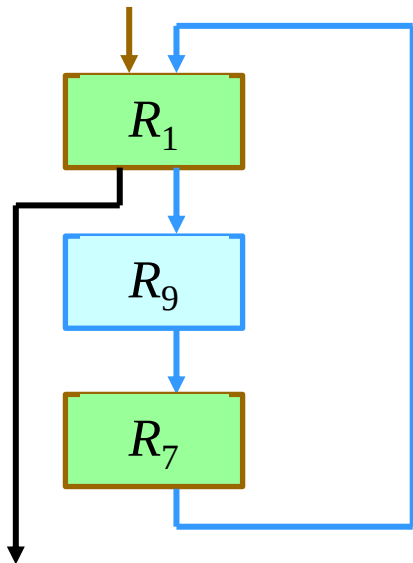
получим область-
цикл R_9



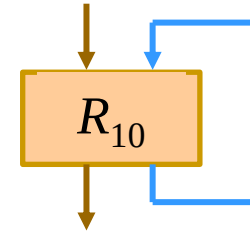
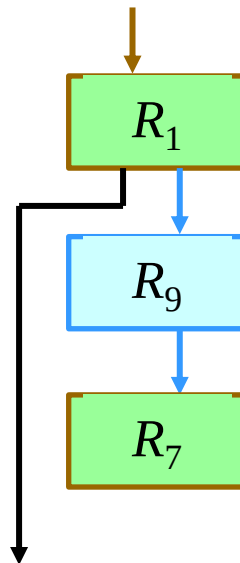
10.1 Структурный анализ графа потока управления

10.1.2. Виды областей. Пример

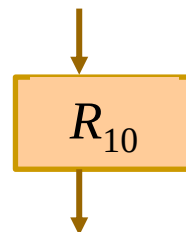
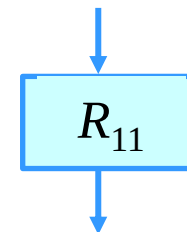
Граф потока
управления примет вид



Тело внешнего цикла



Добавив обратную
дугу, получим
область-цикл R_{11}

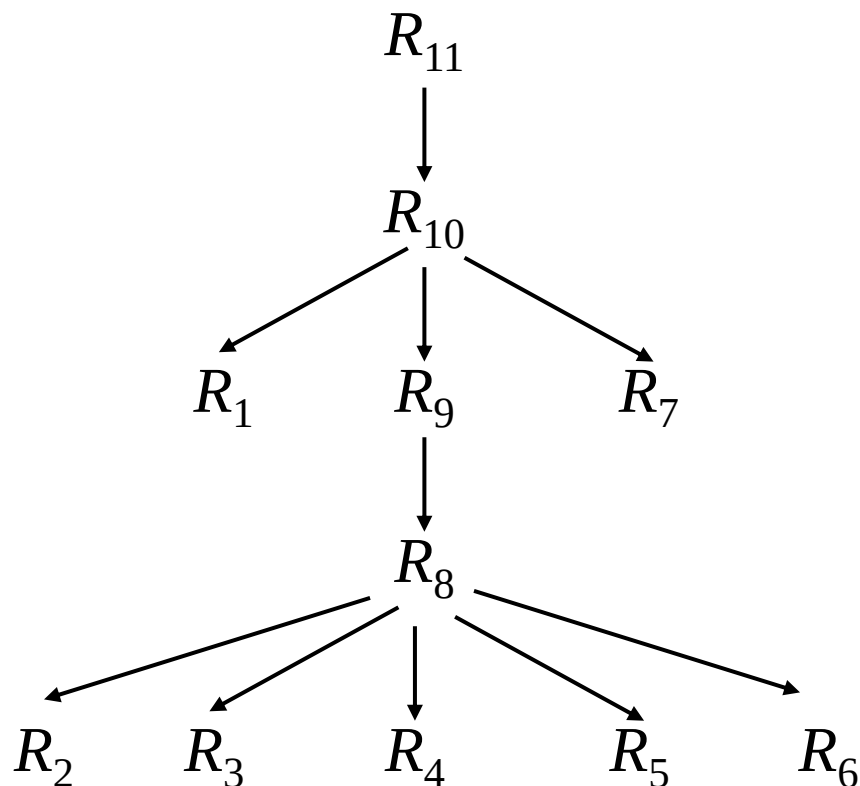


Заменяется
областью-телом R_{10}

10.1 Структурный анализ графа потока управления

10.1.2. Виды областей. Пример

Дерево управления

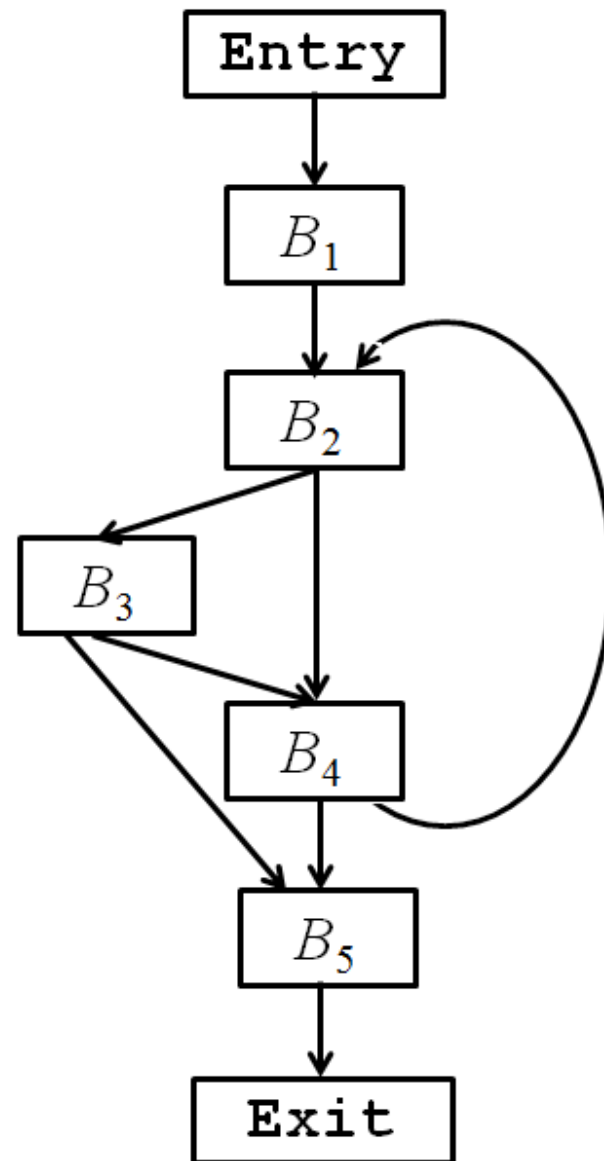


10.1 Структурный анализ графа потока управления

10.1.3. Выделение областей

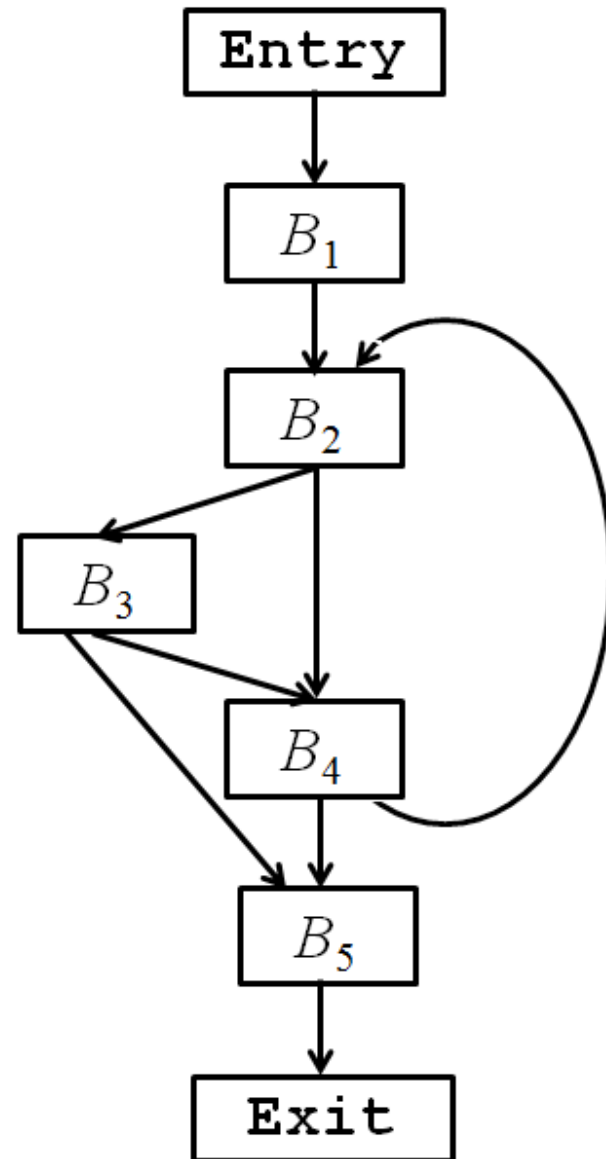
◇ Рассмотрим ГПУ, показанный на рисунке.

B_1	$i \leftarrow -m, 1$	d_1
	$j \leftarrow n$	d_2
	$a \leftarrow u1$	d_3
B_2	$i \leftarrow +i, 1$	d_4
B_3	$a \leftarrow u2$	d_5
B_4	$j \leftarrow u3$	d_6
B_5		

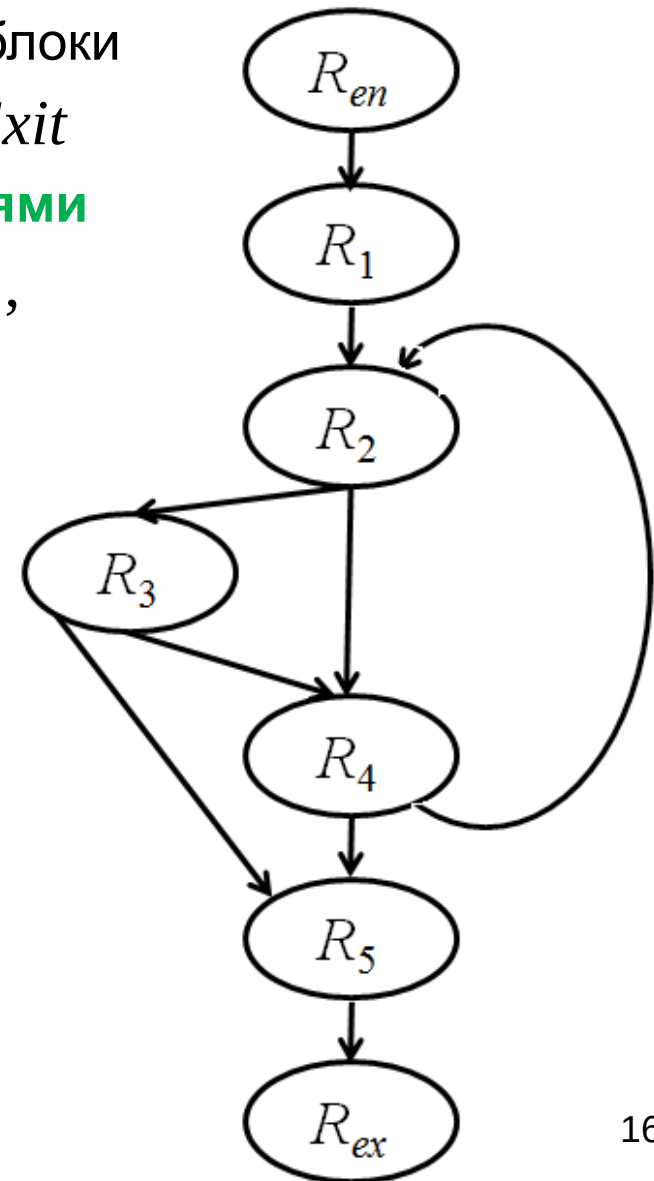


10.1 Структурный анализ графа потока управления

10.1.3. Выделение областей

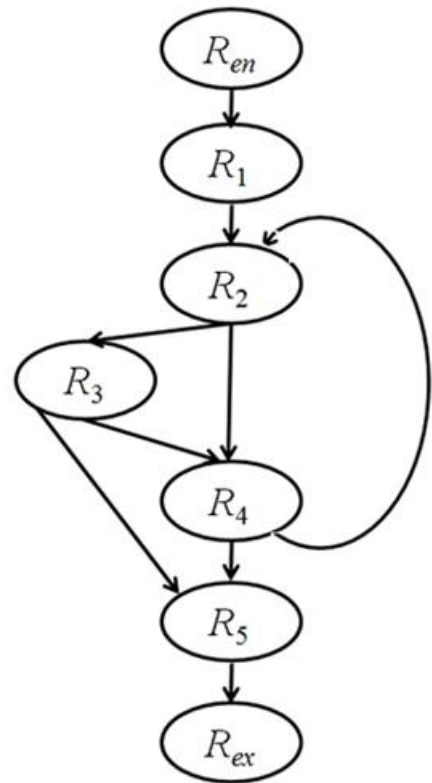


- 1) Заменяя базовые блоки $Entry, B_1, \dots, B_5, Exit$ **областями-листьями** $R_{en}, R_1, \dots, R_5, R_{ex}$, получим граф



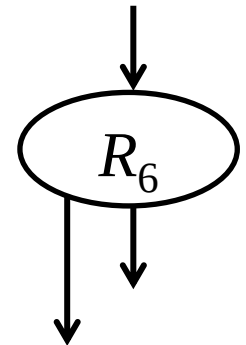
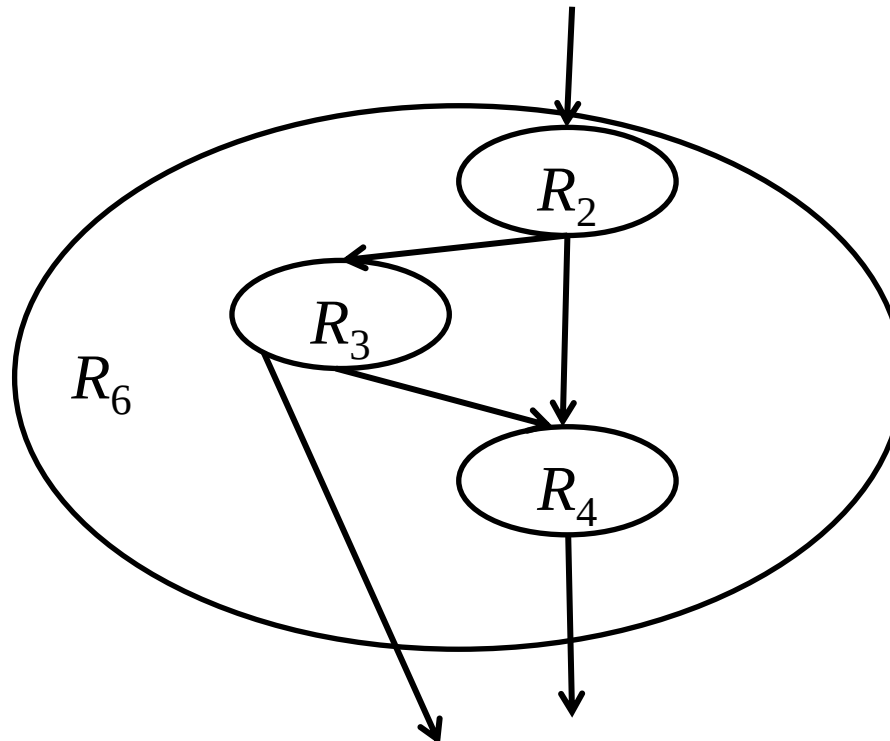
10.1 Структурный анализ графа потока управления

10.1.3. Выделение областей



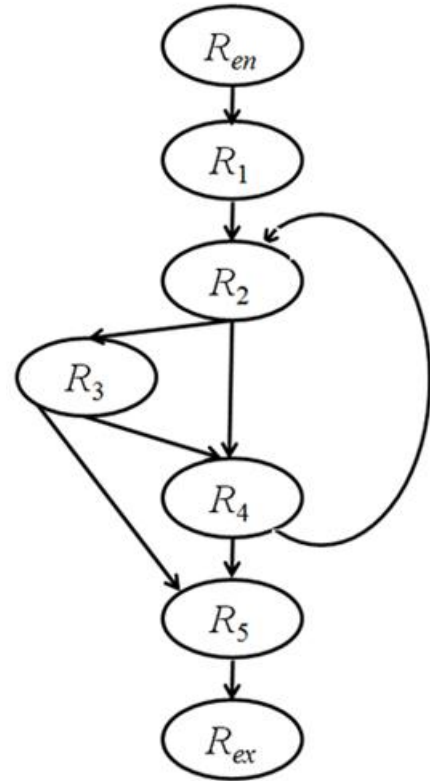
2) Области-листья R_2 , R_3 и R_4 и ребра $R_1 \rightarrow R_2$, $R_2 \rightarrow R_3$, $R_2 \rightarrow R_4$, $R_3 \rightarrow R_4$, $R_3 \rightarrow R_5$, $R_4 \rightarrow R_5$ составляют **область-тело** R_6 .

$R_6 = \langle \{R_2, R_3, R_4\}, \{R_1 \rightarrow R_2, R_2 \rightarrow R_3, R_2 \rightarrow R_4, R_3 \rightarrow R_4, R_3 \rightarrow R_5, R_4 \rightarrow R_5\} \rangle$

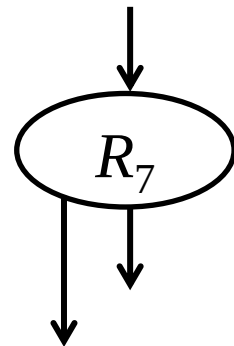
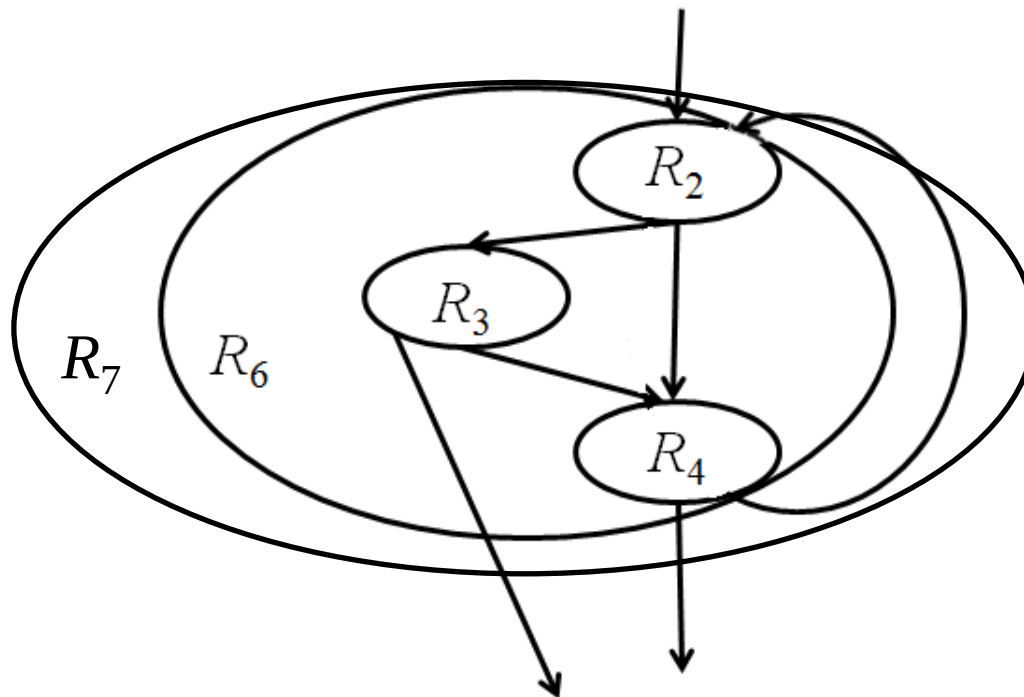


10.1 Структурный анализ графа потока управления

10.1.3. Выделение областей



3) Область-тело R_6 , и обратное ребро $R_4 \rightarrow R_2$ составляют **область-цикл** $R_7 = \langle \{R_6\}, \{R_4 \rightarrow R_2\} \rangle$.



10.1 Структурный анализ графа потока управления

10.1.4. Алгоритм построения иерархии областей

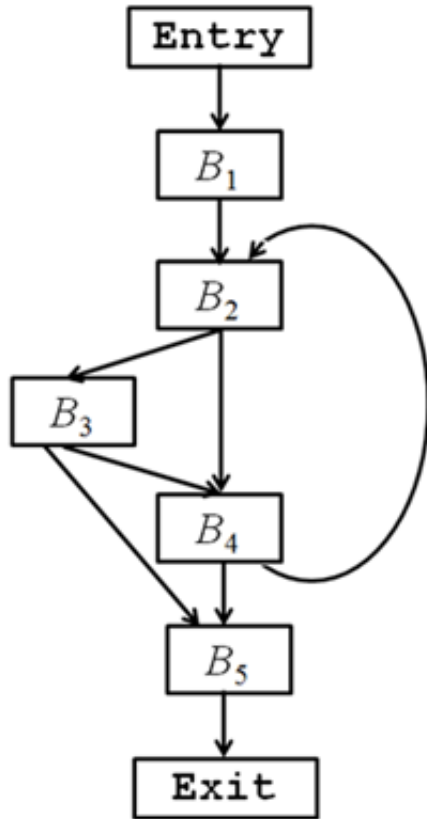
◇ Алгоритм.

10. Найти все естественные циклы.
2. Найденные естественные циклы упорядочить изнутри гнезд наружу, т.е. начиная с наиболее внутренних циклов.
3. Выбрать очередной естественный цикл (сначала – самый первый, потом – следующий по порядку).
Если циклов больше нет, **алгоритм заканчивается**.
4. Тело выбранного цикла L (все узлы и ребра, за исключением обратных ребер к заголовку) заместить узлом R , представляющим область-тело.
После замещения обратное ребро в заголовок L становится петлей.
5. Построить область-цикл Q , представляющую цикл L (в отличие от области R область Q не содержит петли).
Перейти к шагу 3.

10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

◇ **Пример.** Применим алгоритм к следующему ГПУ

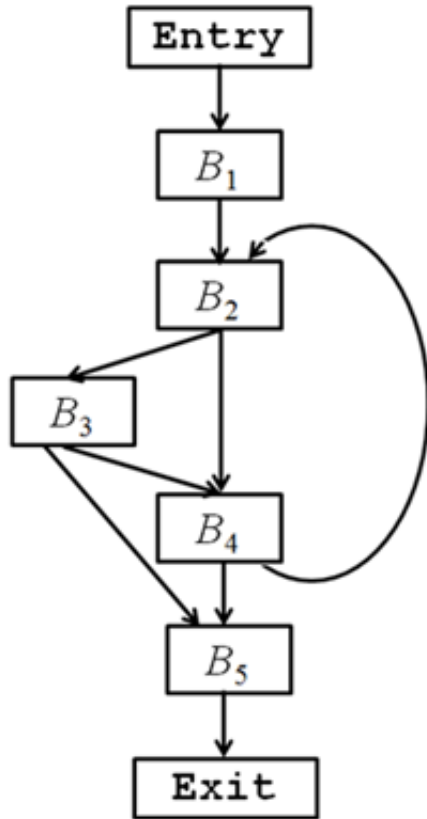


Исходный ГПУ

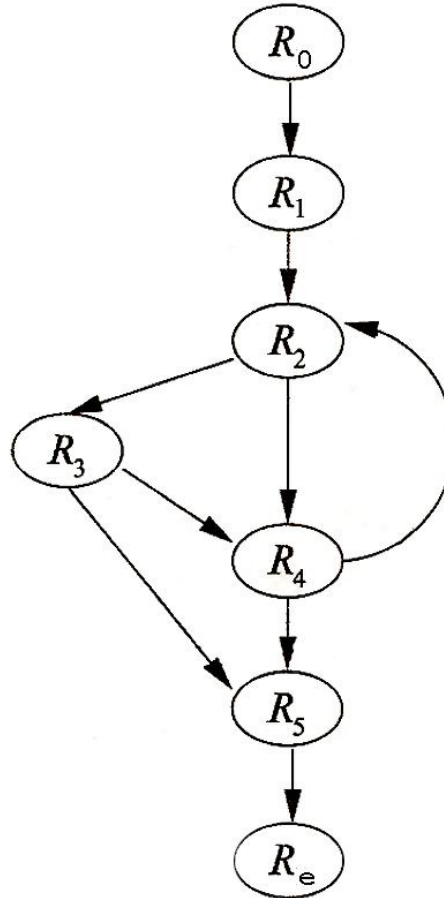
10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

◇ **Пример.** Применим алгоритм к следующему ГПУ



Исходный ГПУ

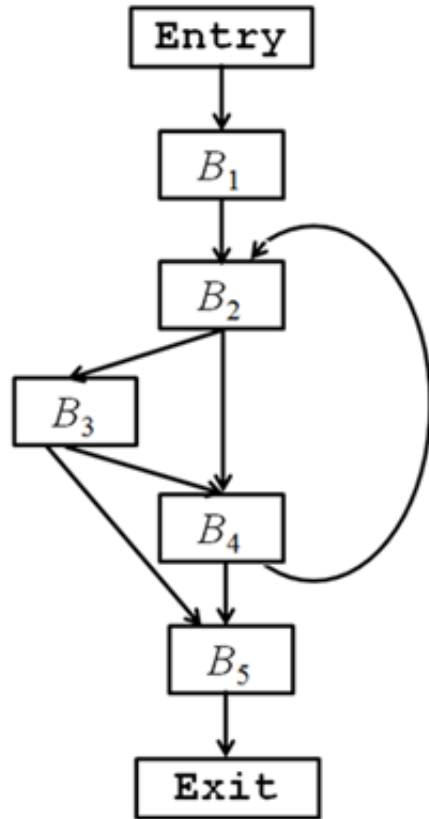


Базовые блоки
заменены областями-
листьями

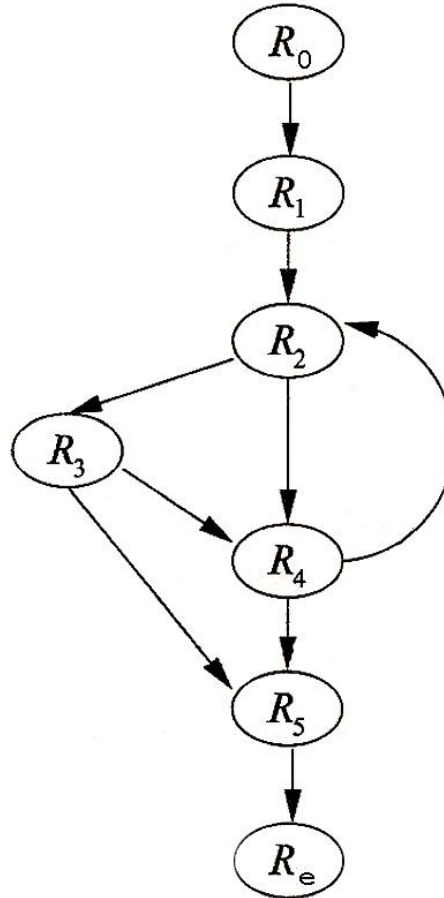
10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

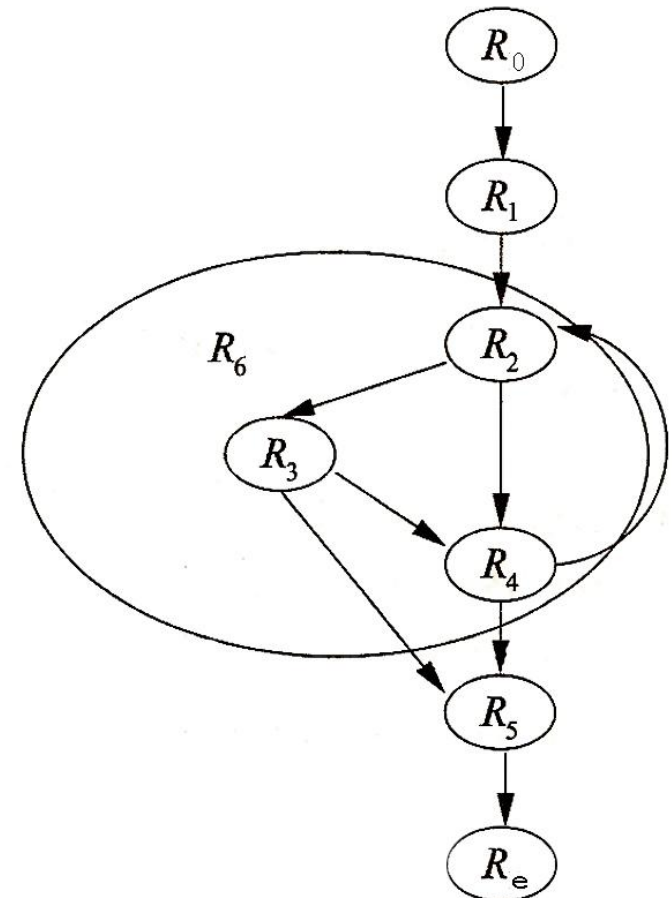
◇ **Пример.** Применим алгоритм к следующему ГПУ



Исходный ГПУ



Базовые блоки
заменены областями-
листьями

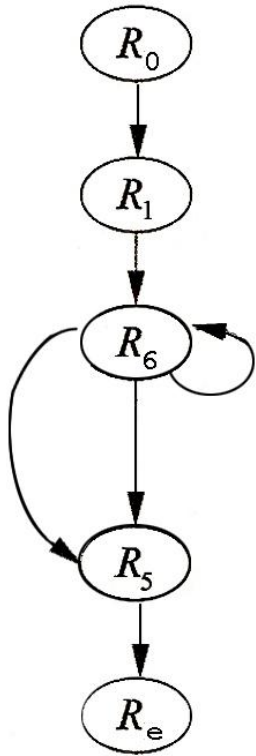


После шага 2

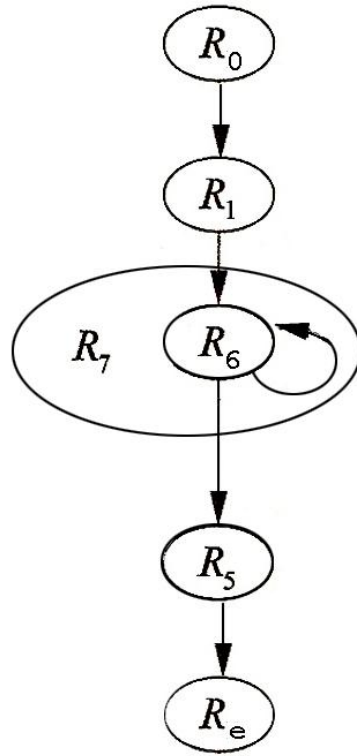
10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

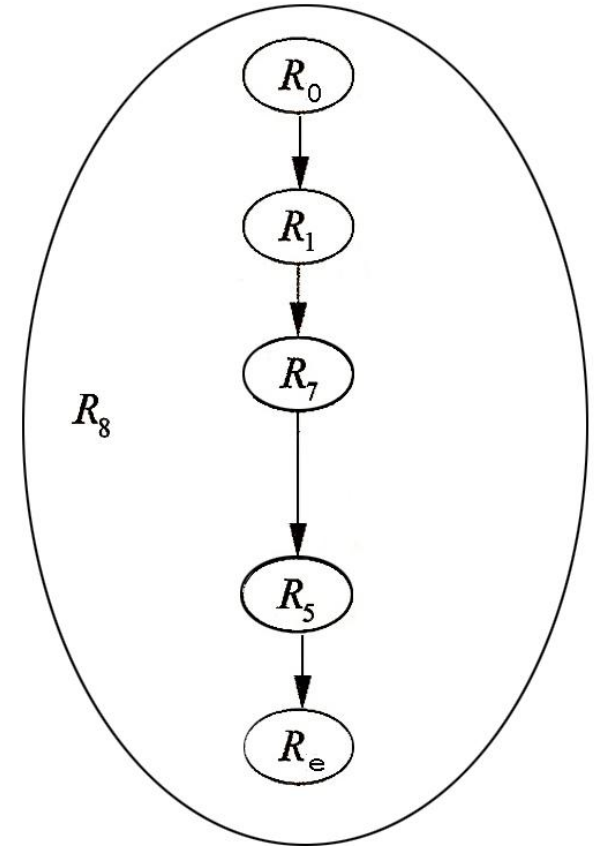
◇ Пример.



После шага 4



После шага 5

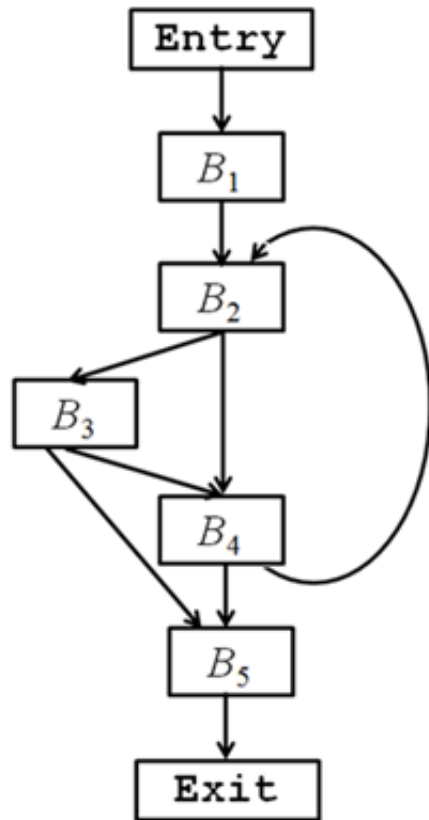


Область R_8 – весь ГПУ

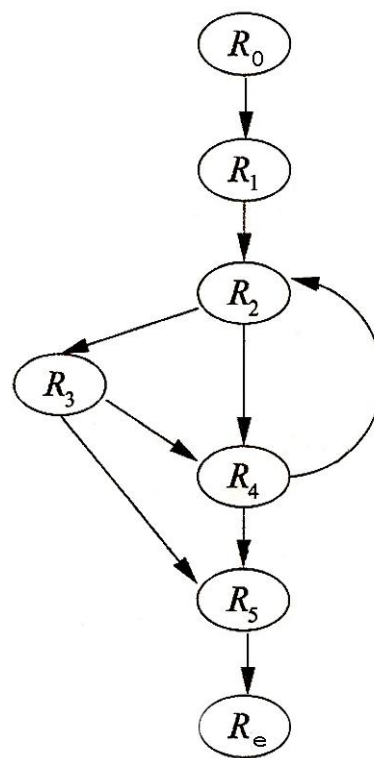
10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

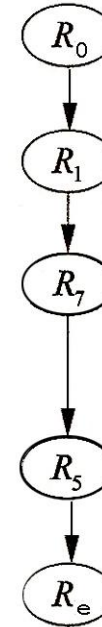
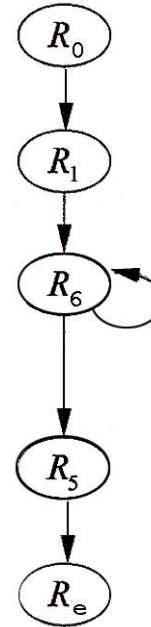
◇ Пример.



Исходный ГПУ



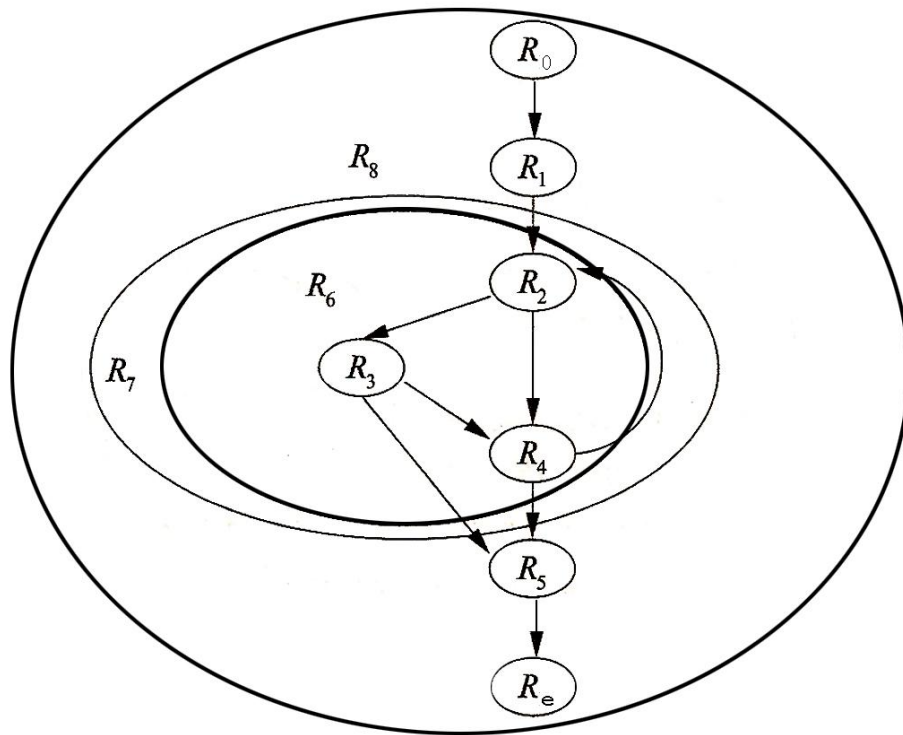
Последовательность преобразований



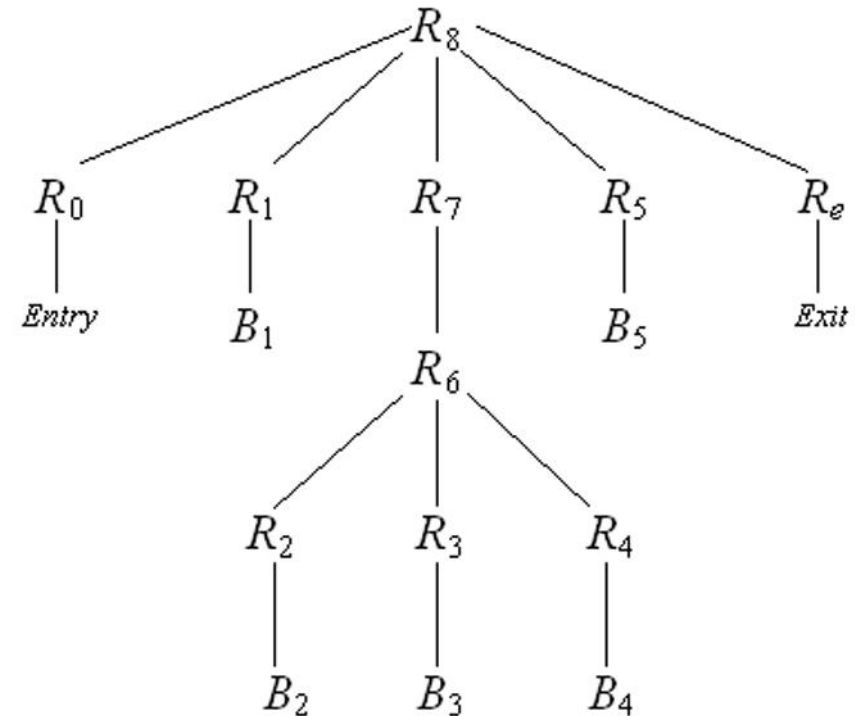
10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

◇ Пример.



Вся иерархия

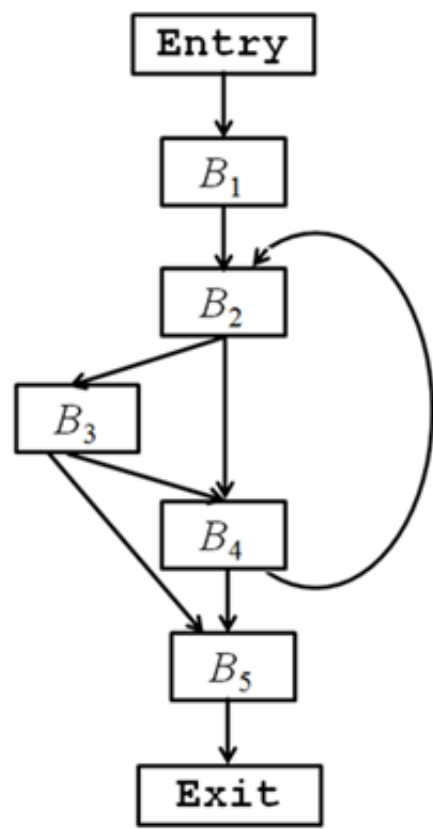


Дерево управления

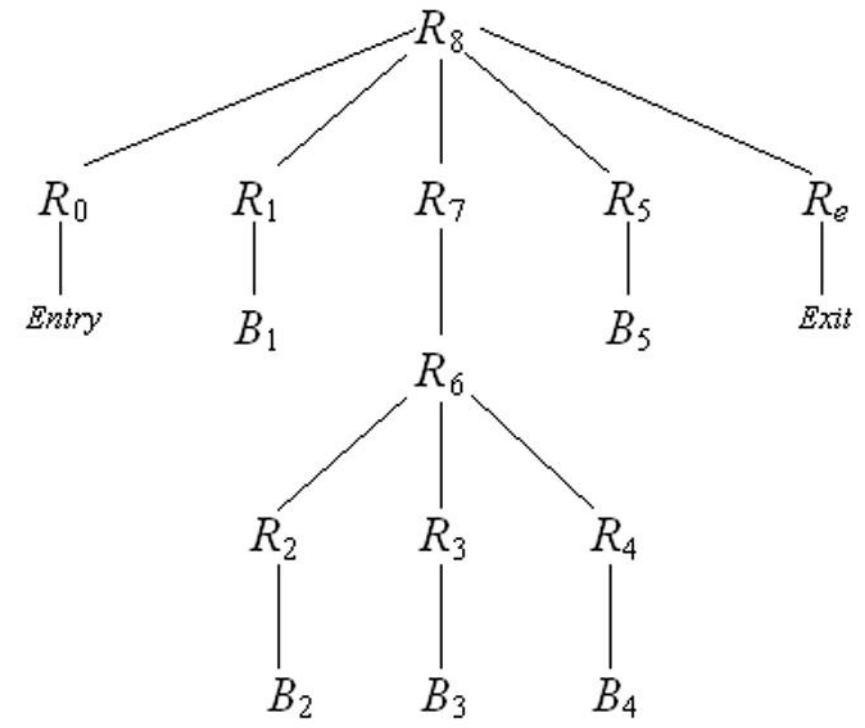
10.1 Структурный анализ графа потока управления

10.1.5. Построение иерархии областей

◇ Пример.



Исходный ГПУ



Дерево управления

10.1 Структурный анализ графа потока управления

10.1.6. Алгоритм построения восходящего порядка областей

- ◇ **Вход:** приводимый ГПУ G .
- ◇ **Выход:** список областей графа G , который может использоваться в задачах анализа потоков данных на основе областей.
- ◇ **Метод:** выполняем следующие действия.
 - 1) Составляем список областей-листьев, состоящих из отдельных блоков в произвольном порядке.
 - 2) Выбираем очередной естественный цикл L , такой, что все области, соответствующие естественным циклам, содержащимся в L , уже внесены в список. Сначала добавляем в список область-тело для L , а затем – область-цикл L .
 - 3) Если весь граф G является естественным циклом, добавляем в конец списка область, состоящую из всего графа потока целиком.

10.1 Структурный анализ графа потока управления

10.1.7. Скорость сходимости итерационных алгоритмов

- ◇ Построение областей и дерева управления позволяет ускорить обход графа потока управления во время выполнения различных алгоритмов анализа потока данных.
- ◇ В алгоритмах анализа потока данных, в которых в качестве сбора используется объединение, удастся, заменив каждый цикл узлом типа область-цикл, оставить только ациклические пути для распространения атрибутов.

Это позволяет сократить время выполнения соответствующего анализа потока данных.

10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

- ◇ На каждом уровне иерархии областей:
 - ◇ Для каждой области R и для каждой подобласти $R' \in R$ вычисляется передаточная функция $f_{R, In[R']}$, суммирующая влияние всех возможных путей в R , ведущих от входа в R ко входу в R' .
- ◇ Мы рассматриваем анализ в прямом направлении (сверху вниз). Анализ в обратном направлении рассматривается аналогично с очевидными изменениями.

10.2. Анализ потока данных на основе областей

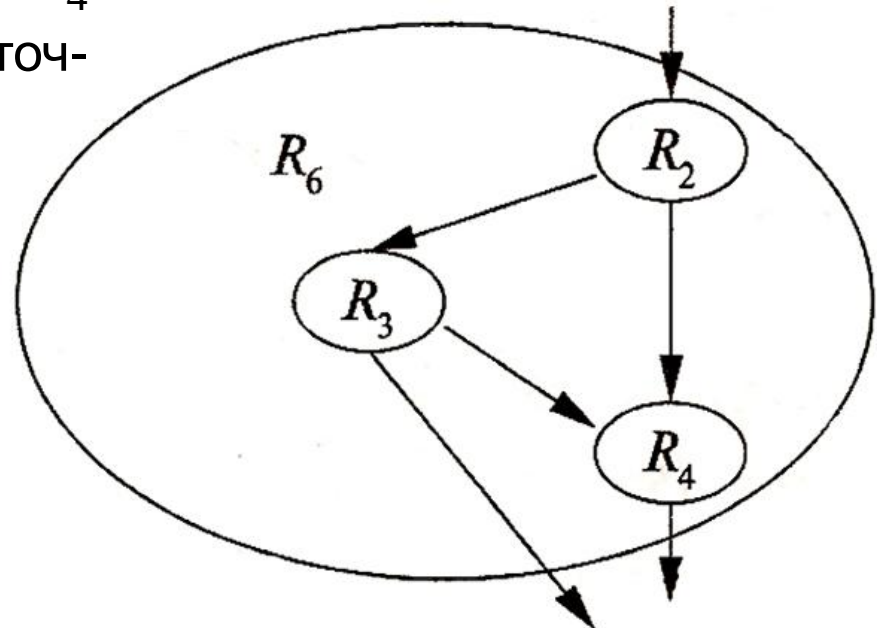
10.2.1. Схема анализа потока данных на основе областей

- ◇ На каждом уровне иерархии областей:
 - ◇ Для каждой области R и для каждой подобласти $R' \in R$ вычисляется передаточная функция $f_{R, In[R']}$, суммирующая влияние всех возможных путей в R , ведущих от входа в R ко входу в R' .
 - ◇ Например, для области R_6 и её подобластей R_2 , R_3 и R_4 будут вычислены передаточные функции

$$f_{R_6, In[R_2]}$$

$$f_{R_6, In[R_3]}$$

$$f_{R_6, In[R_4]}$$



10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

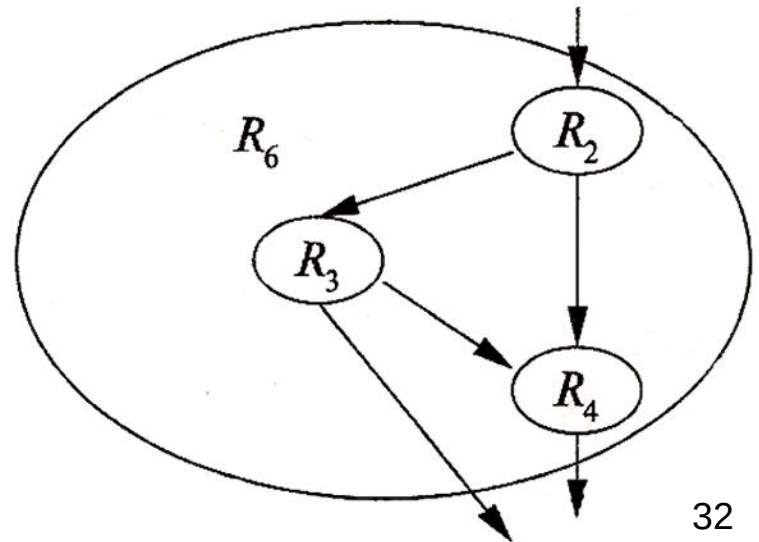
- ◇ На каждом уровне иерархии областей:
 - ◇ Область $R' \in R$ называется *выходной* подобластью области R , если у R' есть выходное ребро к некоторой области, не принадлежащей области R .
 - ◇ Для каждой выходной области $R' \in R$ вычисляется передаточная функция $f_{R,Out}[R']$, суммирующая влияние всех возможных путей в R , ведущих от выхода из R' к выходу из R .

10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

- ◇ На каждом уровне иерархии областей:
 - ◇ Область $R' \in R$ называется *выходной* подобластью области R , если у R' есть выходное ребро к некоторой области, не принадлежащей области R .
 - ◇ Для каждой выходной области $R' \in R$ вычисляется передаточная функция $f_{R,Out}[R']$, суммирующая влияние всех возможных путей в R , ведущих от выхода из R' к выходу из R .
 - ◇ Например, для области R_6 и её выходных подобластей R_3 и R_4 будут вычислены передаточные функции

$$f_{R_6,Out}[R_3] \quad f_{R_6,Out}[R_4]$$



10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

◇ Первый этап. Построение передаточных функций всех областей (от листьев к корню дерева управления).

◇ Сначала обрабатываются области-листья (отдельные блоки):

для каждой области-листа R , состоящей из блока B ,

$$f_{R,In[B]} = I \text{ (тождественная функция)}$$

$$f_{R,Out[B]} = f_B \text{ (передаточная функция блока } B\text{).}$$

10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

◇ Первый этап. Построение передаточных функций всех областей (от листьев к корню дерева управления).

◇ Сначала обрабатываются области-листья (отдельные блоки):

для каждой области-листа R , состоящей из блока B ,

$$f_{R,In[B]} = I \text{ (тождественная функция)}$$

$$f_{R,Out[B]} = f_B \text{ (передаточная функция блока } B\text{).}$$

◇ Перемещение вверх по иерархии:

◆ **R – область-тело:** ребра, принадлежащие R , образуют ациклический граф на подобластях R , что позволяет при вычислении передаточных функций использовать топологический порядок областей.

◆ **R – область-цикл:** учитывается только влияние обратных ребер, ведущих к заголовку R .

10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

- ◇ Первый этап. Построение передаточных функций всех областей (от листьев к корню дерева управления).
 - ◇ Сначала обрабатываются области-листья (отдельные блоки).
 - ◇ Перемещение вверх по иерархии:
 - ◆ **R – область-тело:** ребра, принадлежащие R , образуют ациклический граф на подобластях R , что позволяет при вычислении передаточных функций использовать топологический порядок областей.
 - ◆ **R – область-цикл:** учитывается только влияние обратных ребер, ведущих к заголовку R .
- ◇ В конце обработки достигается вершина иерархии и вычисляются передаточные функции области R_n , представляющей собой весь граф потока.

10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

- ◇ **Второй этап.** Анализ иерархии областей (от корня к листьям дерева управления).
 - ◇ Области просматриваются в обратном порядке, начиная с области R_n и далее, опускаясь вниз по иерархии. Для каждой области вычисляются значения потока данных на входе.
 - ◇ Чтобы получить значения потока данных на входе $R \in R_n$ используется передаточная функция $f_{R_n, In}[R]$
 - ◇ Вычисления повторяются, до тех пор, пока не будут достигнуты области-листья (базовые блоки).

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

- ◇ Для анализа потока данных с использованием итеративного алгоритма в базовых блоках потребовалась замкнутость структуры потока данных относительно операции *композиции* функций.
- ◇ Для анализа потока данных на основе областей к передаточным функциям применяется уже не только операция композиции, но еще две операции, позволяющие вычислять передаточные функции для областей по передаточным функциям для их подобластей:
 - ◇ операция *сбора* (для входов в компоненты подобластей) и
 - ◇ операция *замыкания* (для циклов).
- ◇ Для применимости указанных операций требуется, чтобы структура потока данных (множество передаточных функций) была замкнута относительно трех операций: композиции, сбора и замыкания.

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ 1. Замкнутость относительно композиции

◇ *Замкнутость* множества $\mathcal{F}$ относительно композиции означает, что композиция двух передаточных функций, принадлежащих множеству $\mathcal{F}$, является передаточной функцией, принадлежащей множеству $\mathcal{F}$.

◇ **Утверждение.** Множество $\mathcal{GK}$ передаточных функций вида *gen-kill*, замкнуто относительно композиции:

если $f_1 \in \mathcal{GK}$ $f_2 \in \mathcal{GK}$, а $f = f_1 \circ f_2$, то и $f \in \mathcal{GK}$

Доказательство.

$$\begin{aligned}(f_1 \circ f_2)(x) &= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2) = \\ &= gen_2 \cup (gen_1 - kill_2) \cup (x - (kill_1 \cup kill_2)) = \\ &= gen \cup (x - kill),\end{aligned}$$

$$gen = gen_2 \cup (gen_1 - kill_2) \quad kill = kill_1 \cup kill_2.$$

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ 2. Операция сбора

- ◇ Операция *сбора* $\wedge_{\mathcal{F}}$ на множестве передаточных функций $\mathcal{F}$ определяется с помощью операции $\wedge$ сбора значений потока данных следующим образом:

$$(f_1 \wedge_{\mathcal{F}} f_2)(x) = f_1(x) \wedge f_2(x),$$

- ◇ Множество передаточных функций $\mathcal{F}$ *замкнуто относительно операции сбора* $\wedge_{\mathcal{F}}$, если для любых двух передаточных функций $f_1 \in \mathcal{F}$ и $f_2 \in \mathcal{F}$, их сбор $f = f_1 \wedge_{\mathcal{F}} f_2 \in \mathcal{F}$.
- ◇ **Замечание.** Множество передаточных функций $\mathcal{F}$, замкнутое относительно операции сбора $\wedge_{\mathcal{F}}$ является полурешеткой с операцией сбора $\wedge_{\mathcal{F}}$.

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ 3. Замкнутость относительно операции сбора

- ◇ **Утверждение.** Множество $\mathcal{GK}$ передаточных функций вида *gen-kill*, замкнуто относительно операции сбора $\wedge_{\mathcal{GK}}$.

Доказательство:

$$\begin{aligned}(f_1 \wedge_{\mathcal{GK}} f_2)(x) &= f_1(x) \wedge f_2(x) = \\ &= gen_1 \cup (x - kill_1) \cup gen_2 \cup (x - kill_2) = \\ &= (gen_1 \cup gen_2) \cup (x - (kill_1 \cap kill_2)) = \\ &= gen \cup (x - kill),\end{aligned}$$

$$gen = gen_1 \cup gen_2, \quad kill = kill_1 \cap kill_2.$$

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ 4. Операция замыкания

◇ Пусть f – передаточная функция тела цикла.

Тогда двум итерациям цикла будет соответствовать функция

$$f^2(x) = f \wedge_{\mathcal{F}} f,$$

а n итерациям цикла – функция

$$f^n = f^{n-1} \wedge_{\mathcal{F}} f.$$

Если количество итераций цикла неизвестно, его передаточная функция представляется как *замыкание* f .

◇ Пусть I – тождественная функция. Тогда $f^0 = I$, $f^1 = f$

◇ *Замыканием* передаточной функции $f \in \mathcal{F}$ называется функция f^* , определяемая формулой

$$f^* = I \wedge_{\mathcal{F}} \bigwedge_{n \geq 0} f^n$$

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ 5. Замкнутость относительно операции замыкания

◇ Множество передаточных функций F замкнуто относительно операции замыкания.

◇ **Утверждение.** Множество $\mathcal{GK}$ передаточных функций вида *gen-kill* замкнуто относительно операции замыкания:
если $\forall f \in \mathcal{GK}$ то и $f^* \in \mathcal{GK}$.

Доказательство:

$$\begin{aligned} f^2(x) &= f(f(x)) = \text{gen} \cup (\text{gen} \cup (x - \text{kill}) - \text{kill}) = \\ &= (\text{gen} \cup \text{gen}) \cup (x - (\text{kill} \cup \text{kill})) = \text{gen} \cup (x - \text{kill}). \end{aligned}$$

$$f^n(x) = \text{gen} \cup (x - \text{kill}) \text{ (по индукции)}$$

$$\begin{aligned} f^*(x) &= I \wedge f^1(x) \wedge f^2(x) \wedge \dots = x \cup (\text{gen} \cup (x - \text{kill})) = \\ &= \text{gen} \cup (x \cup (x - \text{kill})) = \text{gen} \cup x \end{aligned}$$

$$(x - \text{kill}) \subseteq x$$

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ 5. Замкнутость относительно операции замыкания

◇ Итак, если $f(x) = (gen \cup (x - kill))$, то $f^*(x) = gen \cup x$

Следовательно, для $\forall f \in \mathcal{GK}$

$$gen_{f^*} = gen_f$$

$$kill_{f^*} = \emptyset$$

Таким образом, циклы не влияют на анализ достигающих определений

10.2. Анализ потока данных на основе областей

10.2.2. Вычисление передаточных функций

◇ Формулы для функций вида GK :

Если $f_1(x) = gen_1 \cup (x - kill_1)$ и $f_2(x) = gen_2 \cup (x - kill_2)$, то

$$(1) \quad (f_1 \circ f_2)(x) = gen^\circ \cup (x - kill^\circ),$$

где $gen^\circ = gen_2 \cup (gen_1 - kill_2)$, $kill^\circ = kill_1 \cup kill_2$.

$$(2) \quad (f_1 \wedge f_2)(x) = gen^\wedge \cup (x - kill^\wedge),$$

где $gen^\wedge = gen_1 \cup gen_2$, $kill^\wedge = kill_1 \cap kill_2$

Если $f(x) = (gen \cup (x - kill))$, то $f^*(x) = gen^* \cup (x - kill^*)$,

где $gen^* = gen$, $kill^* = \emptyset$

10.2. Анализ потока данных на основе областей

10.2.3. Алгоритм анализа на основе областей

- ◇ **Вход:** структура потока данных, замкнутая относительно операций композиции, сбора и замыкания, приводимый граф потока управления G .
- ◇ **Выход:** значения потока данных $In[B]$ для каждого блока $B \in G$.
- ◇ **Метод:** выполнить следующие действия:
 - 1) С помощью алгоритма 10.1.3 определить области и их топологический порядок (снизу-вверх)
 - 2) *Восходящий просмотр* для вычисления передаточных функций областей $R_1, R_2, \dots, R_n$: (R_n – область самого верхнего уровня).
 - 2а) R – область-лист, соответствующая блоку B :

$$f_{R, In[B]} = I, \quad f_{R, Out[B]} = f_B.$$

10.2. Анализ потока данных на основе областей

10.2.3. Алгоритм анализа на основе областей

◇ **Метод:** выполнить следующие действия:

- 2) *Восходящий просмотр* для вычисления передаточных функций областей $R_1, R_2, \dots, R_n$:
- 2b) R – область-тело:

for each $S \in R$ (в топологическом порядке) {

$$f_{R,In}[S] = \bigwedge_{B \in Pred(S)} f_{R,Out}[B]$$

/* Если S – заголовок области R , то $f_{R,In}[S] = I$ */

for each (выходной блок $B \in S$) $f_{R,Out}[B] = f_{S,Out}[B] \circ f_{R,In}[S]$;

}

10.2. Анализ потока данных на основе областей

10.2.3. Алгоритм анализа на основе областей

◇ **Метод:** выполнить следующие действия:

2) *Восходящий просмотр* для вычисления передаточных функций областей $R_1, R_2, \dots, R_n$:

2с) R – область-цикл:

$$f_{R,In}[S] = \left(\bigwedge_{B \in Pred(S)} f_{R,Out}[B] \right)^*$$

/* Если S – заголовок области R , то $f_{R,In}[S] = I$ */

for each (выходной блок $B \in S$) $f_{R,Out}[B] = f_{S,Out}[B] \circ f_{R,In}[S]$;

10.2. Анализ потока данных на основе областей

10.2.3. Алгоритм анализа на основе областей

◇ **Метод:** выполнить следующие действия:

3) *Нисходящий просмотр* для вычисления значений $In[R]$ в начале каждой области:

$In[R_n] = Out[Entry];$

for each $S \in R$ (в нисходящем порядке)

$In[R] = f_{R', In[R]}(In[R']),$

*/** R' – область, непосредственно

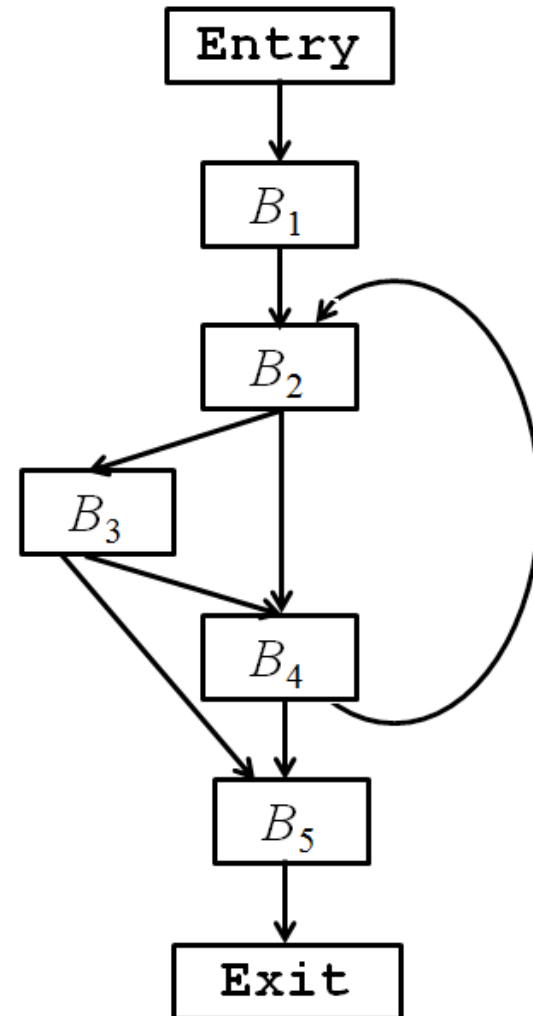
охватывающая область R **/*

10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

- ◇ Применим алгоритм 10.2.3 для поиска достигающих определений с помощью анализа на основе областей в рассматриваемой программе.

B_1	$i \leftarrow -m, 1$	d_1
	$j \leftarrow n$	d_2
	$a \leftarrow u1$	d_3
B_2	$i \leftarrow +i, 1$	d_4
B_3	$a \leftarrow u2$	d_5
B_4	$j \leftarrow u3$	d_6
B_5		



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Замечание:** поиск достигающих определений обычным способом (с помощью итерационного алгоритма, который изучался в прошлом семестре) дает следующие результаты

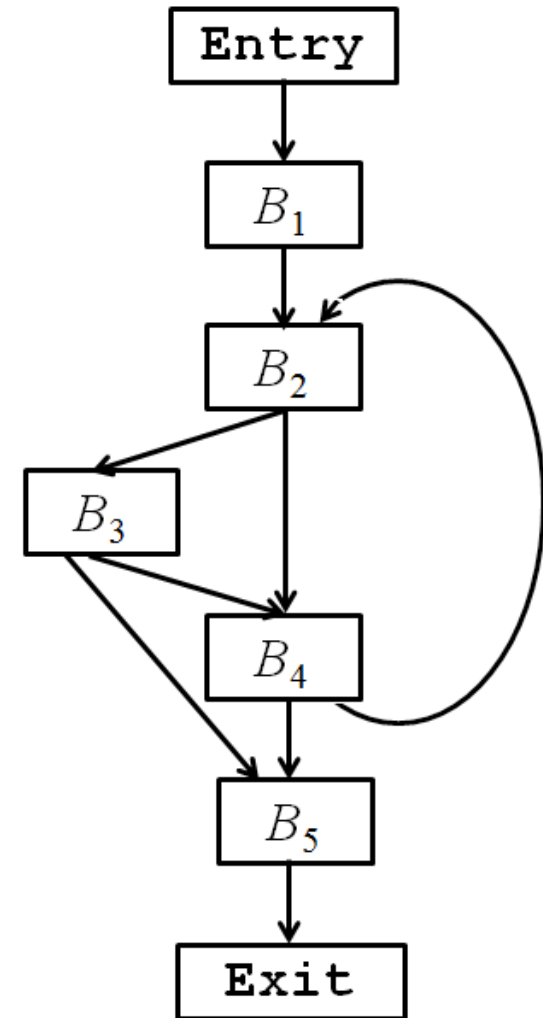
$$In[B_1] = \emptyset$$

$$In[B_2] = \{d_1, d_2, d_3, d_4, d_5, d_6\}$$

$$In[B_3] = \{d_2, d_3, d_4, d_5, d_6\}$$

$$In[B_4] = \{d_2, d_3, d_4, d_5, d_6\}$$

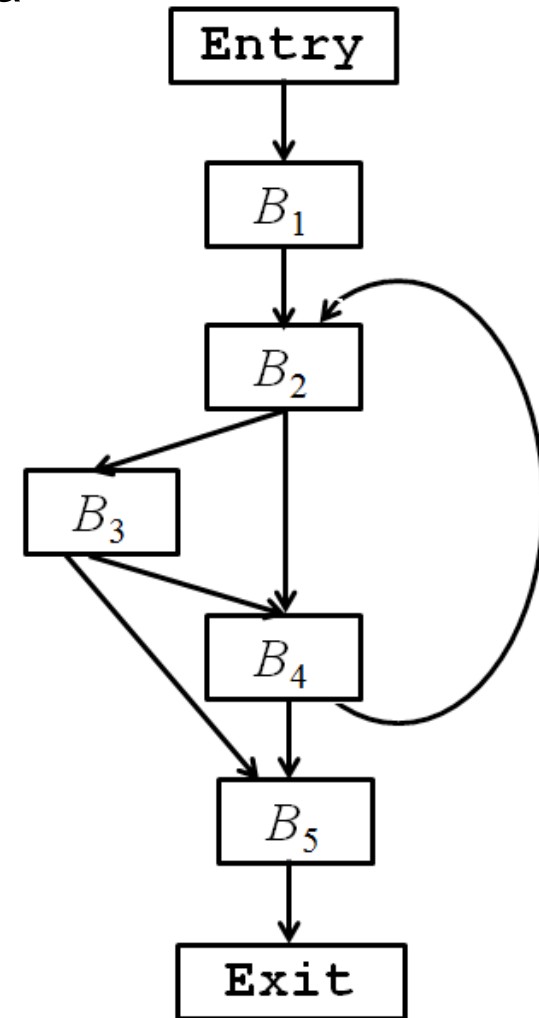
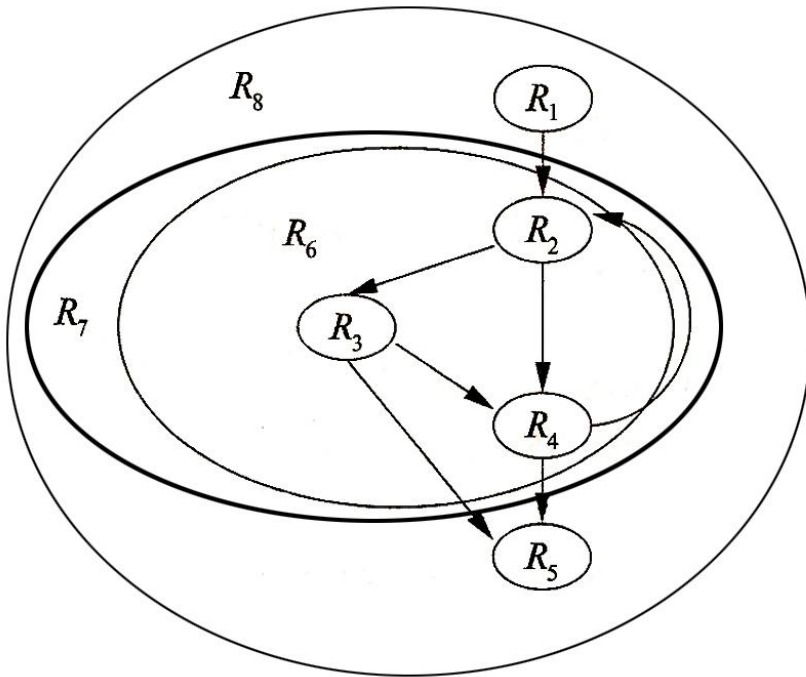
$$In[B_5] = \{d_2, d_3, d_4, d_5, d_6\}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

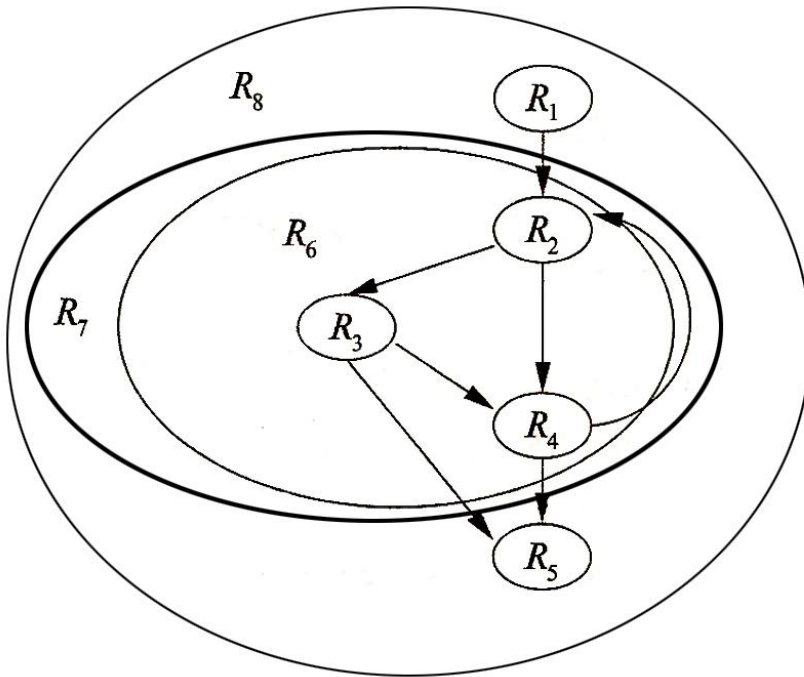
- ◇ Применим алгоритм 10.2.3 для поиска достигающих определений с помощью анализа на основе областей в рассматриваемой программе.
- ◇ **Шаг 1:** с помощью алгоритма 10.1.3 определим области и пронумеруем их в восходящем топологическом порядке (слайды 20 – 27): получим области $R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8$



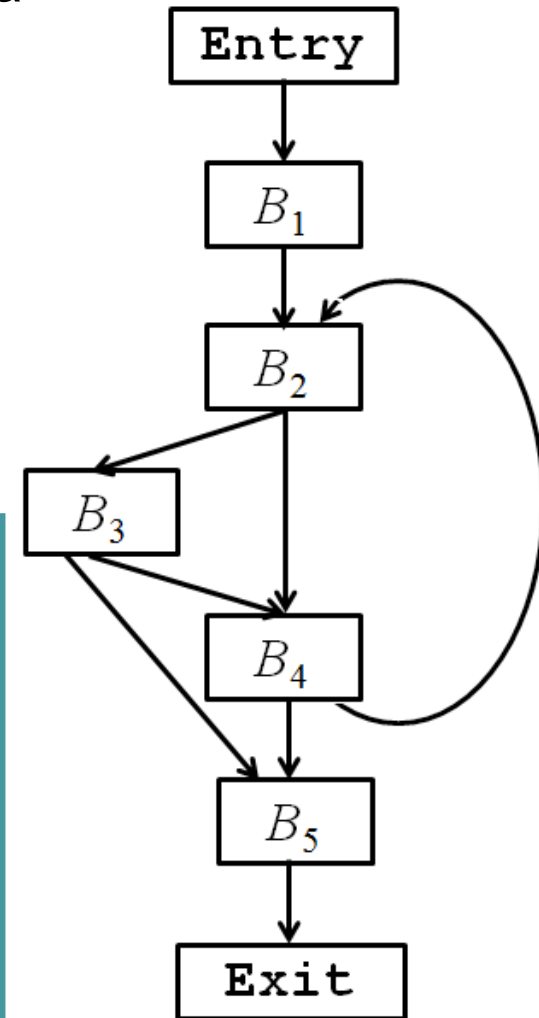
10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

- ◇ Применим алгоритм 10.2.3 для поиска достигающих определений с помощью анализа на основе областей в рассматриваемой программе.
- ◇ **Шаг 1:** с помощью алгоритма 10.1.3 определим области и пронумеруем их в восходящем топологическом порядке (слайды 20 – 27): получим области $R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8$



Области
 R_1, R_2, R_3, R_4, R_5
являются
областями-
листьями и
соответствуют
базовым блокам
 B_1, B_2, B_3, B_4, B_5



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2:** *Восходящий просмотр* для вычисления передаточных функций областей $R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8$

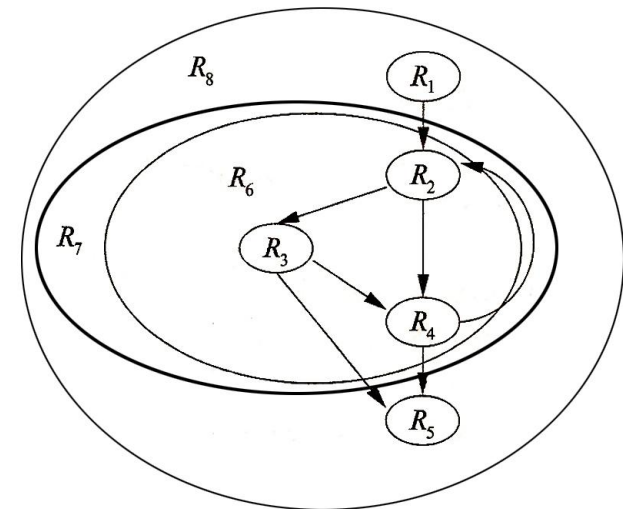
Шаг 2а): вычисление передаточных функций областей-листьев

R_1, R_2, R_3, R_4, R_5

$$f_{R_i, In[B_i]}(x) = I,$$

$$f_{R_i, Out[B_i]}(x) = f_{B_i}(x) = gen_{B_i} \cup (x - kill_{B_i})$$

B	B_1	B_2	B_3	B_4	B_5
gen_B	$\{d_1, d_2, d_3\}$	$\{d_4\}$	$\{d_5\}$	$\{d_6\}$	$\emptyset$
$kill_B$	$\{d_4, d_5, d_6\}$	$\{d_1\}$	$\{d_3\}$	$\{d_2\}$	$\emptyset$



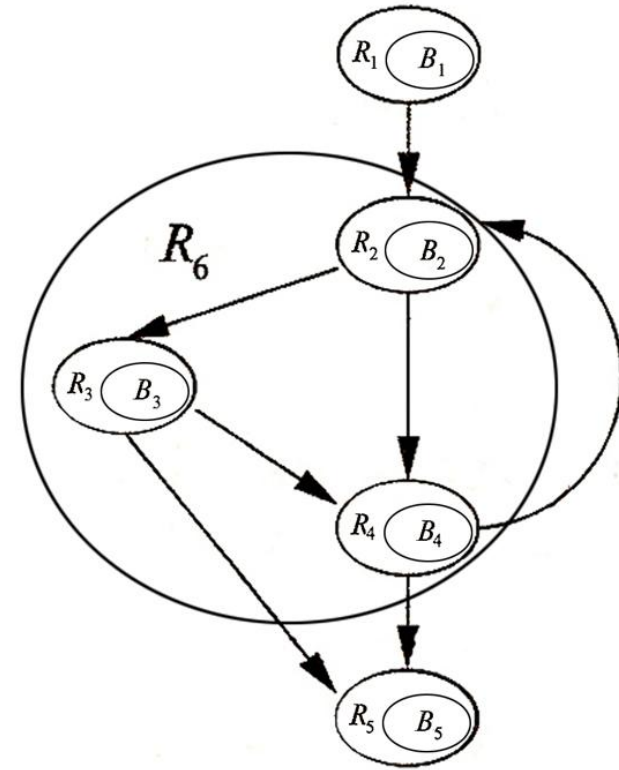
10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2b):** вычисление передаточной функции области-тела R_6

Область R_6 состоит из подобластей R_2 , R_3 и R_4 . Эти подобласти обрабатываются в топологическом порядке, построенном на шаге 1: R_2 , R_3 , R_4 .

Подобласть R_2 : у R_2 нет предшественников в пределах R_6 , так как обратное ребро $R_4 \rightarrow R_2$ не принадлежит R_6 . Следовательно



$$f_{R_6, In[R_2]} = f_{R_2, In[B_2]} = I,$$

$$f_{R_6, Out[B_2]} = f_{R_2, Out[B_2]} \circ f_{R_6, In[R_2]} = f_{B_2} \circ I = f_{B_2}$$

$$gen_{R_6, In[R_2]} = \emptyset, kill_{R_6, In[R_2]} = \emptyset$$

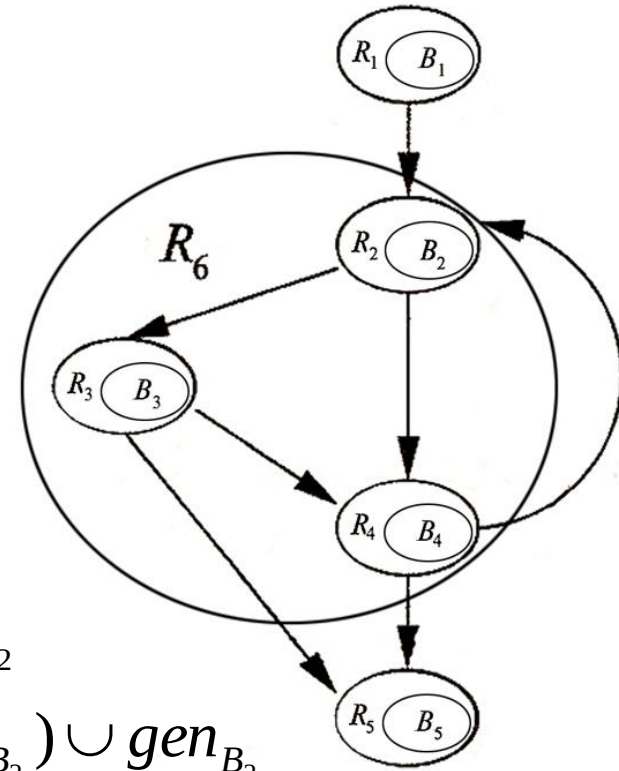
$$gen_{R_6, Out[R_2]} = gen_{B_2} = \{d_4\}, kill_{R_6, Out[R_2]} = kill_{B_2} = \{d_1\}$$

10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2b)**: вычисление передаточной функции области-тела R_6

Подобласть R_3 : у R_3 в пределах R_6 один предшественник R_2 . Следовательно



$$f_{R_6, In[R_3]} = f_{R_6, Out[B_2]} = f_{B_2}$$

$$f_{R_6, Out[B_3]} = f_{R_3, Out[B_3]} \circ f_{R_6, In[R_3]} = f_{B_3} \circ f_{B_2}$$

$$f_{R_6, In[R_3]}(x) = f_{R_6, Out[B_2]}(x) = f_{B_2}(x) = (x - kill_{B_2}) \cup gen_{B_2}$$

$$f_{R_6, Out[B_3]}(x) = f_{R_3, Out[B_3]} \circ f_{R_6, In[R_3]}(x) = (f_{B_3} \circ f_{B_2})(x) =$$

$$(gen_{B_3} \cup (gen_{B_2} - kill_{B_3})) \cup (x - (kill_{B_3} \cup kill_{B_2}))$$

$$gen_{R_6, In[B_3]} = gen_{B_2} = \{d_4\}, kill_{R_6, In[B_3]} = kill_{B_2} = \{d_1\}$$

$$gen_{R_6, Out[B_3]} = gen_{B_3} \cup (gen_{B_2} - kill_{B_3}) = \{d_5\} \cup (\{d_4\} - \{d_3\}) = \{d_4, d_5\}$$

$$kill_{R_6, Out[B_3]} = kill_{B_3} \cup kill_{B_2} = \{d_3\} \cup \{d_1\} = \{d_1, d_3\}$$

10.2. Анализ потока данных на основе областей

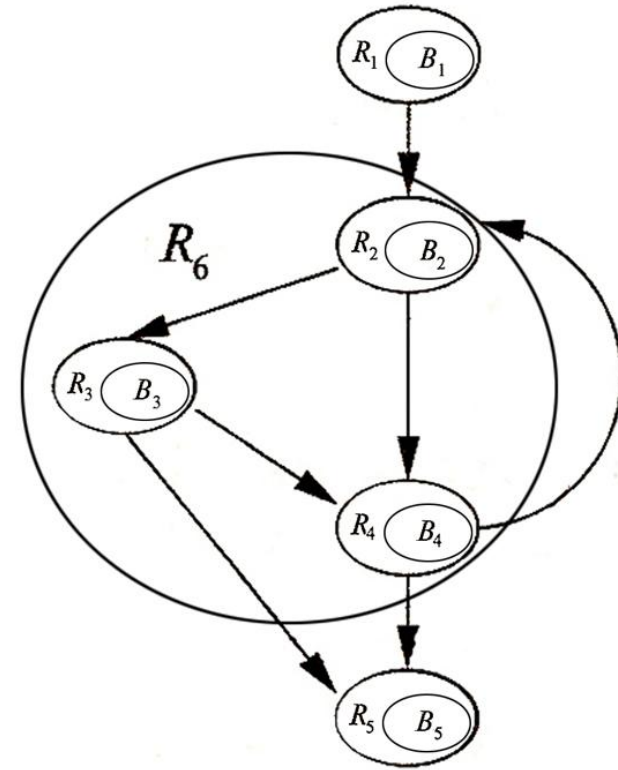
10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2b)**: вычисление передаточной функции области-тела R_6

Подобласть R_4 : у R_4 в пределах R_6 два предшественника R_2 и R_3 . Следовательно

$$\begin{aligned} f_{R_6, In[R_4]} &= f_{R_6, Out[B_2]} \wedge f_{R_6, Out[B_3]} = \\ &= f_{B_2} \wedge f_{R_6, Out[B_3]} \end{aligned}$$

$$\begin{aligned} f_{R_6, Out[B_4]} &= f_{R_4, Out[B_4]} \circ f_{R_6, In[R_4]} = \\ &= f_{B_4} \circ f_{R_6, In[R_4]} \end{aligned}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2b)**: вычисление передаточной функции области-тела R_6

Подобласть R_4

$$\begin{aligned} f_{R_6, In[R_4]}(X) &= (f_{R_6, Out[B_2]} \wedge f_{R_6, Out[B_3]})(X) = \\ &= (f_{B_2} \wedge f_{R_6, Out[B_3]})(X) \end{aligned}$$

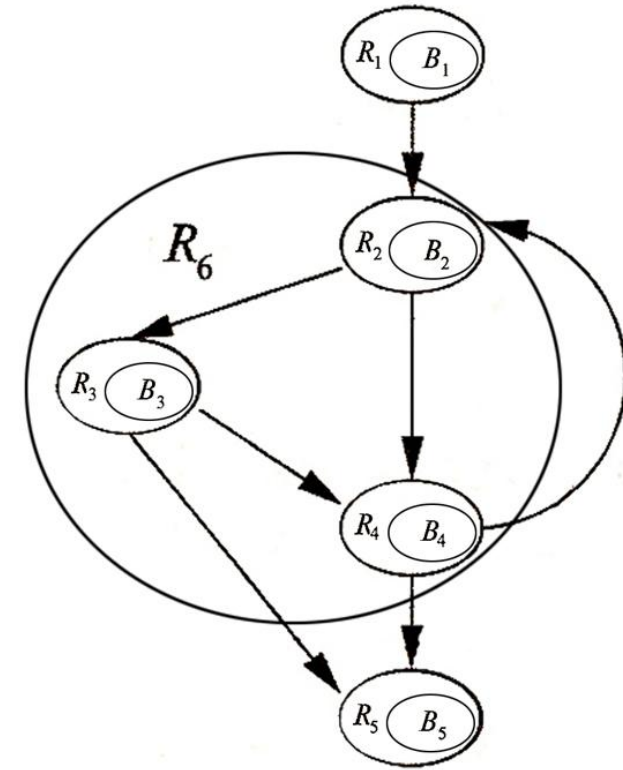
$$\begin{aligned} f_{R_6, Out[B_4]}(X) &= (f_{R_4, Out[B_4]} \circ f_{R_6, In[R_4]})(X) = \\ &= (f_{B_4} \circ f_{R_6, In[R_4]})(X) \end{aligned}$$

$$gen_{R_6, In[R_4]} = gen_{B_2} \cup gen_{R_6, Out[B_3]} = \{d_4\} \cup \{d_4, d_5\} = \{d_4, d_5\}$$

$$kill_{R_6, In[R_4]} = kill_{B_2} \cap kill_{R_6, Out[B_3]} = \{d_1\} \cap \{d_1, d_3\} = \{d_1\}$$

$$\begin{aligned} gen_{R_6, Out[R_4]} &= gen_{R_6, In[R_4]} \cup (gen_{B_4} - kill_{R_6, In[R_4]}) = \\ &= \{d_4, d_5\} \cup (\{d_6\} - \{d_1\}) = \{d_4, d_5, d_6\} \end{aligned}$$

$$kill_{R_6, Out[R_4]} = kill_{B_4} \cup kill_{R_6, In[R_4]} = \{d_2\} \cup \{d_1\} = \{d_1, d_2\}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2с):** вычисление передаточной функции области-цикла R_7

Область R_7 содержит только одну подобласть R_6 (тело цикла). Обратному ребру $B_4 \rightarrow B_2$ к заголовку R_6 соответствует передаточная функция

$$f_{R_7, In[R_6]} = f_{R_6, Out[B_4]}^*$$

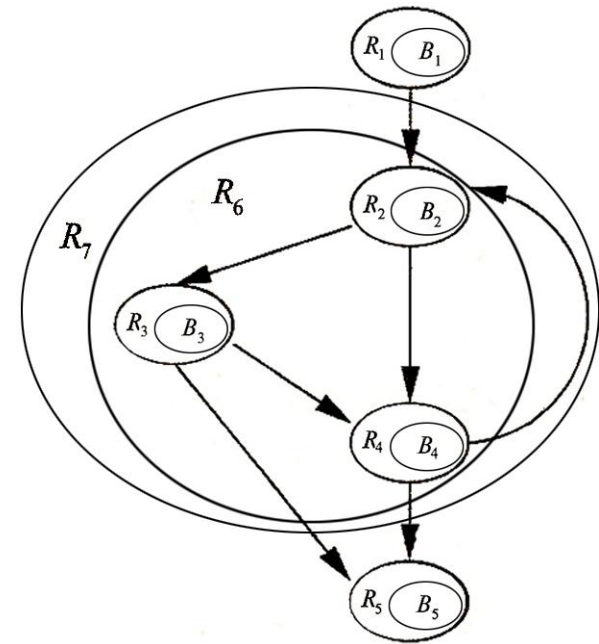
Из области R_7 имеется два выхода – базовые блоки B_3 и B_4 .

Поэтому для получения соответствующих передаточных функций R_7 нужно вычислить композиции $f_{R_7, In[R_6]}$ с $f_{R_6, Out[B_3]}$ и с $f_{R_6, Out[B_4]}$

:

$$f_{R_7, Out[B_4]} = f_{R_6, Out[B_4]} \circ f_{R_7, In[R_6]}$$

$$f_{R_7, Out[B_3]} = f_{R_6, Out[B_3]} \circ f_{R_7, In[R_6]}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2с):** вычисление передаточной функции области-цикла R_7

(а) На входе

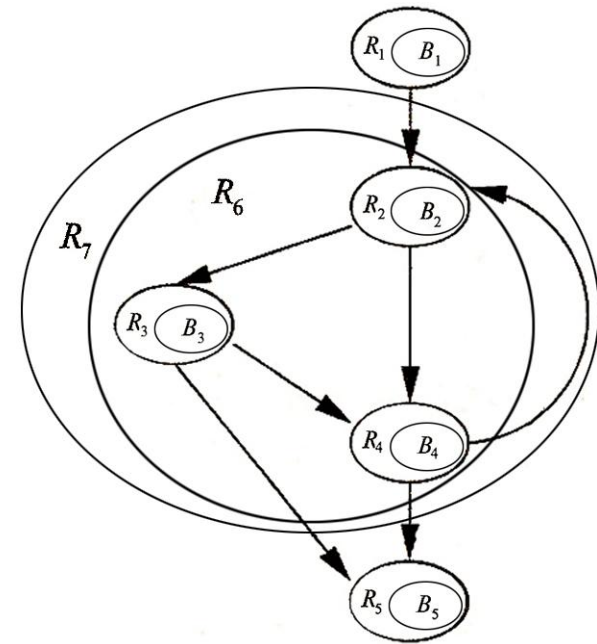
Передаточная функция:

$$f_{R_7, In[R_6]}(x) = f_{R_6, Out[B_4]}^*(x)$$

Множества *gen* и *kill*

$$gen_{R_7, In[R_6]} = gen_{R_6, Out[B_4]} = \{d_4, d_5, d_6\}$$

$$kill_{R_7, In[R_6]} = \emptyset$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2с):** вычисление передаточной функции области-цикла R_7

(b) На выходах

Передаточные функции:

$$f_{R_7, Out[B_4]}(x) = (f_{R_6, Out[B_4]} \circ f_{R_7, In[R_6]})(x)$$

$$f_{R_7, Out[B_3]}(x) = (f_{R_6, Out[B_3]} \circ f_{R_7, In[R_6]})(x)$$

Множества *gen* и *kill*

$$gen_{R_7, Out[B_4]} = gen_{R_7, In[R_6]} \cup (gen_{R_6, Out[B_4]} - kill_{R_7, In[R_6]}) =$$

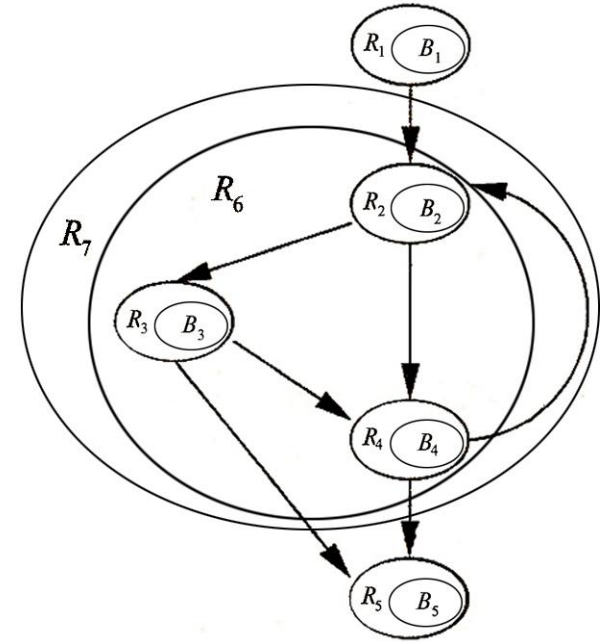
$$= \{d_4, d_5, d_6\} \cup (\{d_4, d_5, d_6\} - \emptyset) = \{d_4, d_5, d_6\}$$

$$kill_{R_7, Out[B_4]} = kill_{R_7, In[R_6]} \cup kill_{R_6, Out[B_4]} = \emptyset \cup \{d_1, d_2\} = \{d_1, d_2\}$$

$$gen_{R_7, Out[B_3]} = gen_{R_7, In[R_6]} \cup (gen_{R_6, Out[B_3]} - kill_{R_7, In[R_6]}) =$$

$$= \{d_4, d_5, d_6\} \cup (\{d_4, d_5\} - \emptyset) = \{d_4, d_5, d_6\}$$

$$kill_{R_7, Out[B_3]} = kill_{R_7, In[R_6]} \cup kill_{R_6, Out[B_3]} = \emptyset \cup \{d_1, d_3\} = \{d_1, d_3\}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2d)**: вычисление передаточной функции области-тела R_8

Подобластями области R_8 (весь граф потока) являются (в топологическом порядке)

R_1 , R_7 и R_5 .

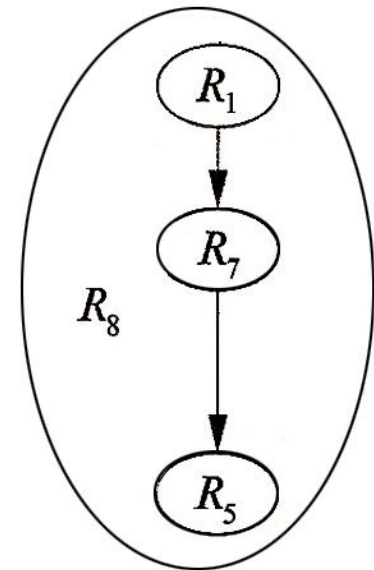
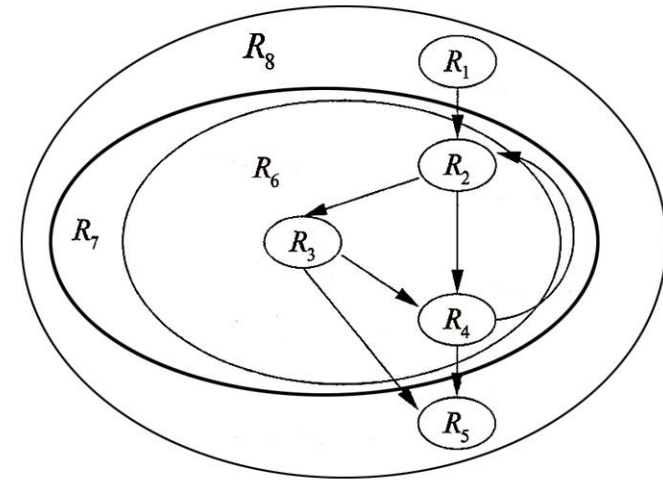
Передаточные функции:

1) для R_1 :

$$f_{R_8, In[R_1]}(x) = I(x) = x$$

$$f_{R_8, Out[B_1]}(x) = f_{B_1}(x)$$

$$gen_{R_8, In[R_1]} = kill_{R_8, In[R_1]} = \emptyset$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2d)**: вычисление передаточной функции области-цикла R_8

2) для R_7 :

Заголовок R_7 (блок B_2) имеет единственного предшественника, B_1 .

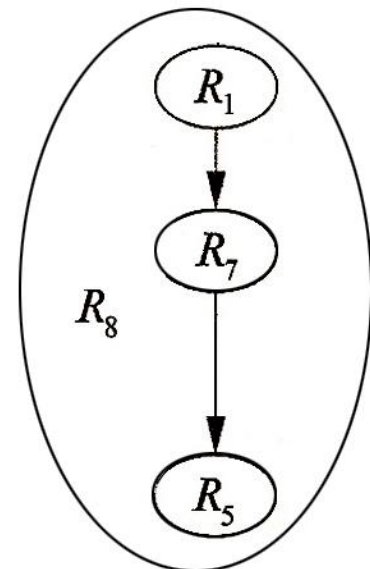
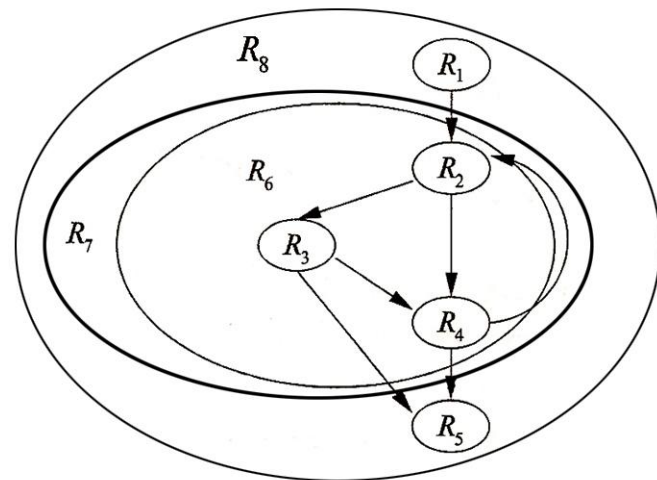
У R_7 два выходных ребра (в блоках B_3 и B_4).

Передаточные функции:

$$f_{R_8, In[R_7]} = f_{R_8, Out[B_1]} = f_{R_1, Out[B_1]} = f_{B_1}$$

$$f_{R_8, Out[B_3]} = f_{R_7, Out[B_3]} \circ f_{R_8, In[R_7]} = f_{R_7, Out[B_3]} \circ f_{B_1}$$

$$f_{R_8, Out[B_4]} = f_{R_7, Out[B_4]} \circ f_{R_8, In[R_7]} = f_{R_7, Out[B_4]} \circ f_{B_1}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2d)**: вычисление передаточной функции области-цикла R_8

2) для R_7 :

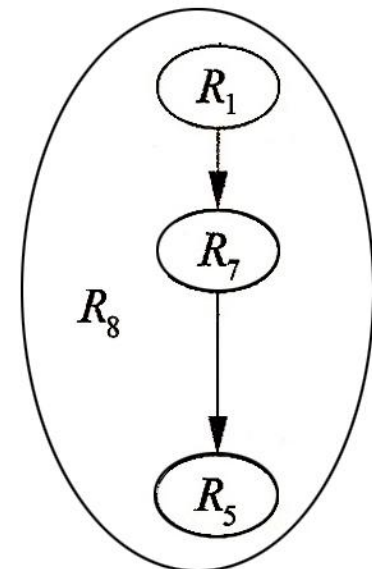
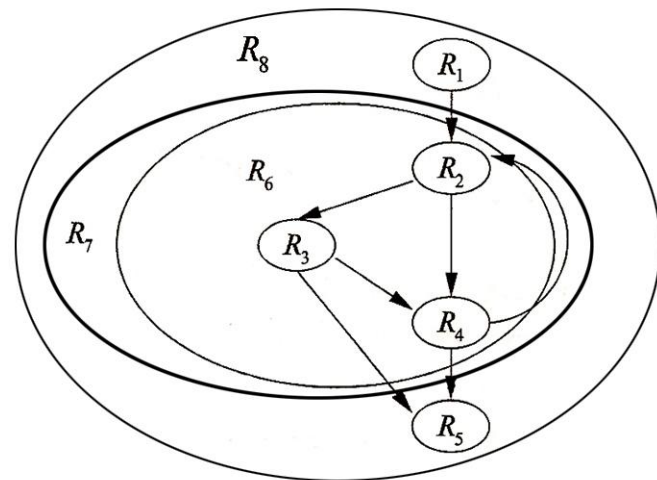
Множества *gen* и *kill*

$$gen_{R_8, In[R_7]} = gen_{B_1} = \{d_1, d_2, d_3\}$$

$$kill_{R_8, In[R_7]} = kill_{B_1} = \{d_4, d_5, d_6\}$$

$$\begin{aligned} gen_{R_8, Out[B_3]} &= gen_{R_7, Out[B_3]} \cup (gen_{B_1} - kill_{R_7, Out[B_3]}) = \\ &= \{d_4, d_5, d_6\} \cup (\{d_1, d_2, d_3\} - \{d_1, d_3\}) = \{d_2, d_4, d_5, d_6\} \end{aligned}$$

$$\begin{aligned} kill_{R_8, Out[B_3]} &= kill_{R_7, Out[B_3]} \cup kill_{B_1} = \\ &= \{d_1, d_3\} \cup \{d_4, d_5, d_6\} = \{d_1, d_3, d_4, d_5, d_6\} \end{aligned}$$



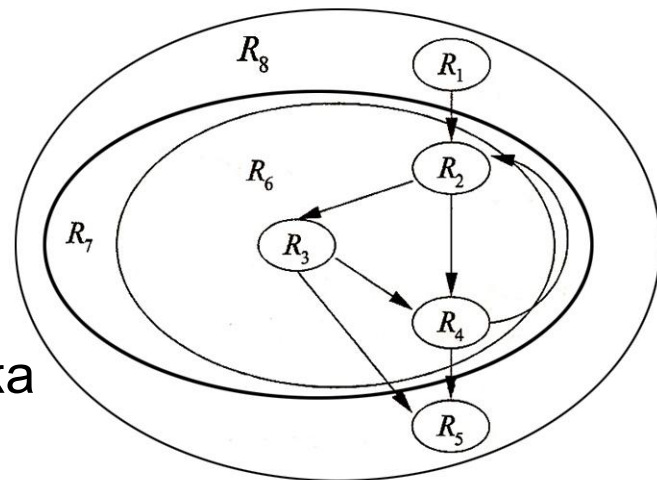
10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 2d)**: вычисление передаточной функции области-цикла R_8

3) для R_5 :

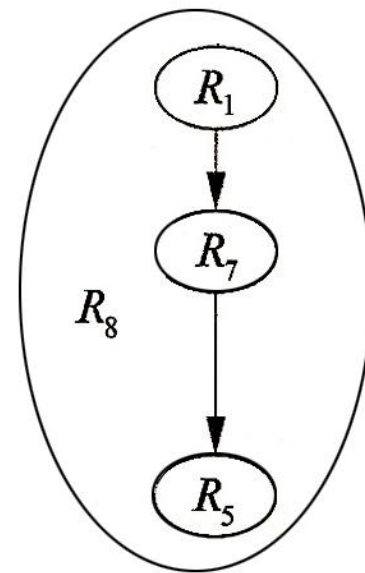
У заголовка R_5 (блок B_5) два предшественника (B_3 и B_4), причем $f_{B_5} = I$



$$f_{R_8, In[R_7]} = f_{R_8, Out[B_1]} = f_{R_1, Out[B_1]} = f_{B_1}$$

$$\begin{aligned} f_{R_8, Out[B_3]} &= f_{R_7, Out[B_3]} \circ f_{R_8, In[R_7]} = \\ &= f_{R_7, Out[B_3]} \circ f_{B_1} \end{aligned}$$

$$\begin{aligned} f_{R_8, Out[B_4]} &= f_{R_7, Out[B_4]} \circ f_{R_8, In[R_7]} = \\ &= f_{R_7, Out[B_4]} \circ f_{B_1} \end{aligned}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ Шаг 2d): вычисление передаточной функции области-цикла R_8

3) для R_5 :

передаточные функции

$$f_{R_8, In[B_5]}(x) = (f_{R_8, Out[B_3]} \wedge f_{R_8, Out[B_4]})(x)$$

$$f_{R_8, Out[B_5]}(x) = f_{R_8, In[B_5]}(x)$$

Множества *gen* и *kill* (см 10.2.2.3)

$$gen_{R_8, In[B_5]} = gen_{R_8, Out[B_3]} \cup gen_{R_8, Out[B_4]} =$$

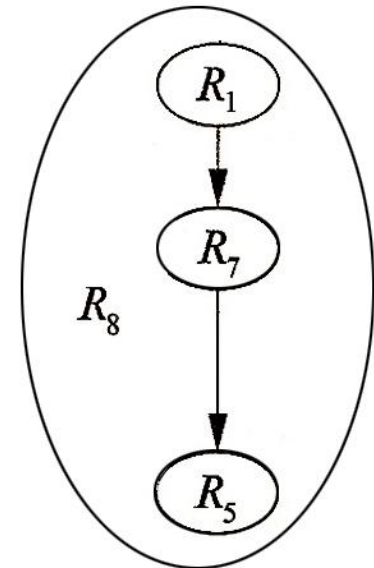
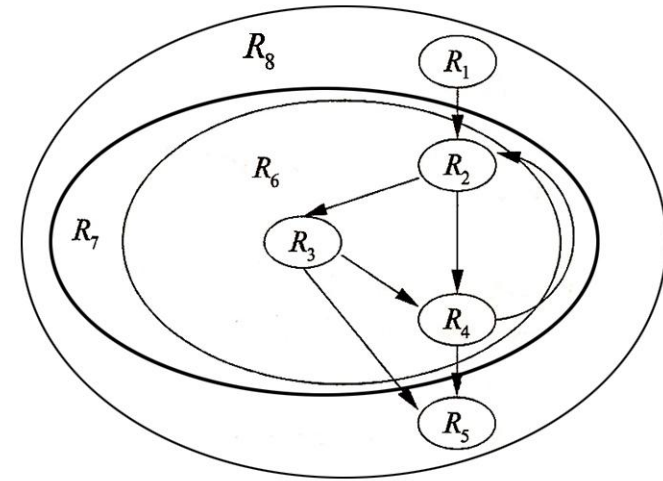
$$= \{d_2, d_4, d_5, d_6\} \cup \{d_3, d_4, d_5, d_6\} = \{d_2, d_3, d_4, d_5, d_6\}$$

$$kill_{R_8, In[B_5]} = kill_{R_8, Out[B_3]} \cap kill_{R_8, Out[B_4]} =$$

$$= \{d_1, d_3, d_4, d_5, d_6\} \cap \{d_1, d_2, d_4, d_5, d_6\} = \{d_1, d_4, d_5, d_6\}$$

$$gen_{R_8, Out[B_5]} = gen_{R_8, In[B_5]} = \{d_2, d_3, d_4, d_5, d_6\}$$

$$kill_{R_8, Out[B_5]} = kill_{R_8, In[B_5]} = \{d_1, d_4, d_5, d_6\}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ Результаты шага 2

	ПЕРЕДАТОЧНАЯ ФУНКЦИЯ	<i>gen</i>	<i>kill</i>
R_6	$f_{R_6, \text{IN}[R_2]} = I$ $f_{R_6, \text{OUT}[B_2]} = f_{R_2, \text{OUT}[B_2]} \circ f_{R_6, \text{IN}[R_2]}$ $f_{R_6, \text{IN}[R_3]} = f_{R_6, \text{OUT}[B_2]}$ $f_{R_6, \text{OUT}[B_3]} = f_{R_3, \text{OUT}[B_3]} \circ f_{R_6, \text{IN}[R_3]}$ $f_{R_6, \text{IN}[R_4]} = f_{R_6, \text{OUT}[B_2]} \wedge f_{R_6, \text{OUT}[B_3]}$ $f_{R_6, \text{OUT}[B_4]} = f_{R_4, \text{OUT}[B_4]} \circ f_{R_7, \text{IN}[R_6]}$	$\emptyset$ $\{d_4\}$ $\{d_4\}$ $\{d_4, d_5\}$ $\{d_4, d_5\}$ $\{d_4, d_5, d_6\}$	$\emptyset$ $\{d_1\}$ $\{d_1\}$ $\{d_1, d_3\}$ $\{d_1\}$ $\{d_1, d_2\}$
R_7	$f_{R_7, \text{IN}[R_6]} = f_{R_6, \text{OUT}[B_4]}^*$ $f_{R_7, \text{OUT}[B_3]} = f_{R_6, \text{OUT}[B_3]} \circ f_{R_7, \text{IN}[R_6]}$ $f_{R_7, \text{OUT}[B_4]} = f_{R_6, \text{OUT}[B_4]} \circ f_{R_7, \text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$ $\{d_4, d_5, d_6\}$ $\{d_4, d_5, d_6\}$	$\emptyset$ $\{d_1, d_3\}$ $\{d_1, d_2\}$
R_8	$f_{R_8, \text{IN}[R_1]} = I$ $f_{R_8, \text{OUT}[B_1]} = f_{R_1, \text{OUT}[B_1]}$ $f_{R_8, \text{IN}[R_7]} = f_{R_8, \text{OUT}[B_1]}$ $f_{R_8, \text{OUT}[B_3]} = f_{R_7, \text{OUT}[B_3]} \circ f_{R_8, \text{IN}[R_7]}$ $f_{R_8, \text{OUT}[B_4]} = f_{R_7, \text{OUT}[B_4]} \circ f_{R_8, \text{IN}[R_7]}$ $f_{R_8, \text{IN}[R_5]} = f_{R_8, \text{OUT}[B_3]} \wedge f_{R_8, \text{OUT}[B_4]}$ $f_{R_8, \text{OUT}[B_5]} = f_{R_5, \text{OUT}[B_5]} \circ f_{R_8, \text{IN}[R_5]}$	$\emptyset$ $\{d_1, d_2, d_3\}$ $\{d_1, d_2, d_3\}$ $\{d_2, d_4, d_5, d_6\}$ $\{d_3, d_4, d_5, d_6\}$ $\{d_2, d_3, d_4, d_5, d_6\}$ $\{d_2, d_3, d_4, d_5, d_6\}$	$\emptyset$ $\{d_4, d_5, d_6\}$ $\{d_4, d_5, d_6\}$ $\{d_1, d_3, d_4, d_5, d_6\}$ $\{d_1, d_2, d_4, d_5, d_6\}$ $\{d_1, d_4, d_5, d_6\}$ $\{d_1, d_4, d_5, d_6\}$

10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 3:** *Нисходящий просмотр* для вычисления значений $In[R]$ в начале каждой области:

1) Для области R_8
 $In[R_8] = \emptyset$, (граничное условие).

2) Для подобластей области R_8 :

2.1) область R_1

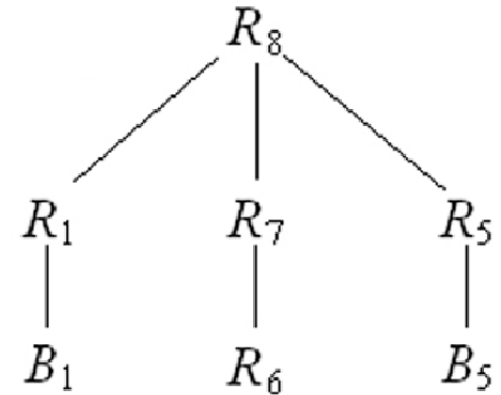
$$In[R_1] = f_{R_8, In[R_1]}(In[R_8]) = I(In[R_8]) = In[R_8] = \emptyset$$

2.2) область R_7

$$\begin{aligned} In[R_7] &= f_{R_8, In[R_7]}(In[R_8]) = f_{B_1}(In[R_8]) = gen_{B_1} \cup (In[R_8] - kill_{B_1}) = \\ &= (\emptyset - \{d_4, d_5, d_6\}) \cup \{d_1, d_2, d_3\} = \{d_1, d_2, d_3\} \end{aligned}$$

2.3) область R_5

$$\begin{aligned} In[R_5] &= f_{R_8, In[R_5]}(In[R_8]) = gen_{R_8, In[R_5]} \cup (In[R_8] - kill_{R_8, In[R_5]}) \\ &= \{d_2, d_3, d_4, d_5, d_6\} \cup (\emptyset - \{d_1\}) = \{d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$



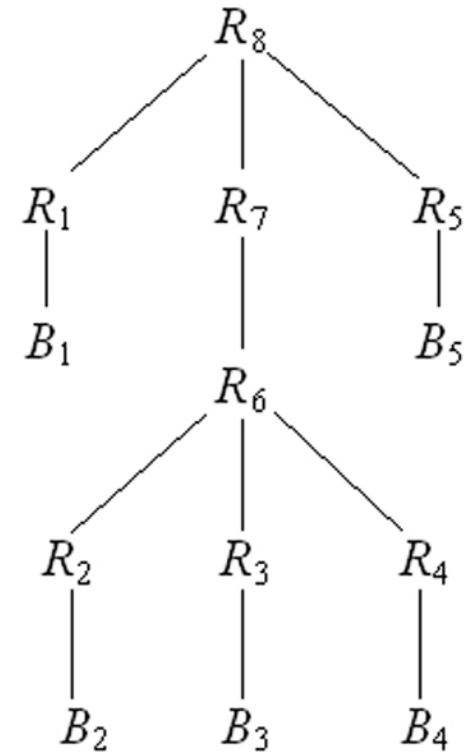
10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 3:** *Нисходящий просмотр* для вычисления значений $In[R]$ в начале каждой области:

3) Для подобласти R_6 области R_7 :

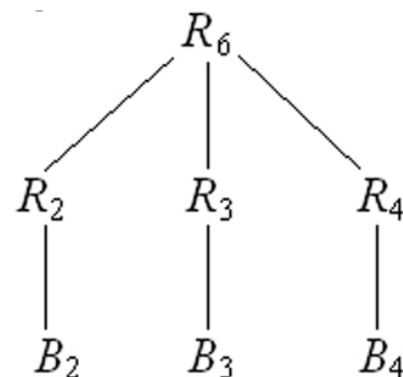
$$\begin{aligned} In[R_6] &= f_{R_7, In[R_6]}(In[R_7]) = \\ &= gen_{R_7, In[R_6]} \cup (In[R_7] - kill_{R_7, In[R_6]}) = \\ &= \{d_4, d_5, d_6\} \cup (\{d_1, d_2, d_3\} - \emptyset) = \\ &= \{d_1, d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ **Шаг 3:** *Нисходящий просмотр* для вычисления значений $In[R]$ в начале каждой области:



4) Для подобластей области R_6 :

4.1) область R_4

$$\begin{aligned} In[R_4] &= f_{R_6, In[R_4]}(In[R_6]) = gen_{R_6, In[R_4]} \cup (In[R_6] - kill_{R_6, In[R_4]}) \\ &= \{d_4, d_5\} \cup (\{d_1, d_2, d_3, d_4, d_5, d_6\} - \{d_1\}) = \{d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$

4.2) область R_3

$$\begin{aligned} In[R_3] &= f_{R_6, In[R_3]}(In[R_6]) = gen_{R_6, In[R_3]} \cup (In[R_6] - kill_{R_6, In[R_3]}) \\ &= \{d_4, d_5\} \cup (\{d_1, d_2, d_3, d_4, d_5, d_6\} - \{d_1\}) = \{d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$

4.3) область R_2

$$\begin{aligned} In[R_2] &= f_{R_6, In[R_2]}(In[R_6]) = gen_{R_6, In[R_2]} \cup (In[R_6] - kill_{R_6, In[R_2]}) \\ &= \emptyset \cup (\{d_1, d_2, d_3, d_4, d_5, d_6\} - \emptyset) = \{d_1, d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$

10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

♦ Результаты шага 3

Значения потока данных для достигающих определений:

$$In[R_8] = \emptyset$$

$$In[R_1] = f_{R_8, In[R_1]}(In[R_8]) = \emptyset$$

$$In[R_7] = f_{R_8, In[R_7]}(In[R_8]) = \{d_1, d_2, d_3\}$$

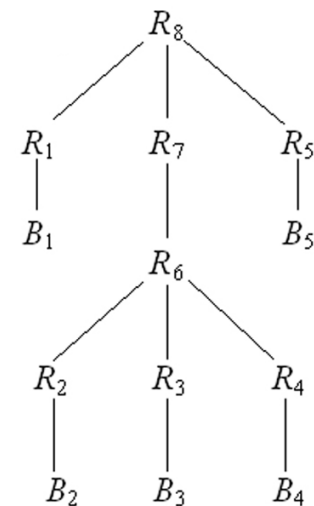
$$In[R_5] = f_{R_8, In[R_5]}(In[R_8]) = \{d_2, d_3, d_4, d_5, d_6\}$$

$$In[R_6] = f_{R_7, In[R_6]}(In[R_7]) = \{d_1, d_2, d_3, d_4, d_5, d_6\}$$

$$In[R_4] = f_{R_6, In[R_4]}(In[R_6]) = \{d_2, d_3, d_4, d_5, d_6\}$$

$$In[R_3] = f_{R_6, In[R_3]}(In[R_6]) = \{d_2, d_3, d_4, d_5, d_6\}$$

$$In[R_2] = f_{R_6, In[R_2]}(In[R_6]) = \{d_1, d_2, d_3, d_4, d_5, d_6\}$$



10.2. Анализ потока данных на основе областей

10.2.4. Пример: применение алгоритма 10.2.3

◇ Сравнение результатов

Множества определений, достигающих входов в блоки B_1, B_2, B_3, B_4, B_5 , вычисленные с помощью алгоритма 10.2.3:

$$\begin{aligned} In[B_1] &= In[R_1] = \emptyset \\ In[B_2] &= In[R_2] = \{d_1, d_2, d_3, d_4, d_5, d_6\} \\ In[B_3] &= In[R_3] = \{d_2, d_3, d_4, d_5, d_6\} \\ In[B_4] &= In[R_4] = \{d_2, d_3, d_4, d_5, d_6\} \\ In[B_5] &= In[R_5] = \{d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$

Множества определений, достигающих входов в блоки B_1, B_2, B_3, B_4, B_5 , вычисленные без выделения областей (слайд 49):

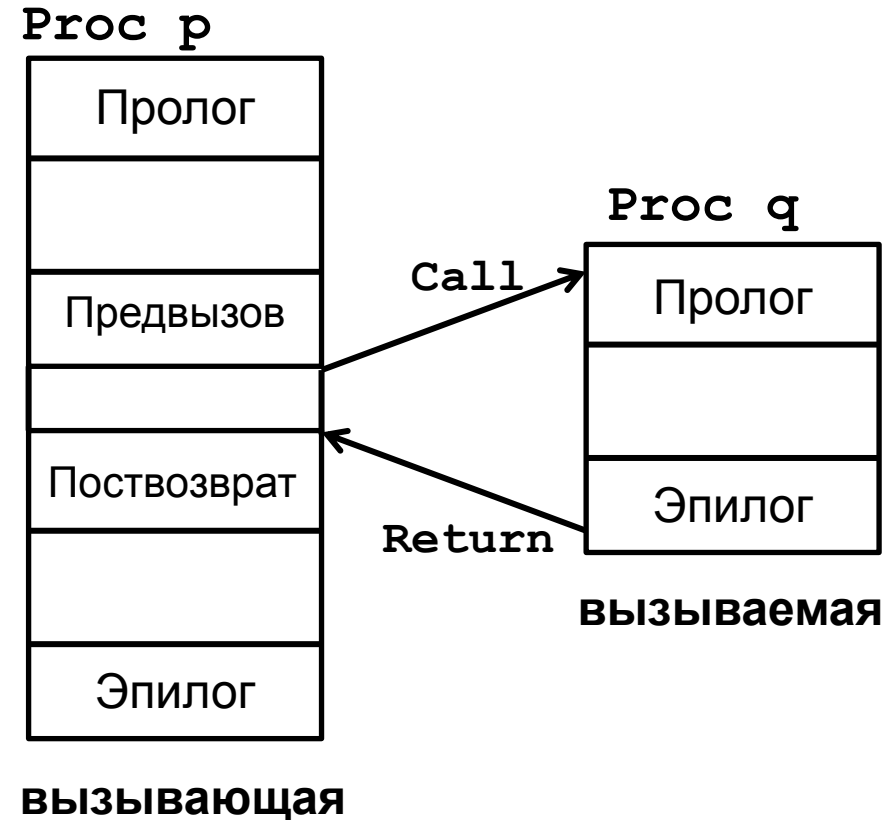
$$\begin{aligned} In[B_1] &= \emptyset \\ In[B_2] &= \{d_1, d_2, d_3, d_4, d_5, d_6\} \\ In[B_3] &= \{d_2, d_3, d_4, d_5, d_6\} \\ In[B_4] &= \{d_2, d_3, d_4, d_5, d_6\} \\ In[B_5] &= \{d_2, d_3, d_4, d_5, d_6\} \end{aligned}$$

11. Открытая вставка

11.1 Постановка задачи

11.1.1 Вызов процедуры

- ◇ На схеме показана типичная реализация вызова процедуры **q** из процедуры **p**.
- ◇ Процедура может иметь несколько *точек вызова*.
- ◇ Общее правило: перенос операторов из "Предвызова" и "Поствозврата" вызывающей соответственно в пролог и эпилог вызываемой позволяет сократить общий размер результирующего кода, так как в случае нескольких вызовов одной и той же процедуры приводит к замене нескольких операций копирования в предвызовах одной операцией копирования в прологе.



11.1 Постановка задачи

11.1.1 Вызов процедуры



Предвызов:

1. Вычисление фактических параметров
2. Если параметр, вызываемый по ссылке, находится на регистре, он загружается в память вызывающей и передает соответствующий адрес вызываемой
3. Определение адреса возврата и, в случае необходимости адресов переменных для приема возвращаемых значений.
Если параметров меньше k , они передаются на регистрах. Если параметров больше k , первые k параметров передаются на регистрах, а остальные помещаются в *буфер обмена*, расположенный либо в памяти вызываемой, либо в памяти вызывающей.



Поствозврат:

1. Запоминает значения, возвращенные на регистрах.
2. В случае необходимости освобождает динамическую память вызываемой.

11.1 Постановка задачи

11.1.1 Вызов процедуры



Пролог:

1. Выделяет память для хранения значений фактических параметров, передаваемых вызывающей на регистрах, а также для локальных переменных вызываемой.
2. Размещает и инициализирует локальные переменные.



Эпилог:

1. Выделяет память для возвращаемых значений.
2. Размещает возвращаемые значения.
3. Передает управление по адресу возврата.

Предвызов, поствозврат, пролог и эпилог – **накладные расходы на вызов процедуры**

11.1 Постановка задачи

11.1.1 Вызов процедуры



Пролог:

1. Выделяет память для хранения значений фактических параметров, передаваемых вызывающей на регистрах, а также для локальных переменных вызываемой.
2. Размещает и инициализирует локальные переменные.



Эпилог:

1. Выделяет память для возвращаемых значений.
2. Размещает возвращаемые значения.
3. Передает управление по адресу возврата.

Предвызов, поствозврат, пролог и эпилог – **накладные расходы на вызов процедуры**

А тело процедуры **`sin(x)`** занимает 4 строчки.

11.1 Постановка задачи

11.1.2 Деление программы на процедуры. Недостатки

- ◇ Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ.
- ◇ Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:
 - ◇ **Накладные расходы на вызов процедур** (прологи, эпилоги и т.п.) существенно снижают эффективность (быстродействие) программы.
 - ◇ **Ограничение возможностей компилятора** понимать, что происходит внутри вызова.

Рассмотрим фрагмент процедуры

```
x = 15;  
p = &x;  
m = n + x;  
q = f(p, m + n);  
m = n + x;
```

Вопрос: можно ли исключить одно из присваиваний
`m = n + x;`?

11.1 Постановка задачи

11.1.2 Деление программы на процедуры. Недостатки

- ◇ Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ.
- ◇ Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:

^ Нежелательное расхождение вызовов процедур (проблема)

- ◇ Для преодоления первого недостатка используется **открытая вставка**.

- ◇ **Ограничение возможностей компилятора** понимать, что происходит внутри вызова.

Рассмотрим, например, фрагмент процедуры

```
x = 15;  
p = &x;  
m = n + x;  
q = f(p, m + n);  
m = n + x;
```

Вопрос: можно ли исключить одно из присваиваний
`m = n + x;`?

11.1 Постановка задачи

11.1.2 Деление программы на процедуры. Недостатки

- ◇ Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ.
- ◇ Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:

^ Нежелательное взаимодействие вызовов процедур (проблема)

- ◇ Для преодоления первого недостатка используется **открытая вставка**.

- ◇ **Ограничение возможностей компилятора** понимать, что происходит внутри вызова.

Рассмотрим фрагмент процедуры **g**

x = 15;

- ◇ Для преодоления второго недостатка используется межпроцедурный анализ (совместный анализ процедур **g** и **f**).

m = n + x;

Вопрос: можно ли исключить одно из присваиваний

m = n + x;?

11.2 Открытая вставка

11.2.1 Что это такое

- ◇ Вызовы процедур в среднем составляют примерно 5% всех инструкций программы, причем вызов процедуры – самый дорогой оператор программы
- ◇ Оптимизация состоит в том, что **вызов процедуры исключается из программы и вместо него в программу вставляется тело вызываемой процедуры**, а также ДВЕ группы операторов присваивания, чтобы совместить контекст вызываемой процедуры с контекстом вызывающей.

11.2 Открытая вставка

11.2.2 Как выполняется открытая вставка

- ◇ Открытая вставка это очень просто:
 - ◇ заменить вызов на тело вызываемой процедуры
 - ◇ превратить передачу параметров и возврат результатов в инструкции присваивания
 - ◇ выполнить оптимизацию "распространение копий", чтобы исключить лишние переменные
 - ◇ правильно разобраться с областями видимости переменных, переименовывая переменные в случае необходимости (для программ в промежуточном представлении MIR это не требуется)

11.2 Открытая вставка

11.2.3 Преимущества и недостатки открытой вставки

◇ **Преимущества:**

- ◇ исключаются накладные расходы на организацию вызова процедуры и возврата из нее
- ◇ исключаются накладные расходы на передачу параметров и возврат результатов
- ◇ появляется возможность оптимизации вызываемой процедуры с учетом контекста вызывающей (открытая вставка как один из способов реализации межпроцедурного анализа)

◇ **Недостатки:**

- ◇ могут возрасти требования скомпилированного кода по памяти
- ◇ может замедлиться компиляция
- ◇ сложности с рекурсией

11.2 Открытая вставка

11.2.4 Виртуальная открытая вставка

- ◇ Виртуальная открытая вставка: вставка моделируется (симулируется) во время анализа вызывающей процедуры, **но фактически не выполняется.**

11.2 Открытая вставка

11.2.4 Виртуальная открытая вставка

- ◇ Виртуальная открытая вставка: вставка моделируется (симулируется) во время анализа вызывающей процедуры, **но фактически не выполняется.**
- ◇ Ответ на вопрос: "Как изменится процедура **P**, если в нее будет вставлена процедура **A**?" и другие схожие вопросы.

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

- ◇ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:
 - ◇ размеры вызывающей и вызываемой процедур (легко вычислить размеры **до** открытой вставки а что можно сказать о размерах **после** открытой вставки?)
 - ◇ частота вызова (статические оценки или динамическое профилирование)
 - ◇ вызовы, при которых вызываемая процедура существенно улучшается после совместной оптимизации (неясно, как это оценить численно)
 - ◇ **аннотации** (наборы соотношений)
 - ◆ вызываемой процедуры (в начале вызываемой процедуры)
 - ◆ контекста в точке вызова вызываемой процедуры

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

◇ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:

◇ размеры вызывающей и вызываемой процедур

Что такое профилирование? *Профилирование* – это численная оценка времени выполнения каждого фрагмента процедуры (например, базового блока, области, всей процедуры).

В результате профилирования получается *временной*, или *частотный*, или еще какой-нибудь *профиль* процедуры.

для каждого места вызова процедуры (например, как это оценить численно)

◇ **аннотации** (наборы соотношений)

♦ вызываемой процедуры (в начале вызываемой процедуры)

♦ контекста в точке вызова вызываемой процедуры

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

- ◇ Как принимаются решения, какие вызовы лучше заменить открытой вставкой?

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Инструментирование – в различные точки процедуры вставляются вызовы соответствующего инструмента (счетчика количества обращений к фрагменту, секундомера и т.п.), выбираются наборы исходных данных, обеспечивающие достаточно полное покрытие процедуры и исследуемая процедура запускается на этих наборах, в результате чего получается требуемый профиль.

- ◇ вызываемой процедуры (в начале вызываемой процедуры)
- ◇ контекста в точке вызова вызываемой процедуры

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

- ◇ Как принимаются решения, какие вызовы лучше заменить открытой вставкой?

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Семплирование – во время выполнения процедуры профилировщик периодически спрашивает у VM (или у ОС) как выглядят трассы всех потоков и, получив ответ, соответственно обновляет свою статистику. Это, как правило меньше замедляет профилируемую программу, но дает менее точный профиль.

- ◇ **аннотации** (наборы соотношений)
 - ◆ вызываемой процедуры (в начале вызываемой процедуры)
 - ◆ контекста в точке вызова вызываемой процедуры

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

- ◇ Как принимаются решения, какие вызовы лучше заменить открытой вставкой?

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Семплирование – во время выполнения процедуры профилировщик периодически спрашивает у VM (или у ОС) как выглядят трассы всех потоков и, получив ответ, соответственно обновляет свою статистику.

Это, как правило, дает менее точные результаты, но

How to Profile a C program in Linux using GNU gprof

- ◇ а (<https://www.maketecheasier.com/profile-c-program-linux-using-gprof/>)

ваемой

процедуры)

- ◇ контекста в точке вызова вызываемой процедуры

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

- ◇ Как принимаются решения, какие вызовы лучше заменить открытой вставкой?

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Семплирование – во время выполнения процедуры профилировщик периодически спрашивает у VM (или у ОС) как выглядят трассы всех потоков и, получив ответ, соответственно обновляет свою статистику.

Это, как правило, **How to Profile a C program in Linux** тому, но

дает менее Средство профилирования **Windows**



называется **Performance Counters**

(см. сайт <https://docs.microsoft.com/en-us/windows/desktop/PerfCtrs/about-performance-counters>)

[gram-](#)

ваемой

емой процедуры

11.3 Открытая вставка при компиляции С-программ

11.3.1 Когда следует выполнять открытую вставку

- ◇ Открытую вставку нужно выполнять **прежде** остальных оптимизаций, таких как сворачивание констант и исключение мертвого кода. Поэтому естественно выполнять открытую вставку во время компиляции.
- ◇ Открытая вставка может выполняться как на фазе построения АСД (в рамках переднего плана), так и над программой, представленной в машинно-независимом трехадресном коде (в виде последовательности "четверок").
- ◇ Открытая вставка требует знания характеристик всех вызовов функций, включая библиотечные, что ограничивает возможности открытой вставки при раздельной компиляции.
Для С-программ последнее замечание ослабляется тем, что единица компиляции (модуль) обычно содержит определения нескольких процедур и объявления всех остальных (даже внешних) процедур, вызываемых в модуле.

11.3 Открытая вставка при компиляции С-программ

11.3.1 Когда следует выполнять открытую вставку

- ◇ В последнее время открытую вставку нередко выполняют в самом конце процесса компиляции – во время связывания программ, так как на этом этапе доступны вообще все процедуры программы.
- ◇ Недостатком такого подхода является **необходимость повторной оптимизации программы после вставки**, так как только в этом случае можно получить все выгоды от открытой вставки.

11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки

- ◇ При открытой вставке используется представление оптимизируемой программы в виде взвешенного графа вызовов.
- ◇ **Взвешенный граф вызовов (ВГВ)** задается пятеркой

$$G = \langle N, p_N, E, p_E, main \rangle$$

N – множество вершин (представляющих процедуры программы), с каждой вершиной связан ее "вес"

p_N – целое число, равное, например, количеству инструкций соответствующей процедуры

E – множество ребер (каждое ребро соединяет вызываемую и вызывающую процедуры), с каждым ребром связан его "вес"

p_E – количество выполнений соответствующего вызова

$main$ – имя процедуры, выполняемой первой.

11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки

- ◇ При открытой вставке используется представление оптимизируемой программы в виде взвешенного графа вызовов.
- ◇ **Взвешенный граф вызовов (ВГВ)** задается пятеркой

$$G = \langle N, p_N, E, p_E, main \rangle$$

N – множество вершин (представляющих процедуры программы), с каждой вершиной связан ее "вес"

p_N – целое число, равное, например, количеству инструкций соответствующей процедуры

Обычно p_N и p_E определяются во время профилирования

вызываемых процедур, с каждым ребром связан его "вес"

p_E – количество выполнений соответствующего вызова

$main$ – имя процедуры, выполняемой первой.

11.3 Открытая вставка при компиляции С-программ

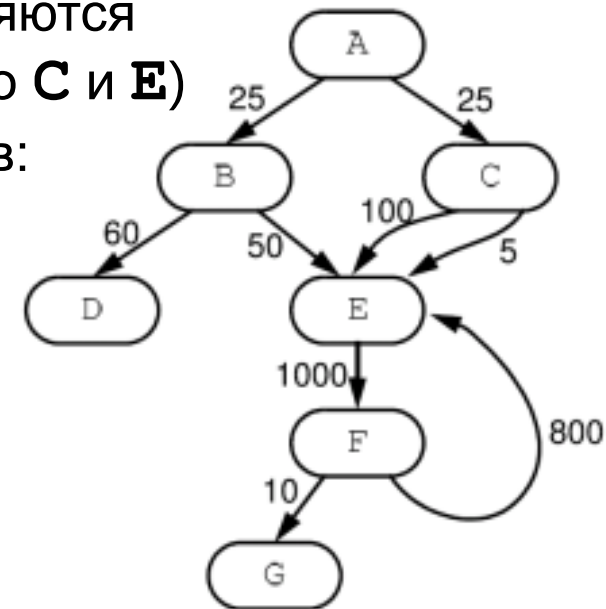
11.3.2 Представление программы для реализации открытой вставки

◇ Граф вызовов:

- ◇ вершины соответствуют процедурам
- ◇ дуги соответствуют вызовам процедур (дуги помечены числами, означающими либо количество, либо частоту вызовов)
- ◇ если процедура вызывается несколько раз, соответствующие вершины соединяются несколькими дугами (на рисунке это **С** и **Е**)

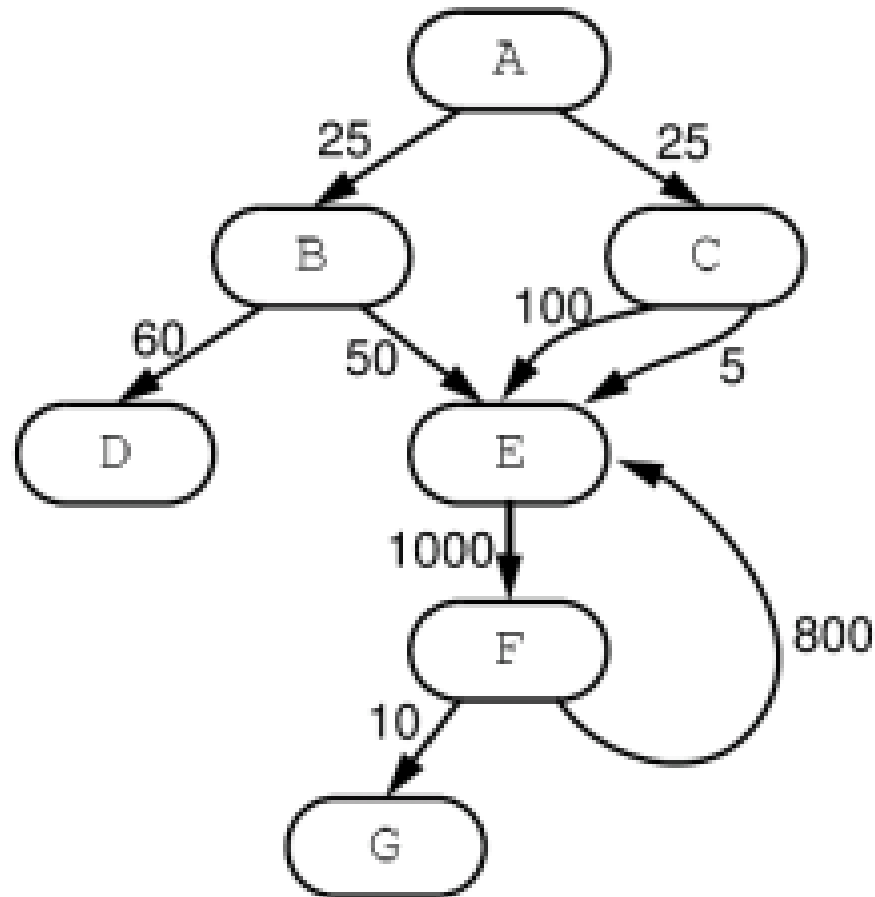
◇ Трудные случаи построения графа вызовов:

- ◇ вызовы внешних процедур
- ◇ вызовы из внешних процедур
- ◇ вызовы по указателям, значениям функций



11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки



11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки

$$G = \langle N, p_N, E, p_E, main \rangle$$

- ◇ Каждая **вершина** ВГВ содержит следующую информацию:
 - ◇ тело процедуры
 - ◇ вес
 - ◇ множество исходящих ребер
- ◇ Каждое **ребро** ВГВ содержит следующую информацию:
 - ◇ уникальный идентификатор ребра (для задания ребра недостаточно указать пару вершин, так как если вызов происходит несколько раз, существует несколько соответствующих ребер)
 - ◇ имя вызывающей процедуры
 - ◇ имя вызываемой процедуры
 - ◇ вес ребра
 - ◇ статус вызываемой процедуры (статус вычисляется в процессе анализа и имеет три значения: анализа не было, вставка запрещена, вставка выполнена)

11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки

$$G = \langle N, p_N, E, p_E, main \rangle$$

- ◇ **Вес** вершин и ребер можно определить:
 - ◇ либо с помощью структурного анализа ГПУ
 - ◇ либо с помощью профилирования
(мы рассматриваем два способа профилирования – инструментирование и семплирование)
- ◇ Войти в каждую вершину ВГВ можно через любое из входящих в нее ребер, поэтому необходимо знать веса всех исходящих ребер, соответствующих каждому входящему ребру.
В этом случае после выполнения открытой вставки веса ребер остаются правильными.
- ◇ Будет показано, что для эффективного выполнения открытой вставки необходимо иметь полный набор точных весов всех вершин и ребер ВГВ

11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки

$$G = \langle N, p_N, E, p_E, main \rangle$$

- ◇ Открытая вставка статического вызова состоит в удалении из ВГВ часто выполняемой дуги с одновременной вставкой тела вызываемой процедуры в тело вызывающей в соответствии с идентификатором дуги.
- ◇ Удаление дуги может привести к тому, что вызываемая процедура станет недостижимой в ВГВ. В этом случае мы вставили мертвый код, и его нужно удалить
- ◇ Когда вершина имеет ребро, которое является одновременно входящим и исходящим, имеет место рекурсивный вызов. В этом случае следует использовать стандартную технику обработки рекурсивных вызовов, заменяя их циклами.

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

- ◇ Казалось бы, выполняя открытую вставку, выгодно заменить как можно больше "тяжелых" ребер. К сожалению это не всегда так.
 - ◇ Необходимо избегать вставки слишком длинного кода (для этого следует установить верхнюю границу длины вставляемого кода)
 - ◇ Вставка кода, использующего слишком много места на стеке вызовов (например, рекурсия) может привести к быстрому переполнению стека. Этого тоже желательно избегать.

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

- ◇ **Взрывной рост объема кода**
Для того, чтобы развернуть вызов, необходимо сдублировать тело вызываемой процедуры и подставить полученную копию в вызывающую процедуру. Ясно, что в результате общий объем программы возрастает.
- ◇ Скорее всего, верно, что многие С-функции вызываются только один раз и поэтому копии оригинала этих один раз вызываемых функций можно удалить как недостижимые после их вставки. К сожалению, так удастся удалить не все недостижимые узлы в неполном ВГВ (Большая часть ВГВ больших процедур неполны, так как не могут учесть все системные вызовы). Поэтому необходимо установить верхнюю границу на размер области инструкций процедуры. Эта граница может быть задана либо как фиксированное целое число, либо как функция от размера процедуры.

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

◇ Переполнение стека вызовов.

Стек вызовов поддерживает передачу параметров, сохранение регистров, размещение локальных (автоматических) переменных, выделение памяти под возвращаемое значение, в связи с вызовом процедуры. Для каждой вызываемой процедуры нетрудно подсчитать размер необходимой для ее вызова области стека вызовов.

Переполнение стека вызовов может произойти при вставке рекурсивной процедуры, которой требуется очень большая область для поддержки рекурсивного вызова.

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

◇ Переполнение стека вызовов.

Например, пусть определены рекурсивная функция **m(x)** и еще одна функция **n(x)** :

```
m(x) {return(x ? m(x-1)+m(x-2)+n(x) : 1) ; }
```

```
n(x) {int y[100000] ; ...}
```

Когда **m(x)** вызывается для достаточно больших значений **x**, **n(x)** может существенно увеличить размер стека вызовов.

◇ Для предотвращения переполнения стека вызовов необходимо ввести фиксированный предельный размер области стека вызовов, необходимой для поддержки рекурсивного вызова.

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

◇ **Функция затрат**

Для заданного ВГВ функция затрат (ФЗ) предназначена для выявления наиболее подходящего ребра для открытой вставки. ФЗ позволяет сформулировать открытую вставку как оптимизационную задачу минимизации затрат на открытую вставку для произвольного количества вставок.

◇ Ввиду того, что ни реальные затраты на вставку, ни выгода от вставки неизвестны во время компиляции, получить оптимальные решения невозможно. Более того, пространство поиска для сформулированной оптимизационной задачи слишком велико для практической реализации.

Поэтому желательно использовать эвристику, позволяющую существенно сохранять небольшие размеры пространства поиска.

◇ При разработке функции затрат необходимо не только предотвратить рассмотренные опасности, связанные с переполнением, но и исключить из ВГВ несущественные ребра.

11.3 Открытая вставка при компиляции C-программ

11.3.3 Выбор вызова для замены

◇ Пример функции затрат

```
cost(G, arc Ai) =  
    if((caller is recursive) &&  
        (control_stack_usage(Ai) > BOUND) )  
    then cost = INFINITY;                // ∞  
    else  
        if(weight(Ai) < MIN)  
        then cost = INFINITY;  
        else  
            if(instruction_space_after_expansion  
                (G, Ai) > MAX)  
            then cost = INFINITY;  
            else  
                cost = Code_expansion_Cost of Ai -  
                    Benefits_of_Inlining Ai;
```

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

◇ Пример функции затрат

```
cost(G, arc Ai) =  
    if((caller is recursive) &&  
        (control_stack_usage(Ai) > BOUND) )  
    then cost = INFINITY; // ∞  
else  
    if INFINITY означает, что затраты на вставку непомерно  
        высокие, так что вставка нецелесообразна  
    then cost = INFINITY;  
else  
    Code expansion Cost - затраты на вставку  
    Benefits_of_Inlining - выгоды от вставки  
    then cost = INFINITY;  
else  
    cost = Code_expansion_Cost of Ai -  
        Benefits_of_Inlining Ai;
```

11.3 Открытая вставка при компиляции С-программ

11.3.3 Выбор вызова для замены

◇ Функция затрат

Затраты на вставку кода – это увеличение памяти, необходимой для размещения программы, и влияние на производительность кэш-памяти для инструкций. Точные значения этих величин не могут быть получены во время компиляции.

Грубая оценка затрат может быть получена путем умножения эмпирической константы на размер кода.

◇ Точные оценки выгоды от вставки также очень трудно получить. Если предположить, что

(1) при организации вызова функций превалирующими инструкциями являются инструкции загрузки и освобождения регистров (**Load** и **Store**), а также инструкции перехода,

и что

(2) эти затраты приблизительно равны для всех вызовов, член, связанный с оценкой выгоды может быть исключен из функции затрат.

11.3 Открытая вставка при компиляции С-программ

11.3.4. Вставка функции

- ◇ После того как определены функции для вставки и места, в которые их нужно вставить, выполняется **фаза физической вставки функций**.
Эта фаза состоит в решении трех основных задач:
 - 1) дублирование вызываемой функции
 - 2) переименование переменных
 - 3) модификация таблицы символов.
- ◇ Выполнение дублирования тривиально. Однако при его реализации бывает необходимо выполнять такие оптимизации реализуемого алгоритма как кэширование определений наиболее часто вставляемых функций, чтобы сократить количество чтений файлов.

11.3 Открытая вставка при компиляции С-программ

11.3.4. Вставка функции

- ◇ После того как определены функции для вставки и места, в которые их нужно вставить, выполняется **фаза физической вставки функций**.
Эта фаза состоит в решении трех основных задач:
 - 1) дублирование вызываемой функции
 - 2) переименование переменных
 - 3) модификация таблицы символов.
- ◇ Выполнение дублирования тривиально. Однако при его реализации бывает необходимо выполнять такие оптимизации реализуемого алгоритма как кэширование определений наиболее часто вставляемых функций, чтобы сократить количество чтений файлов.

Дублирование вставляемой функции необходимо, так как ее могут вызывать и функции, в которые она не вставлена. Иногда дублирование заменяют клонированием, чтобы учесть контекст вызова.

11.3 Открытая вставка при компиляции С-программ

11.3.4. Вставка функции

- ◇ Переименование формальных параметров и локальных переменных должно выполняться на новой копии вызываемой функции перед ее вставкой в вызывающую.
Если вставка выполняется на уровне исходного кода, для упрощения задачи можно вводить новые области видимости локальных переменных, так что переименовывать придется только формальные параметры.
Можно ввести новые временные переменные для буферизации значений фактических параметров и заменять формальные параметры на эти временные переменные.
На последней стадии лишние инструкции можно исключить с помощью таких оптимизаций как распространение копий, исключение мертвого кода, исключение избыточных вычислений и др.

11.3 Открытая вставка при компиляции С-программ

11.3.5. Выяснение недостающей информации

- ◇ Когда для некоторых функций компилятор или линкер не может получить доступ к вызову функции и характеристикам использования в ней переменных, граф вызовов является неполным.
Как правило это происходит потому, что определения указанных функций и переменных содержатся в модулях программы или в библиотеках, к которым нет доступа. Такие функции называются *внешними функциями*.
- ◇ Другим источником неопределенностей в графе вызовов являются функции, вызываемые по указателю (вызов по указателю – отличительная черта языка С).

11.3 Открытая вставка при компиляции С-программ

11.3.5. Выяснение недостающей информации

- ◇ Для вызовов внешних функций и вызовов по указателю имеет смысл ввести специальный тип ребер графа вызовов.
- ◇ Если в случае вызова по указателю нет возможности решить, какие функции реально могут быть вызваны приходится предполагать наихудший возможный вариант, когда может быть вызвана каждая функция. Такое предположение приводит к множеству дополнительных ребер и циклов в графе вызовов

11.3 Открытая вставка при компиляции С-программ

11.3.6. Исключение недостижимых функций

- ◇ Выполнение программы всегда начинается с функции **main**, поэтому любая функция, недостижимая из функции **main**, никогда не будет выполняться, и ее следует исключить.
- ◇ Функция **f ()** *достижима* из **main**, если
 - ◇ на ВГВ имеется путь из **main** в **f ()** или
 - ◇ **f ()** может быть вызвана обработчиком исключительных ситуаций или может быть активизирована каким-нибудь другим асинхронным событием.
- ◇ Для С-программ такие функции можно обнаружить, идентифицируя все функции, адреса которых используются в выполняемой программе.
- ◇ Кроме того, мы вынуждены считать (консервативность), что все внешние функции достижимы.

11.3 Открытая вставка при компиляции С-программ

11.3.7. Сокращение количества вставок

- ◇ Решение произвести открытую вставку не только отбирает процедуры, но и определяет порядок, в котором эти процедуры должны быть вставлены.
- ◇ Пусть $(A \rightarrow B)$ обозначает открытую вставку A в B ,
 $[X \ Y \ Z]$ – определяет упорядоченную последовательность событий, начинающуюся с X , а
 $\{X \ Y \ Z\}$ – три неупорядоченных события.
- ◇ Точный порядок вставок очень важен: $[(A \rightarrow B) (B \rightarrow C)]$
не то же самое, что $[(B \rightarrow C) (A \rightarrow B)]$,
так как
 $[(A \rightarrow B) (B \rightarrow C)] == [(B^* = A \cup B) (C^* = B^* \cup C)]$, а
 $[(B \rightarrow C) (A \rightarrow B)] == [(C' = B \cup C) (B' = A \cup B)]$

11.4 Эвристики

11.4.1 Простейшая стратегия

- ◇ Стратегия 1: Поверхностный анализ
 - ◇ изучение исходного кода вызываемой функции, чтобы оценить затраты
 - ◇ использование оценок затрат для решения, когда выполнять вставку
 - ◇ оптимизация программы после вставки, чтобы удалить лишние переменные (распространение копий)

11.4 Эвристики

11.4.2 Стратегия 2

- ◇ Стратегия 2: Глубокий анализ
 - ◇ выполнить вставку
 - ◇ выполнить после вставки анализ и оптимизацию полученной программы
 - ◇ оценить улучшения программы от оптимизаций и измерить объем кода после оптимизаций
 - ◇ отказаться от вставки, если сумма затрат превышает суммарные улучшения
 - ◇ намного увеличивает время компиляции

11.4 Эвристики

11.4.3 Стратегии 3 и 4

- ◇ Стратегия 3: Эвристики (модифицированная версия стратегии 2)
 - ◇ выполнить стратегию 2: "пробную" вставку
 - ◇ записать компромиссы между затратами и выгодами в постоянную базу данных
 - ◇ использовать предыдущие результаты по компромиссам для "похожих" точек вызова

- ◇ Стратегия 4: Использование методов машинного обучения

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

- ◇ Область действия *глобальной* оптимизации ограничена рамками одной процедуры. Открытая вставка – один из способов расширить область действия оптимизирующих преобразований. Ее можно рассматривать как один из видов межпроцедурной оптимизации. После вставки оптимизации, ограниченные пределами процедуры можно применять к объединенному коду вызывающей и вызываемой процедур.
- ◇ Агрессивная вставка включает предварительное клонирование вызываемой процедуры, чтобы учесть в клоне важные особенности контекста вызова.
- ◇ Разница между копией и клоном.

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

- ◇ Термин *агрессивная вставка*, введенный в 1997 году, означает систематическое применение клонирования и вставки процедур, определенных во всех модулях оптимизируемой программы, используя как аннотации пользователя, так и результаты профилирования, с минимальными ограничениями. Измерения показали, что для набора стандартных тестов обеспечивается ускорение в 1.5 – 2 раза (а не 5 – 10%, достигаемых в остальных оптимизациях).
- ◇ Агрессивная вставка состоит в выполнении нескольких фаз клонирования и вставки. Такая многофазная структура выбрана потому, что очень трудно предсказать влияние каждого клона и каждой вставки на оптимизацию. Выполняя все вставки в одну фазу, трудно учесть все особенности вставляемой процедуры прежде всего потому, что они выявляются в процессе первого этапа вставки.

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

◇ Управление фазой вставки обеспечивается введением параметра *бюджет*. Бюджет (**B**) это оценка возрастания времени компиляции в связи с увеличением длины вызывающей процедуры в результате данной вставки.

◇ Бюджет вычисляется по формуле

$$B = Q * \text{sizeof}(R)$$

По умолчанию для каждой вставляемой процедуры **Q** полагается равным 2, хотя, как показал опыт эксплуатации, среднее значение **Q** = 1.2 (длина процедуры возрастает примерно на 20%).

◇ Пользователь имеет возможность вручную уточнять бюджет в любом направлении.
В начале алгоритма бюджет распределяется между отдельными фазами клонирования и вставки (распределение бюджета тоже может уточняться пользователем).

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

- ◇ Затем начинается *основной цикл*, который состоит в выполнении клонирований и вставок и заканчивается либо при исчерпании бюджета, либо по истечении лимита времени (**limit**), отведенного на выполнение данной группы вставок. Клоны сохраняются в файле (базе данных) **D** для ускорения процесса.
- ◇ В программе (см. следующий слайд) использованы следующие обозначения:
 - G** = $\langle \mathbf{R}, \mathbf{E} \rangle$ – граф вызовов
 - B** – бюджет
 - C** – стоимость
 - Q** – приращение стоимости
 - S[]** – распределение бюджета
 - D** – база клонов
 - limit**

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

```
    Inline and Clone (G)
    INPUT G:⟨R, E⟩
    // оценка стоимости компиляции
    C = 0
    FOREACH R IN G
        C = C + (sizeof(R))
    // вычисление бюджета
    growth factor Q = 1.2
    budget B = C * Q
    // вычисление распределения бюджета
    S[0] = C + B * 0.2
    . . .
    S[limit-1] = C + B
    // клонирование и вставка
    clone database D = { }
    pass number P = 0
    WHILE ( C < B & P < limit ) DO
        C = Clone (G, S[P], C, D)
        C = Inline (G, S[P], C)
        P = P+1
```

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- ◇ Клонирование начинается с **отбора вставляемых процедур**. Вызываемые процедуры исследуются по очереди.
 - ◇ Сначала проверяется **допустимость вставки** (например, вставка недопустима, если имеются грубые несоответствия типов между вызывающей и вызываемой процедурами, или нет соответствия между числом передаваемых параметров).
 - ◇ Затем проверяется, предоставляет ли вызывающая процедура какую-либо интересную дополнительную информацию (например, может в качестве дополнительной информации передаваться значение параметра по умолчанию, например, 0).
 - ◇ Если контекст вызова достаточно интересен, из базы данных запрашивается, как этот контекст используется в вызываемой процедуре. Если выясняется, что вызываемая процедура может быть оптимизирована после замены формальных параметров на фактические, вызов считается подходящим для вставки.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- ◇ **Отбор вставляемых процедур.**
 - ◇ Клонер (фаза клонирования) эффективно совмещает информацию, поставляемую вызывающей процедурой, со своей внутренней информацией для создания спецификации клона.
 - ◇ Обычно на этой фазе ограничиваются спецификацией констант, поставляемых вызывающей процедурой. Однако отметим, что на самом деле можно (и нужно) использовать и другую информацию из вызывающей процедуры:
 - ◆ сведения об алиасах,
 - ◆ сведения о том, что вызываемая процедура не использует часть фактических параметров,
 - ◆ сведения о том, что вызывающая процедура не использует возвращаемого значения и т.п.
 - ◇ Каждый параметр рассматривается независимо от остальных.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- ◇ **Формирование групп клонов.**
 - ◇ Вычисляется информация о профиле блока, содержащего вызов ("вес" вызова), с целью определить, насколько вызов интересен в аспекте оптимизации.
 - ◇ Найдя **интересный вызов**, Клонер мог бы перейти к построению аналогичных спецификаций для остальных вызовов, но это привело бы к ненужному резкому росту числа клонов.
 - ◇ Вместо этого после обнаружения **интересного вызова** Клонер пытается использовать спецификации клонов для "жадного" построения *группы клонов*: множества вызовов, которые могут безопасно обратиться к клону, удовлетворяющему рассматриваемой спецификации. Для этого исследуется каждый вызов вызываемой процедуры, чтобы убедиться, что контекст вызова совместим со спецификацией, созданной для группы клонов. Если это так, то вызов включается в группу клонов.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- ◇ **Формирование групп клонов.**
 - ◇ После того как группа клонов полностью сформирована, Клонер оценивает выгоду от замены вызываемой процедуры клоном из этой группы.
При этом учитываются такие факторы как оценка общего количества вызовов, которые будут вызывать клон из группы вместо исходной процедуры, и значение контекстной информации в точке вызова.
 - ◇ После исследования вызовов у Клонера будет набор групп клонов, описывающих конкретные клоны, которые могут быть созданы, и оценки выигрыша от создания каждого клона.
 - ◇ В заключение Клонер ранжирует группы клонов по выигрышу, "жадно" создавая клоны и модифицируя вызовы, пока не будет исчерпана соответствующая доля бюджета.
 - ◇ Любая группа клонов, которая не была обработана на рассматриваемой фазе, отбрасывается.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

◇ Создание клона.

- ◇ Создать клон очень просто. Копируется промежуточное представление (**IR**) клонируемой процедуры, и каждый формальный параметр, известный из контекста вызова, превращается в локальную переменную вызываемой процедуры и инициализируется подходящей константой, известной из контекста вызова, в блоке **Entry** рассматриваемого клона. Если имеются незначительные несоответствия типов, используется явное приведение типов.
- ◇ Клон всегда помещается в тот же модуль, что и клонируемая процедура.
Если вызывающая процедура, определенная в другом модуле, передает символьную информацию, доступную только в ее модуле (например, адрес статической процедуры), эту информацию необходимо сделать глобальной и присвоить ей уникальное имя, не конфликтующее ни с каким пользовательским именем₅₆

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

◇ Хранение клонов.

- ◇ По мере создания клонов каждый клон и связанная с ним спецификация записываются в специальную базу данных.**

Эта база данных используется на более поздних фазах: если становится возможной ситуация, когда Клонер выработает такую же спецификацию клона, что и на более ранней фазе, так как промежуточные оптимизации добавили информацию к вызовам, которые до этого не заслуживали внимания.

В этом случае наличие клона в базе данных приводит к его повторному использованию, ускоряя работу Клонера.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования



Модификация вызовов.



Модификация вызовов группы клонов, чтобы обеспечить вызов требуемого клона, тоже очень проста.

Спецификация клона описывает сигнатуру новой процедуры, следовательно, любые параметры, включенные в клон могут редактироваться согласно списку фактических параметров. После этого вызов модифицируется, чтобы ссылаться на клон, а не на исходную процедуру.



Такая модификация, естественно влечет соответствующие изменения в графе вызовов, чтобы тот мог отражать новые отношения между вызывающей, вызываемой и клонируемой процедурами и клоном.

В частности, если все вызовы клонируемой процедуры будут заменены на вызовы одного из ее клонов, клонируемая процедура может стать недостижимой в графе вызовов и будет удалена.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

◇ Модификация вызовов.

- ◇ Оценивая влияние конкретной группы клонов на время компиляции, Клонер пытается предсказать последующее удаление клонируемой процедуры, так как считается, что группа клонов, для которой клонируемая процедура будет гарантированно удалена, не влияет на время компиляции.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

Clone (G, B, C, D) : returns C

INPUT

call graph $G:\langle R, E \rangle$

budget B

current cost C

clone database D

ALGORITHM

//Настройка: создать дескрипторы параметров

//и дескрипторы контекстов вызова

FOR EACH routine R **IN** G

create parameter-usage descriptor P(R)

FOR EACH edge E **IN** G

create calling-context descriptor S(E)

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
// Построение групп клонов; R – вызываемые
// процедуры intersect() и matches() проверяют
// совместимость со спецификацией
    FOREACH edge E in G
        R = E.target
        IF(clonable(R) & clonable(E)) THEN
// CS – спецификация клона
// CG – группа клонов
            CS = intersect(S(E), P(R))
            IF( CS is nonempty) THEN
                CG = (R, CS, E)
                FOREACH edge EJ incident on R
                    IF(clonable(E') & matches(S(E), CS)) THEN
                        add E' to CG
                        estimate benefit of CG
```


11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
// отбор клонов
    sort CGs by benefit;  $C' = C$ 
    FOREACH CG IN CGs
        cost  $x = (\text{sizeof}(R))$ 
        IF (  $C' + X < B$  ) THEN
            accept CG;  $C' = C' + X$ 
// создание клонов и фиксация вызовов
    FOREACH CG IN accepted CGe
        IF(!lookup(D, R, CS) > 0) THEN
             $R' = \text{make clone}(R, CS)$ 
            add database entry ( R, CS,  $R'$  )
        FOREACH edge  $E'$  IN CC
            change target of  $E'$  from R to  $R'$ 
```

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
// оптимизация клонов и перестройка ГВ
  FOREACH newly created clone R'
    optimize(R')
    C = C + (sizeof(R'))
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- ◇ **Общая структура фазы вставки** аналогична структуре фазы клонирования.
- ◇ **Отбор процедур и клонов для вставки**
 - ◇ Вставщик (**Inliner**) рассматривает все вызовы, выявляя любые **ограничения** на вставку (языковые, технические, прагматические, пользовательские). Выявляются также **неверные вызовы**, содержащие ошибки (несогласованность типов, несогласованность по местности операций и т.п.)
 - ◇ Технические ограничения на вызовы – это случаи, когда информация, связанная с вызываемой процедурой противоречит информации, связанной с вызывающей процедурой. Например, в коде вызывающей процедуры может быть указано, что перестановка операций с плавающей точкой разрешена, тогда как вызываемая процедура может декларировать, что такая перестановка неприемлема. По большому счету ограничения такого рода призваны упростить задачу представления информации.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- ◇ **Отбор процедур и клонов для вставки**
 - ◇ К прагматическим проблемам относятся такие вопросы как обработка вызываемых процедур, использующих **alloca()** для динамического расширения стека, или вставка вызова, фактические параметры которого описывают перекрывающиеся области памяти, а вызываемая процедура разработана в предположении, что ее формальные параметры не могут быть алиасами.
 - ◇ Пользовательские ограничения получаются в результате использования различных опций командной строки и прагм.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- ◇ **Ранжирование отобранных вставок.**
 - ◇ После того, как требуемые вызовы, заменяемые вставкой, отобраны, им на время выполнения алгоритма присваивается показатель качества (приоритет). Высший приоритет присваивается наиболее частым вызовам. Вызовам, находящимся в блоках, выполняемых менее часто, начисляются штрафы. Это помогает избегать вставок в не критический путь; невыполнение этого требования может вызвать повышение регистрового давления, приводящего к сливам на критических путях, что существенно снижает производительность.
 - ◇ После этого Вставщик следует по списку вставок в порядке их приоритетов. Оценивается влияние каждой вставки на время компиляции и, в пределах текущего бюджета, вставка принимается к исполнению.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- ◇ **Взаимодействие между вставками.**
 - ◇ Вычисление влияния на время компиляции осложняется взаимодействием между вставками.
Например, если **A** вызывает **B** и **B** вызывает **C**, цена вставки **B** в **A** зависит от того, была ли предварительно выполнена вставка **C** в **B**.
 - ◇ Для моделирования указанной зависимости, Вставщик поддерживает график (расписание) порядка, в котором он будет выполнять принятые вставки.
В общем и целом Вставщик пытается обрабатывать граф вызовов снизу-вверх.
 - ◇ Чтобы вычислить цену вставки **B** в **A**, дескриптор указанной вставки вставляется в соответствующую точку графика вставок. Если потом выясняется, что **B** является целью более ранней вставки (или вставок), то для вычисления цены оптимизации **A** используется оценка размера **B** после этих вставок.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- ◇ **Вставщик** "жадно" принимает и обрабатывает вызовы пока не исчерпается выделенная доля бюджета. При этом оставшиеся жизнеспособные вызовы отбрасываются, так как не осталось времени для их обработки.
- ◇ В случае косвенных вызовов (вызовов по указателю) вызываемая функция становится известной во время выполнения, так что эти вызовы не могут быть вставлены или клонированы. Однако, иногда удастся во время компиляции выявить вызываемую функцию.
Например, Вставщик может клонировать вызовы, в которых вызывающая процедура передает указатель вызываемой процедуре, которая использует для косвенного вызова формальный параметр. Последующая оптимизация "Распространение констант" может вычислить и передать информацию, необходимую для превращения косвенного вызова в прямой, что позволит на более поздней фазе клонировать или вставить этот вызов.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

Inline (G, B, C) : returns C

INPUT

call graph G: (R, E)

budget B

current cost C

ALGORITHM

// screen inline candidates

FOR EACH edge E IN G

IF (inlineable(E)) > THEN

accept E; compute benefit(E)

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

```
// отобрать вызовы для вставки
// отсортировать отобранные вызовы E'
// по выгоде
C' = C
FOR EACH accepted edge E
    insert E into schedule
    cost X =
        (sizeof(E.target+E.source)2
        - (sizeof(E.target)2
    C'' = C'
    C' = C' + X
IF ( E.target is source in later inline )
THEN adjust C' for cascaded cost
IF ( C' > B ) THEN
    remove E from schedule
C' = C''
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

```
// выполнить вставки
FOR EACH scheduled edge E
    inline E.target into E.source
// оптимизировать вставки и перестроить ГВ
FOR EACH routine inlined into R'
    optimize(R')
    C = C + (sizeof(R'))
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

◇ Обоснование эвристик.

- ◇ Методики, позволяющей определять во время компиляции оптимальное множество вставок или клонов, к сожалению, не существует. Необходимо оценивать как выигрыш времени во время исполнения программы, так и затраты на оптимизацию во время компиляции. Более того, при разработке любой программы разумных размеров применяется и клонирование, и открытая вставка. При этом даже упрощенный вариант проблемы эквивалентен известной задаче о рюкзаке, которая, как известно, является NP полной.
- ◇ Каждый вызов может быть либо заменен вставкой, либо клонирован, причем окончательный код существенно зависит от порядка клонирований и вставок. Поэтому порядок вставок и клонирований должен определяться с помощью эвристики. Для выбранной эвристики полезно показать, что решения, принимаемые с ее помощью, выглядят разумными.

11.5 Алгоритм «Агрессивная вставка»

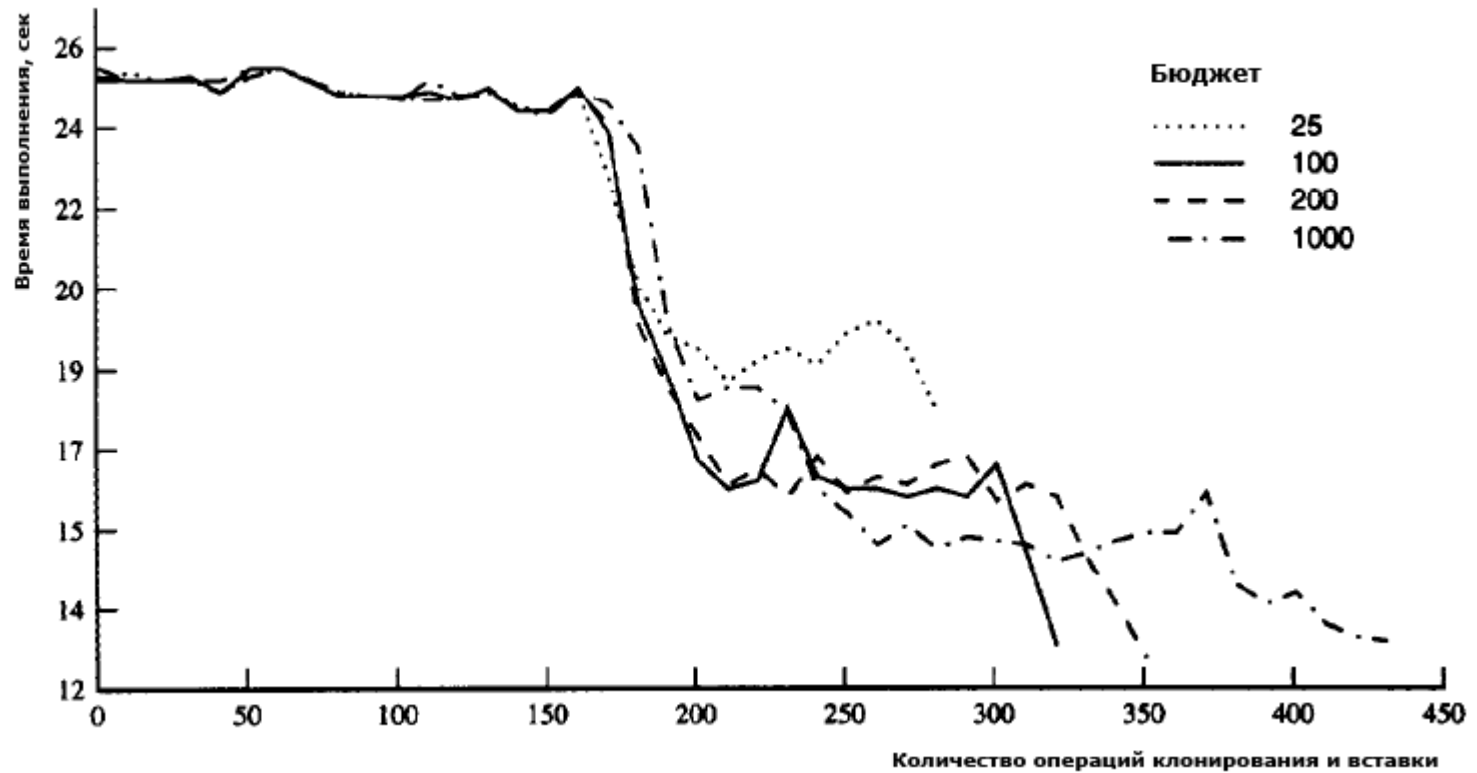
11.5.3 Алгоритм. Фаза вставки

- ◇ **Обоснование эвристик.**
 - ◇ Для оценки качества рассмотренных эвристик были проведены экспериментальные исследования на тестах (бенчмарках) из *SpecInt*.
Результаты для теста **022.li** в виде графика, иллюстрирующие одну из методик оценки качества эвристик см. на слайде 29.
 - ◇ Предоставляемый бюджет менялся от сравнительно малого значения 25 до очень большого бюджета 1000. Для каждого уровня бюджета несколько раз выполнялась компиляция теста. Во время каждой компиляции Вставщик искусственно останавливался после определенного количества вставок и/или клонирований.
 - ◇ Полученные кривые показывают дополнительную пользу каждой последующей вставки или клонирования. Как можно видеть из графика, лишь очень немногие вставки или клоны оказывают неблагоприятное воздействие на производительность.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

◇ Обоснование эвристик



11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

◇ Обоснование эвристик.

- ◇ Как показали экспериментальные исследования, после того как выделенный бюджет достигает достаточно большого значения (например, 100), дальнейшее приращение производительности за счет дополнительных вставок **для большей части тестов** из *SpecInt* прекращается (производительность достигает своего предельного значения). Поэтому в подсистеме компилятора, реализующей агрессивную вставку, значение бюджета, назначаемое по умолчанию, было выбрано равным 100, что позволило максимизировать производительность тестов из *SpecInt*, без выполнения ненужных вставок

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

◇ Обоснование эвристик.

- ◇ Как показало исследование вызовов в тестах (бенчмарках) из *SpecInt*, среди них имеется значительное количество межмодульных вызовов. Обеспечение возможности заменять вставками межмодульные вызовы имеет решающее значение для обеспечения хорошей производительности оптимизируемой программы.

12. Межпроцедурный анализ

12.1 Введение в межпроцедурный анализ

12.1.1 Вводные замечания

- ◇ Неэффективность, связанная с вызовом процедуры может проявляться как:
 - ◇ потеря информации при анализе процедуры, из которой происходит вызов, и связанная с этим невозможность выполнить требуемую оптимизацию (консервативность)
 - ◇ дополнительные накладные расходы, связанные с вызовом процедуры.
- ◇ Межпроцедурный анализ помогает решить первую из указанных проблем.

12.1 Введение в межпроцедурный анализ

12.1.2 Контекстно-нечувствительный межпроцедурный анализ

- ◇ Простейший, но очень неточный подход к межпроцедурному анализу, известный как *контекстно-нечувствительный анализ*, заключается в **рассмотрении каждой команды вызова и каждой команды возврата как операции безусловного перехода**.
- ◇ Создается надграф ГПУ, в котором, помимо обычных ребер внутри-процедурных потоков управления, имеются дополнительные ребра, соединяющие:
 - ◇ каждую точку вызова с началом вызываемой в ней процедуры;
 - ◇ каждую команду возврата с точкой, следующей за точкой вызова.
- ◇ Добавляются команды присваивания каждого фактического параметра соответствующему формальному параметру, а также присваивания возвращаемого значения переменной, получающей результат.
- ◇ К построенному надграфу ГПУ, который называется межпроцедурным графом потока управления (МГПУ), применяется стандартный глобальный анализ.

12.1 Введение в межпроцедурный анализ

12.1.2 Контекстно-нечувствительный межпроцедурный анализ

- ◇ Простейший, но очень неточный подход к межпроцедурному анализу, известный как *контекстно-нечувствительный анализ*, заключается в **рассмотрении каждой команды вызова и каждой команды возврата как операции безусловного перехода**.
- ◇ Создается надграф ГПУ, в котором, помимо обычных ребер внутри-процедурных потоков управления, имеются дополнительные ребра,

Построение МГПУ похоже на слияние всех процедур программы в одну большую процедуру (похоже на открытую вставку, но вставки не производится, просто вставляются команды перехода – дуги МГПУ)

- ◇ добавляются команды присваивания каждого фактического параметра соответствующему формальному параметру, а также присваивания возвращаемого значения переменной, получающей результат.
- ◇ К построенному надграфу ГПУ, который называется межпроцедурным графом потока управления (МГПУ), применяется стандартный глобальный анализ.

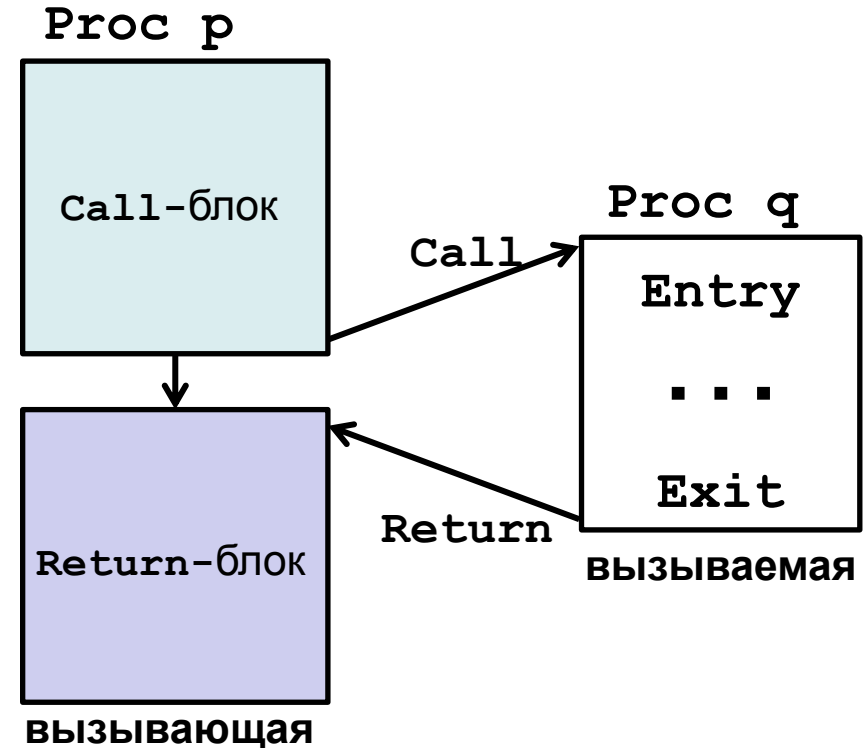
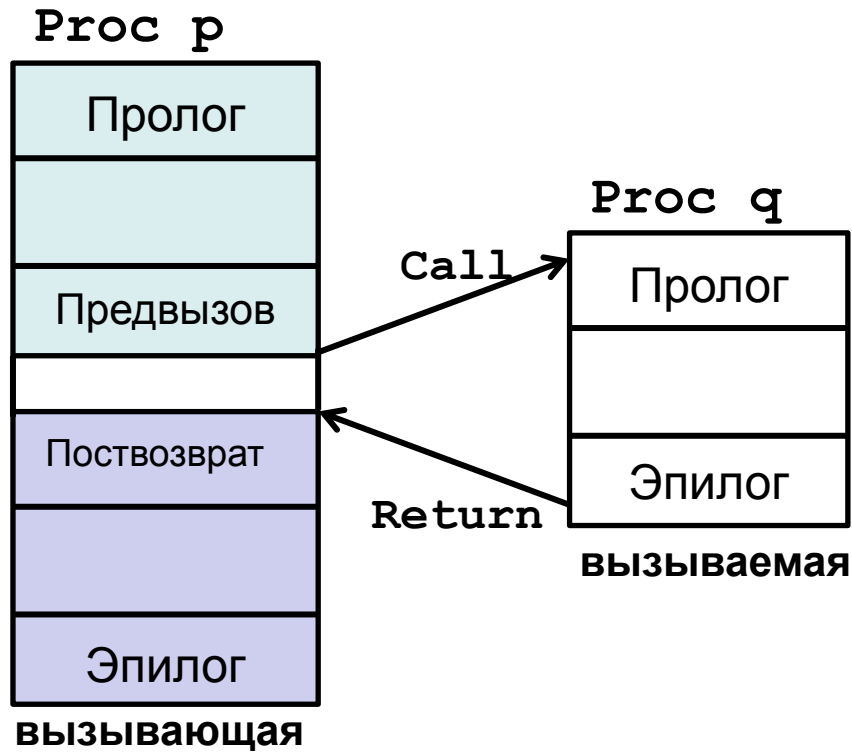
12.1 Введение в межпроцедурный анализ

12.1.2 Контекстно-нечувствительный межпроцедурный анализ

- ◇ Для **контекстно-нечувствительного** межпроцедурного анализа МГПУ строится следующим образом:
 - ◇ каждый предвызов вызывающей (*caller*) удаляется, а соответствующий вызов выделяется в отдельный базовый блок (*call*-блок)
 - ◇ каждый поствозврат вызывающей удаляется и вместо него вставляется дополнительный базовый блок (*return*-блок)
 - ◇ каждый *call*-блок соединяется дугой с блоком *Entry* вызываемой
 - ◇ блок *Exit* вызываемой (*cally*) соединяется дугой с соответствующим *return*-блоком вызывающей
 - ◇ пролог и эпилог вызываемой удаляются
 - ◇ если вызывающая содержит несколько вызовов одной и той же вызываемой, все ее *call*-блоки и *return*-блоки соединяются с одним и тем же экземпляром вызываемой

12.1 Введение в межпроцедурный анализ

12.1.2 Контекстно-нечувствительный межпроцедурный анализ

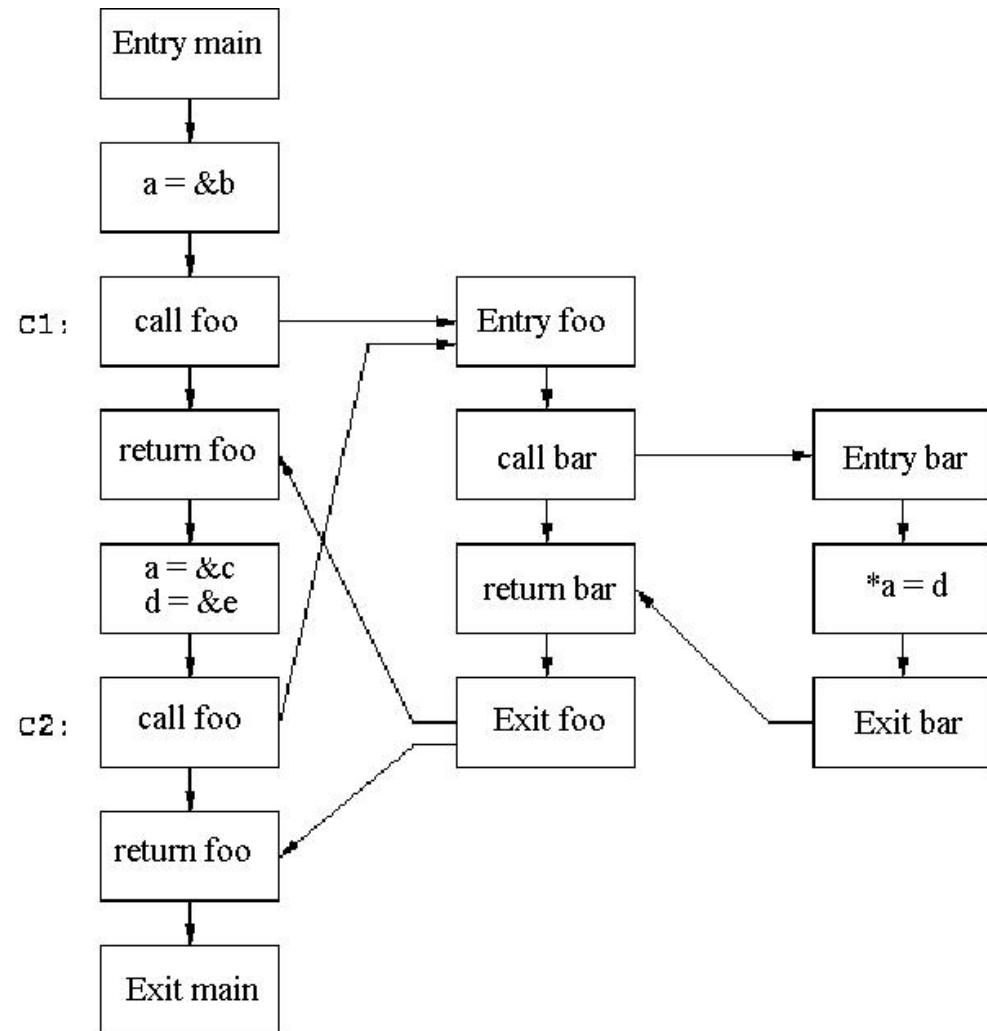


12.1 Введение в межпроцедурный анализ

12.1.2 Контекстно-нечувствительный межпроцедурный анализ

◇ На рисунке представлен МГПУ программы:

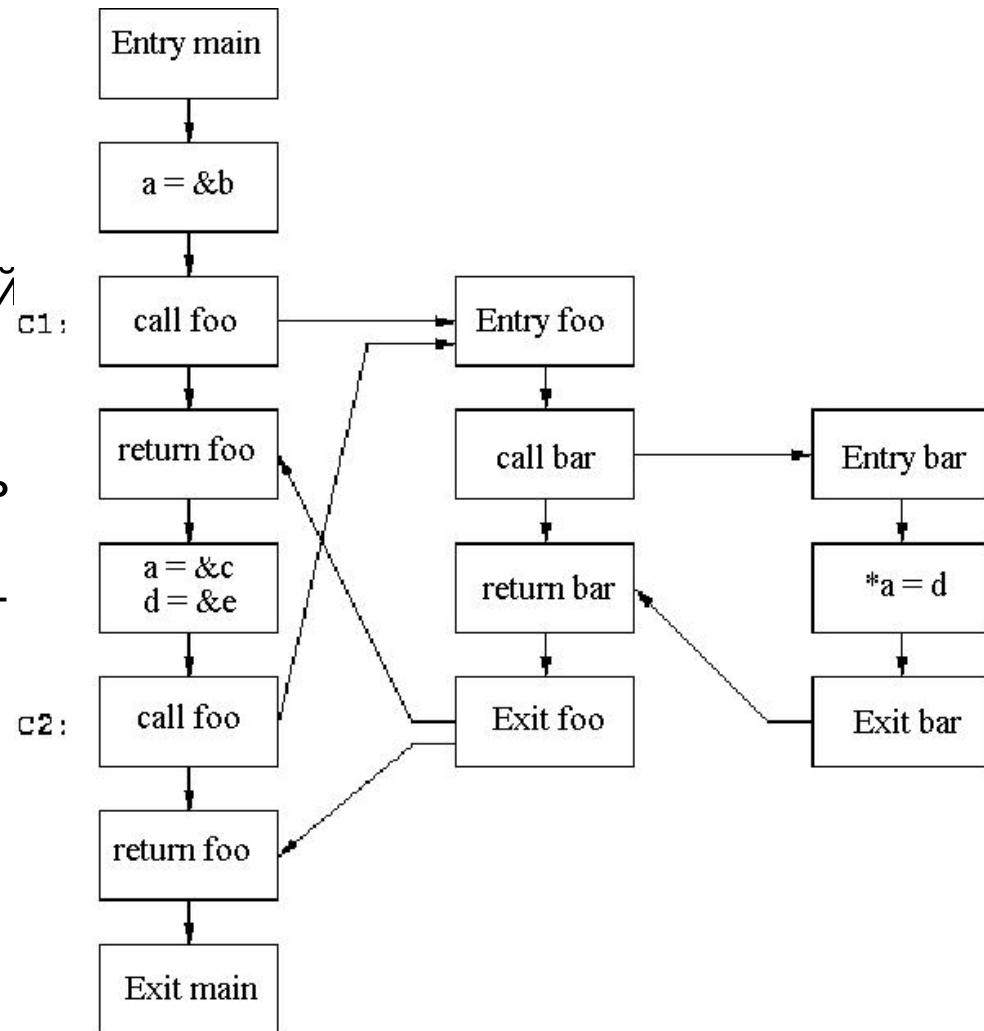
```
int **a, *b, *c, *d, e
foo( ) {
    bar( );
}
bar ( ) {
    *a = d;
}
main( ) {
    a = &b;
    foo( );
    a = &c;
    d = &e;
    foo( );
}
```



12.1 Введение в межпроцедурный анализ

12.1.2 Контекстно-нечувствительный межпроцедурный анализ

- ◇ Основной недостаток — нечувствительность к контексту: оба вызова передают управление на один и тот же экземпляр вызываемой функции
- ◇ К сожалению, этот недостаток неустраним: попытка увеличить количество экземпляров вызываемых функций приводит к экспоненциальному росту накладных расходов по памяти, а затем и по времени выполнения
- ◇ Отметим также, что в рассматриваемом примере процедуры не имеют параметров



12.1 Введение в межпроцедурный анализ

12.1.3 Построение графа вызовов

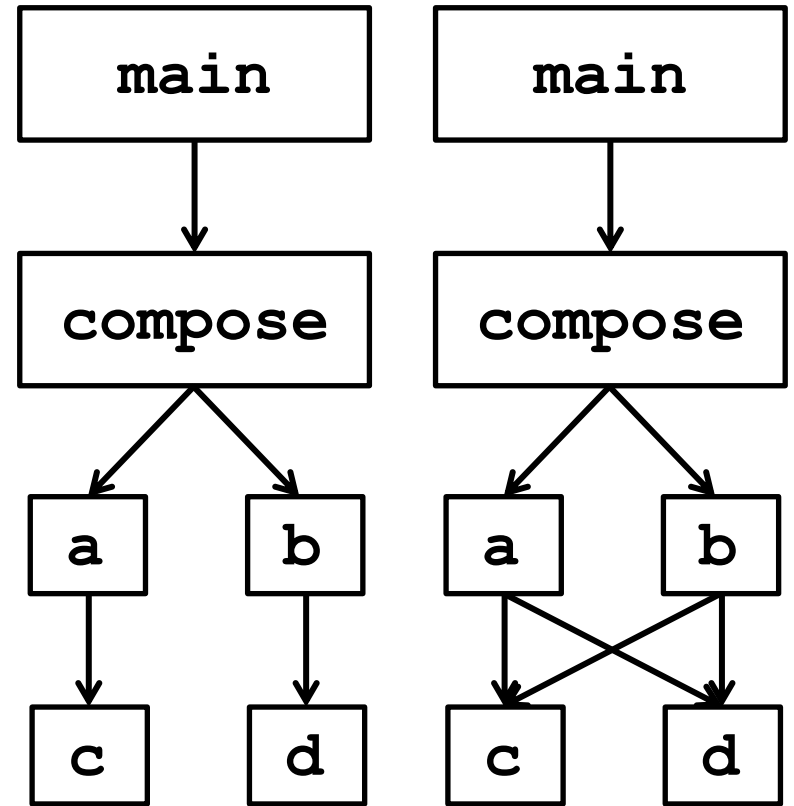
- ◇ В простейшем случае, когда каждая процедура вызывается по имени, задача построения графа вызовов (ГВ) проста. Компилятор создаёт вершину ГВ для каждой процедуры программы и добавляет дугу ГВ для каждого места вызова. Этот процесс занимает время, пропорциональное числу процедур и числу мест вызова.

12.1 Введение в межпроцедурный анализ

12.1.4 Пример

◇

```
int compose( int f(), int g()) {  
    return f(g);  
}  
int a( int z() ) {  
    return z();  
}  
int b( int z() ) {  
    return z();  
}  
int c( ) {  
    return ...;  
}  
int d( ) {  
    return ...;  
}  
int main(int argc, char *argv[]) {  
    return compose(a,c) + compose(b,d);  
}
```



Точный ГВ

Приближенный ГВ

Трудно понять, какими могут быть параметры **compose**

12.1 Введение в межпроцедурный анализ

12.1.5 Пример

◇ Рассмотрим программу на языке C.

```
int (*pf) (int) ;
int fun1(int x) {
    if (x < 10)
c1:        return (*pf) (x+1) ;
    else
        return x;
}
int fun2(int y) {
    pf = &fun1;
c2:        return (*pf) (y) ;
}
void main() {
    pf = &fun2;
c3:        (*pf) (5) ;
}
```

Метки *c1*, *c2* и *c3*
помечают точки вызова
и не являются частью
программы.

В программе имеются
две функции типа
int (*pf) (int) –
fun1 и **fun2**
(тип функции **main** не
соответствует типу
указателя **pf**).

12.2. Межпроцедурный анализ

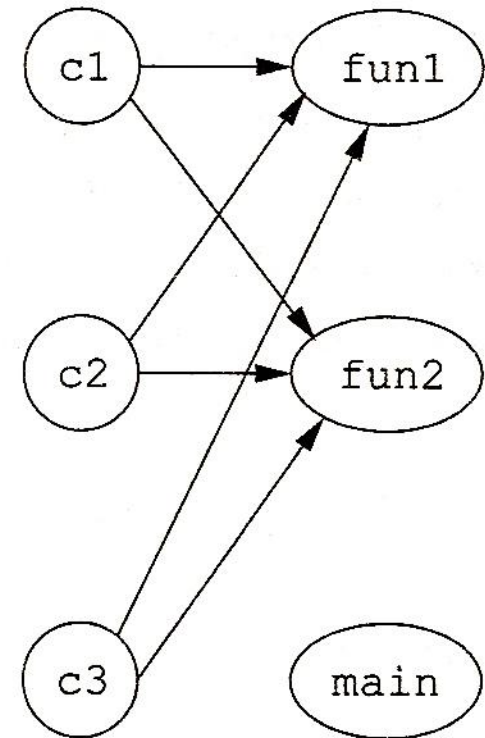
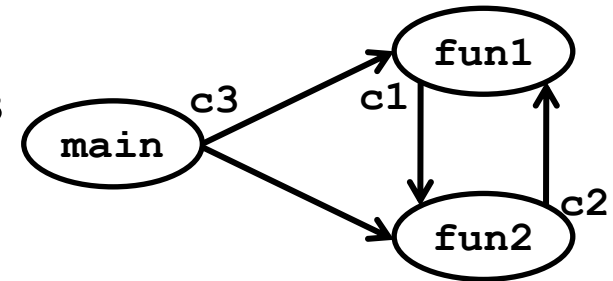
12.2.1 Граф вызовов. Пример

◇ Простейший анализ того, на что может указывать **pf**, состоит в исследовании типов функций:

- ◇ функции **fun1** и **fun2** имеют тот же тип, что и указатель **pf**,
- ◇ функция **main** имеет другой тип.

В результате получится *консервативный граф вызовов*

(граф вызовов, построенный с помощью консервативного анализа)



12.2. Межпроцедурный анализ

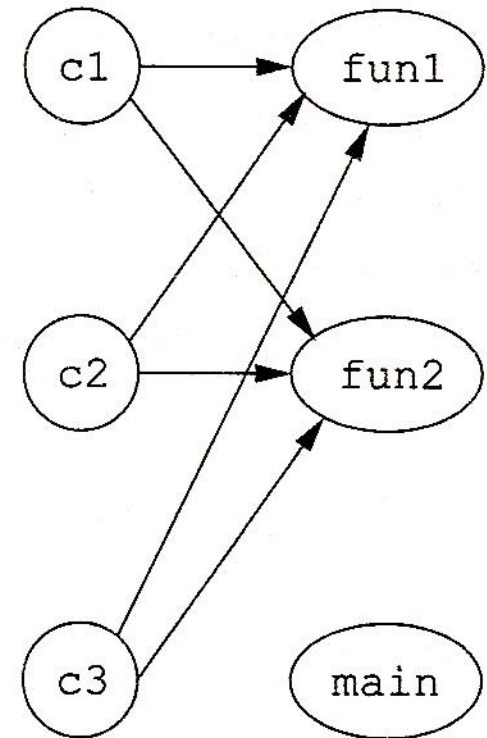
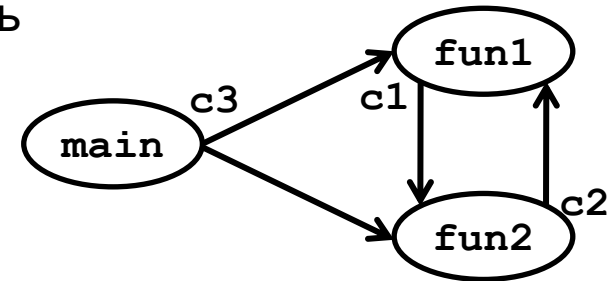
12.2.1 Граф вызовов. Пример

◇ Простейший анализ того, на что может указывать **pf**, состоит в исследовании типов функций:

- ◇ функции **fun1** и **fun2** имеют тот же тип, что и указатель **pf**,
- ◇ функция **main** имеет другой тип.

В результате получится *консервативный граф вызовов*

```
int (*pf)(int);
int fun1(int x) {
    if (x < 10)
c1:    return (*pf)(x+1);
    else
        return x;
}
int fun2(int y) {
    pf = &fun1;
c2:    return (*pf)(y);
}
void main() {
    pf = &fun2;
c3:    (*pf)(5);
}
```

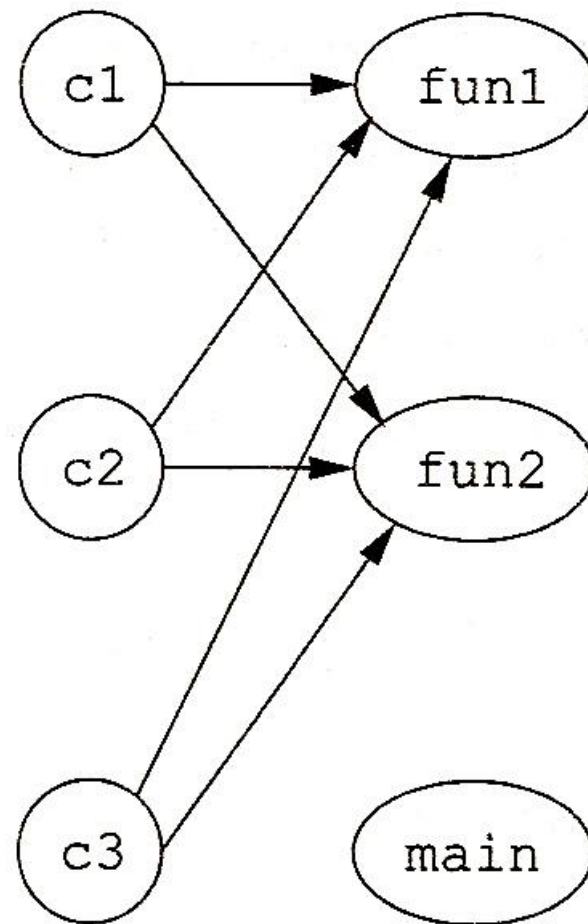


12.2. Межпроцедурный анализ

12.2.1 Граф вызовов. Пример

- ◇ Более точный анализ позволяет обнаружить:
 - ◇ в функции **main** указателю **pf** присваивается значение **&fun2**;
 - ◇ в функции **fun2** указателю **pf** присваивается значение **&fun1**;
 - ◇ никаких иных присваиваний указателю **pf** в программе нет (следовательно, указатель **pf** не может указывать на функцию **main**).

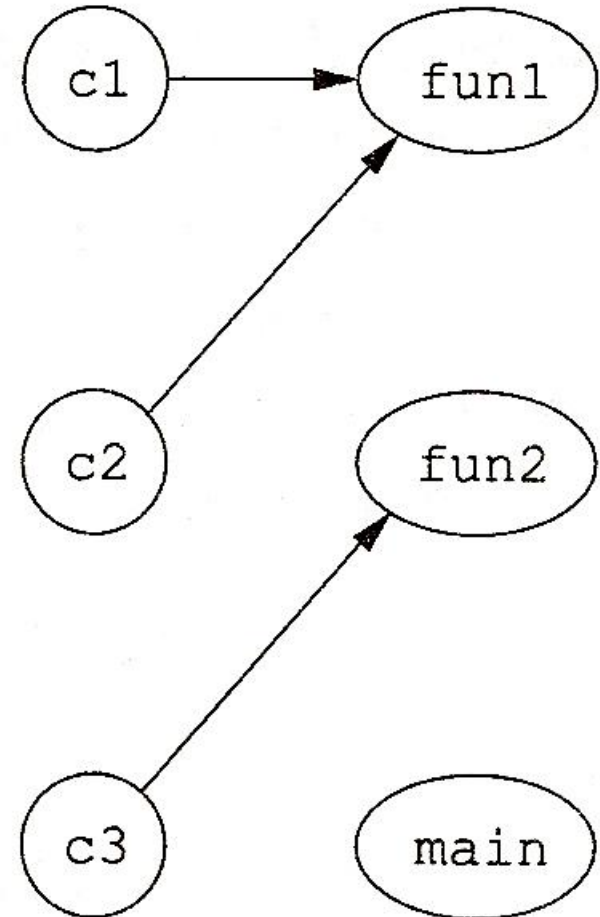
В результате получится тот же самый граф вызовов.



12.2. Межпроцедурный анализ

12.2.1 Граф вызовов. Пример

- ◇ Еще более точный анализ позволяет установить:
 - ◇ **pf** ссылается на **fun2** только в точке *c3* (этому вызову непосредственно предшествует присваивание **pf = &fun2**)
 - ◇ в точке *c2* **pf** ссылается только на **fun1**
 - ◇ в функцию **fun1** можно попасть только из функции **fun2**,
 - ◇ в функции **fun1** значение **pf** не изменяется, следовательно внутри функции **fun1** **pf** всегда ссылается на **fun1** (в частности, в точке *c1* **pf** ссылается на **fun1**).



12.2. Межпроцедурный анализ

12.2.2 Чувствительность к контексту. Пример

- ◇ Поведение каждой процедуры, вообще говоря, зависит от контекста, в котором она вызвана.
- ◇ **Пример.** Рассмотрим фрагмент программы:

```
        for (i = 0; i < n; i++) {  
c1:          y1 = 0;  
              t1 = f(y1) ;  
c2:          y2 = 243; y1 += y2  
              t2 = f(y2) ;  
c3:          t3 = f(y1) ;  
              x[i] = t1 + t2 + t3;  
        }  
  
int f(int v) {  
        return (v+1)  
}
```

12.2. Межпроцедурный анализ

12.2.2 Чувствительность к контексту. Пример

- ◇ Функция **f** вызывается в трех точках: *c1*, *c2* и *c3*.
 - ◇ вызов в точке *c1*: параметр имеет значение 0, возвращаемое значение равно 1
 - ◇ вызовы в точках *c2* и *c3*: значение параметра 243, возвращаемое значение равно 244

Таким образом, значение, возвращаемое функцией **f**, зависит от контекста.

- ◇ Установить, что переменным **t1**, **t2** и **t3** (а значит, и **x[i]**) присваиваются константные значения, можно только выяснив, что при вызове в контексте точки *c1* **f** возвращает значение 1, а при вызове в контекстах точек *c2* и *c3* – значение 244.
- ◇ После простейшего анализа можно заключить, что в любом вызове в данной программе **f** может возвращать либо 1, либо 244.

12.2. Межпроцедурный анализ

12.2.3 Строка вызовов

- ◇ Контекст вызова определяется содержимым всего стека вызовов. Последовательность точек вызова в стеке вызовов называется *строкой вызовов*.
- ◇ **Пример.** Рассмотрим фрагмент кода, полученный модификацией примера 12.2.2: вызовы функции **f** заменены вызовами функции **g**, которая вызывает **f** с тем же аргументом: появляется еще одна точка вызова **c4**.
- ◇ Имеется три строки вызовов функции **f**: $(c1, c4)$, $(c2, c4)$ и $(c3, c4)$. При этом в каждой строке вызовов значение **v**, передаваемое функции **f** зависит не от последней точки вызова **c4**, а определяется первым элементом строки вызовов.
- ◇ Таким образом, важная для анализа информация может определяться ранними элементами строки вызовов: **для получения точного ответа необходимо рассматривать всю строку вызовов.**

12.2. Межпроцедурный анализ

12.2.3 Строка вызовов

```
        for (i = 0; i < n; i++) {  
c1:          t1 = g(0) ;  
c2:          t2 = g(243) ;  
c3:          t3 = g(243) ;  
              X[i]= t1 + t2 + t3 ;  
        }  
  
        int g(int v) {  
c4:          return f(v) ;  
        }  
  
        int f(int v) {  
              return (v+1) ;  
        }
```

12.2. Межпроцедурный анализ

12.2.4 Рекурсивные вызовы

◇ **Пример.** Заменим в предыдущем примере функцию **g**:
пусть для $v > 1$ **g** рекурсивно вызывается v раз,
а для $v \leq 1$ **g** такое же, как и раньше.

```
int g (int v) {  
c4:     if (v > 1) {return g(v-1) ;}  
c5:     else {return f(v) ;}  
}
```

12.2. Межпроцедурный анализ

12.2.4 Рекурсивные вызовы

```
        for (i = 0; i < n; i++) {
c1:          t1 = g(0) ;
c2:          t2 = g(243) ;
c3:          t3 = g(243) ;
              X[i] = t1+t2+t3;
        }

        int g (int v) {
c4:          if (v > 1) {return g(v-1) ;}
c5:          else {return f(v) ;}
        }

        int f (int v) {
              return (v+1) ;
        }
```

12.2. Межпроцедурный анализ

12.2.4 Рекурсивные вызовы

- ◇ Для функции **f** имеются три возможные строки вызовов.
- 1 После вызова в точке $c1$ функция **g** немедленно вызывает функцию **f** с параметром **0**;
соответствующая строка вызовов ($c1, c5$).
 - 2 После вызова в точке $c2$ функция **g** вызывается **243** раза,
после чего вызывается функция **f** с параметром **1**;
соответствующая строка вызовов ($c2, c4, \dots, c4, c5$)
 - 3 После вызова в точке $c3$ функция **g** вызывается **243** раза, после
чего вызывается функция **f** с параметром **1**;
соответствующая строка вызовов ($c3, c4, \dots, c4, c5$)

12.2. Межпроцедурный анализ

12.2.5 k -ограниченный контекстно-чувствительный анализ (КЧА)

- ◇ В КЧА можно рассматривать не полные строки вызовов, а ограничиться только k последними точками вызовов:
 k -ограниченный КЧА.
- ◇ Контекстно-нечувствительный анализ – это 0-ограниченный КЧА.
- ◇ В примере 12.2.2 все константы могут быть найдены с помощью 1-ограниченного КЧА,
в примере 12.2.3 – с помощью 2-ограниченного КЧА.
в примере 12.2.4 (рекурсивный вызов) никакой k -ограниченный КЧА не в состоянии найти все константы.
- ◇ Заметим, что
при использовании КЧА даже для программ без рекурсии количество разных контекстов вызова может экспоненциально зависеть от количества процедур в программе.

12.2. Межпроцедурный анализ

12.2.6 Контекстно-чувствительный анализ на основе клонирования

- ◇ Сначала контекстно-зависимые процедуры клонируются по одной для каждого уникального контекста.
Затем к клонированному графу вызовов применяется контекстно-нечувствительный анализ.
- ◇ Отметим, что в реальности клонировать (размножать) код не требуется – можно просто использовать эффективное внутреннее представление для отслеживания результатов анализа каждого клона.
- ◇ Применим клонирование к рассмотренным примерам (12.2.3 и 12.2.4).
- ◇ **Пример.** Клонированная версия кода примера 12.2.3 показана на следующем слайде
Поскольку каждый контекст вызова работает со своим клоном, никакой путаницы не возникает:
клон **g1** получает в качестве аргумента **0** и возвращает **1**;
клоны **g2** и **g3** получают в качестве аргументов **243** и возвращают **244**.

12.2. Межпроцедурный анализ

Исходная программа	Клонированная программа
<pre> for(i = 0; i < n; i++) { c1: t1 =g(0); c2: t2 =g(243); c3: t3 =g(243); X[i]= t1 + t2 + t3; } int g(int v) { c4: return f(v); } int f (int v) { return (v+1); }</pre>	<pre> for (i = 0; i < n; c1: t1 = g1(0); c2: t2 = g2(243); c3: t3 = g3(243); X[i] = t1 + t2 + t3; } int g1 (int v) { c4.1: return f1(v); } int g2 (int v) { c4.2: return f2(v); } int g3 (int v) { c4.3: return f3(v); } int f1 (int v) { return (v+1); } int f2 (int v) { return (v+1); } int f3 (int v) { return (v+1); } }</pre>

12.2. Межпроцедурный анализ

12.2.6 Контекстно-чувствительный анализ на основе клонирования

- ◇ **Пример.** Клонированная версия кода примера 12.2.4 показана на следующем слайде
Для процедуры **g** создается клон для представления всех экземпляров **g**, впервые вызываемых из точек вызова **c1**, **c2** и **c3**.

В этом случае анализ может определить, что вызов в точке **c1** возвращает **1**

(в предположении, что этот анализ в состоянии вывести, что при **v = 0** результат проверки **v > 1** отрицательный).

Однако этот анализ недостаточно хорош, чтобы вычислить константы для точек вызова **c2** и **c3**.

12.2. Межпроцедурный анализ

Исходная программа	Клонированная программа
<pre> for (i = 0; i < n; i++) { c1: t1 = g{0); c2: t2 = g(243); c3: t3 = g(243); X[i] = t1 + t2 + t3; } int g (int v) { if (v > 1) { c4: return g(v-1); } else { c5: return f(v); } } int f (int v) { return (v+1); }</pre>	<pre> for (i =0; i < n; i++) { c1: t1 = g1(0); c2: t2 = g2(243); c3: t3 = g3(243); X[i] = t1+t2+t3; } int g1 (int v) { if (v > 1) { c4.1: return g1(v-1); } else { c5.1: return f1(v); } } int g2 (int v) { if (v > 1) { c4.2: return g2(v-1); } else { c5.2: return f2(v); } } int g3 (int v) { if (v > 1) { c4.3: return g3(v-1); } else { c5.3: return f3(v); } } int f1 (int v) {return (v+1); } int f2 (int v) {return (v+1); } int f3 (int v) {return (v+1);}</pre>

12.2. Межпроцедурный анализ

12.2.6 Контекстно-чувствительный анализ на основе аннотаций

- ◇ Межпроцедурный анализ на основе аннотаций представляет собой обобщение анализа на основе областей.
По сути, в анализе на основе аннотаций каждая область представлена кратким описанием («*аннотацией*»), которое концентрирует некоторое наблюдаемое поведение области. В рассматриваемом случае роль областей играют процедуры. Основная цель аннотации – избежать повторного анализа тела процедуры в каждой точке, где эта процедура может быть вызвана. Это позволяет сделать учет особенностей контекста вызова не слишком дорогим.
- ◇ Каждая процедура моделируется как область с единой точкой входа, при этом все пары «вызывающая/вызываемая процедуры» описываются отношением «внешняя/внутренняя области». Единственное отличие от внутрипроцедурной версии заключается в том, что в межпроцедурном случае область процедуры может быть вложена внутрь нескольких различных внешних областей. (В этом случае может помочь клонирование: в каждую область будет вложен свой экземпляр процедуры).

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

- ◇ Анализ (как и в случае анализа на основе областей) состоит из двух фаз:
 - ◇ *восходящая фаза*, на которой вычисляются передаточные функции для аннотирования действий процедуры («распространение аннотаций»);
 - ◇ *нисходящая фаза*, на которой информация вызывающей процедуры передается вызываемым процедурам для вычисления результатов их выполнения.
- ◇ Для полного учета контекстной чувствительности информация от разных контекстов вызова должна распространяться вниз к отдельным вызываемым функциям.
- ◇ Для повышения эффективности (с понижением точности) можно сначала объединять информацию от всех вызывающих функций с использованием оператора сбора, а затем передавать ее вниз вызываемым функциям.

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ **Пример 1.** Для распространения констант каждая процедура аннотируется передаточной функцией, определяющей, как константы распространяются через тело этой процедуры.

◇ Во фрагменте программы

```
for (i = 0; i < n; i++) {  
    c1:    t1 = f(0) ;  
    c2:    t2 = f(243) ;  
    c3:    t3 = f(243) ;  
           x[i] = t1 + t2 + t3;  
}
```

```
int f(int v) {return (v+1)}
```

для функции **f** можно составить аннотацию:

"v == c $\Rightarrow$ f(v) == c + 1"

(если фактическим параметром функции **f** является константа **c**, то возвращаемым значением будет константа **c + 1**)

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ **Пример 1.** Для распространения констант каждая процедура аннотируется передаточной функцией, определяющей, как константы распространяются через тело этой процедуры.

◇ Во фрагменте программы

```
for (i = 0; i < n; i++) {  
    c1:    t1 = f(0) ;  
    c2:    t2 = f(243) ;  
    c3:    t3 = f(243) ;  
          x[i] = t1 + t2 + t3;  
}
```

```
int f(int v) {return (v+1)}
```

для функции **f** можно составить аннотацию:

"v == c $\Rightarrow$ f(v) == c + 1"

На основе этой аннотации анализ может определить, что переменные **t1**, **t2** и **t3** получат значения **1**, **244** и **244** соответственно.

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ **Пример 2.** Рассмотрим фрагмент программы:

```

        for (i = 0; i < n; i++) {
c1:            t1 =g(0) ;
c2:            t2 =g(243) ;
c3:            t3 =g(243) ;
                x[i]= t1 + t2 + t3;
        }

    int g(int v) {
c4:        return f(v) ;
    }

    int f (int v) {
        return (v+1) ;
    }
```

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ **Пример 2.** Рассмотрим фрагмент программы:

```
    for(i = 0; i < n; i++) {  
c1:      t1 =g(0) ;  
c2:      t2 =g(243) ;  
c3:      t3 =g(243) ;  
          X[i]= t1 + t2 + t3;  
          }  
          int g(int v) {  
c4:              return f(v) ;  
          }  
          int f (int v) {  
                  return (v+1) ;  
          }
```

Этот фрагмент отличается от фрагмента из предыдущего примера тем, что в нем добавлена функция **g**, которая вызывает функцию **f**.

Следовательно, **передаточная функция для g** такая же, как и **передаточная функция для f**, что и будет указано в аннотации. Прочитав во время анализа эту аннотацию, можно будет сделать вывод: переменные **t1**, **t2** и **t3** получают значения **1**, **244** и **244** соответственно.

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ Пример 1. (продолжение)

Анализ позволяет понять, что $v == \text{NAC}$

(в точке **c1** $v == 0$, в точке **c2** $v == 243$)

Можно создать два специализированных клона функции **f** для входных значений **0** и **243** соответственно

```
                for (i =0; i < n; i++) {
c1:                t1 = f0(0);
c2:                t2 = f243(243);
c3:                t3 = f243(243);
                    X[i] = t1+t2+t3;
                }
int f0 (int v) {
                    return (1);
                }
int f243 (int v) {
                    return (244);
                }
```

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ Пример 1. (продолжение)

Анализ позволяет понять, что $v == \text{NAC}$

(в точке **c1** $v == 0$, в точке **c2** $v == 243$)

Можно создать два специализированных клона функции **f** для входных значений **0** и **243** соответственно

Отличие от КЧА на основе клонирования:
в случае КЧА на основе клонирования клонирование выполняется до анализа, на основе строк вызовов;
в случае КЧА на основе аннотаций клонирование выполняется после анализа, на основе его результатов.

```
for (i =0; i < n; i++) {  
c1:    t1 = f0(0);  
c2:    t2 = f243(243);  
c3:    t3 = f243(243);  
       x[i] = t1+t2+t3;  
}  
int f0 (int v) {  
    return (1);  
}  
int f243 (int v) {  
    return (244);  
}
```

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

◇ Пример 1. (продолжение)

Анализ позволяет понять, что $v == \text{NAC}$

(в точке $c1$ $v == 0$, в точке $c2$ $v == 243$)

Можно создать два специализированных клона функции f для входных значений 0 и 243 соответственно

Вывод: Только контекстно-чувствительный анализ дает значимые результаты, но он требует большого времени выполнения и существенных расходов памяти, так как предполагает новый анализ функции при каждом ее вызове. Это связано с зависимостью фактических параметров вызова от контекста в точке вызова. Методы, рассмотренные выше позволяют сократить количество анализов функции, учитывая особенности каждого её вызова.

```
for (i =0; i < n; i++) {  
c1:    t1 = f0(0) ;  
c2:    t2 = f243(243) ;  
c3:    t3 = f243(243) ;  
        X[i] = t1+t2+t3;  
}  
int f0 (int v) {  
    return (1) ;  
}  
int f243 (int v) {  
    return (244) ;  
}
```

12.2. Межпроцедурный анализ

12.2.7 Контекстно-чувствительный анализ на основе аннотаций

- ◇ Отличие от предыдущего варианта КЧА заключается в том, что в случае КЧА на основе клонирования клонирование выполняется до анализа, на основе строк вызовов, а в случае КЧА на основе аннотаций клонирование выполняется после анализа, на основе его результатов.
- ◇ Даже если клонирование не применяется, выводы КЧА на основе аннотаций о результатах выполнения вызываемой процедуры делаются точно, без проблемы нереализуемых путей.
- ◇ Клонирование функций можно заменить открытой вставкой их кода. Открытая вставка дает дополнительный эффект, устраняя накладные расходы, связанные с вызовами процедур.
- ◇ Справиться с рекурсией можно с помощью итераций.

Вывод: Только контекстно-чувствительный анализ дает значимые результаты, но он требует значительного времени выполнения и существенных расходов памяти, так как предполагает новый анализ функции при каждом ее вызове. Это связано с зависимостью фактических параметров вызова от контекста в точке вызова. Методы, рассмотренные выше позволяют сократить количество анализов функции, учитывая особенности каждого её вызова.

12.3. Межпроцедурное распространение констант

12.3.1. Введение

- ◇ Межпроцедурное распространение констант отслеживает **известные константные значения** глобальных переменных и параметров процедур при их распространении по графу вызовов как через тела процедур, так и вдоль ребер графа вызовов.
- ◇ Цель межпроцедурного распространения констант – выявить ситуации, когда процедура всегда получает известное константное значение или когда она всегда возвращает такое значение. Когда обнаруживается такое значение (константа), анализ может преобразовать код с учетом обнаруженного значения.
- ◇ Концептуально межпроцедурное распространение констант сводится к решению следующих трех подзадач:
 - ◇ Выявление исходного множества константных значений
 - ◇ Распространение известных константных значений по графу вызовов
 - ◇ Моделирование передачи значений через процедуры.

12.3. Межпроцедурное распространение констант

12.3.2. Выявление исходного множества констант

◇ В каждой точке вызова анализатор должен определить, какие параметры и (или) глобальные переменные имеют известные константные значения.

Для решения этой подзадачи существует много методов от выявления фактических параметров, являющихся литеральными константами, до многократного выполнения процедуры глобального распространения констант (что, конечно, делает анализ намного более затратным).

12.3. Межпроцедурное распространение констант

12.3.3. Распространение известных константных значений по графу вызовов



Получив начальное множество констант, анализатор распространяет константные значения вдоль ребер графа вызовов и через каждую процедуру от ее входа (*Entry*) до каждой точки вызова других процедур, имеющейся в рассматриваемой процедуре.

По существу это – усложненный итеративный анализ потока данных, который требует намного больше итераций, чем в случае анализа достигающих определений или живых переменных.

12.3. Межпроцедурное распространение констант

12.3.4. Моделирование передачи значений через процедуры

- ◇ Во время обработки каждой вершины графа вызовов анализатор должен определять **как константные значения, известные на входе в процедуру, влияют на множество константных значений в каждой точке вызова.**

Это позволяют сделать небольшие модели, называемые *функциями скачка*.

Для каждого фактического параметра строится своя функция скачка, так что точке вызова s с n параметрами соответствует вектор функций скачка $J_s = (J_s^{p_1}, J_s^{p_2}, \dots, J_s^{p_n})$, где p_i ($i = 1, \dots, n$) – фактические параметры вызова в точке s .

Каждая функция скачка $J_s^{p_i}$ может зависеть не только от p_i , но и от некоторого подмножества других формальных параметров вызова. Это подмножество будем обозначать $Support(J_s^{p_i})$. Требуется, чтобы функция $J_s^{p_i}$ возвращала значение *Undef*, если хотя бы один параметр из $Support(J_s^{p_i})$ имеет значение *Undef*.

12.3. Межпроцедурное распространение констант

12.3.5. Алгоритм

- ◇ Рассмотрим простой алгоритм межпроцедурного распространения констант. Он похож на алгоритм глобального распространения констант, рассмотренный в разделе 7.2.
- ◇ На фазе инициализации все значения фактических параметров полагаются равными *Undef*.
- ◇ После этого выполняются итерации для каждого фактического параметра a в каждой точке вызова s , в результате чего значения фактических параметров уточняются, а уточненные параметры снова помещаются в *Worklist*.
- ◇ В конце фазы инициализации образуется начальное множество констант, представленных функциями скачка, а в *Worklist* записывается список всех формальных параметров.
- ◇ Псевдокод первой фазы на следующем слайде.

12.3. Межпроцедурное распространение констант

12.3.5. Алгоритм

// Фаза 1: Инициализация

// Построение всех функций скачка и поддержка отображений

Worklist $\leftarrow \emptyset$;

for each procedure p in the program

 for each formal parameter f to p

 Value(f) $\leftarrow \top$ // $\top = \text{Undef}$ (начальное значение)

 Worklist $\leftarrow \text{Worklist} \cup \{f\}$

for each call site s in the program

 for each formal parameter f that receives value at s

 Value(f) $\leftarrow \text{Value}(f) \wedge J_s^f$ // Учет J_s^f в начальных значениях

call site – точка вызова

12.3. Межпроцедурное распространение констант

12.3.5. Алгоритм

- ◇ **Вторая фаза** многократно выбирает формальные параметры из *Worklist* и распространяет их.

Для того, чтобы распространить формальный параметр f процедуры p анализатор находит каждую точку вызова s процедуры p и каждый формальный параметр x (который соответствует фактическому параметру точки вызова s), такой что $f \in \text{Support}(\mathcal{J}_s^x)$. Затем вычисляется $\mathcal{J}_s^x$ и полученное значение сравнивается с текущим значением x .

Если сравниваемые значения не совпадают, значение x заменяется и x добавляется в *Worklist*.

- ◇ Вторая фаза завершается, так как каждый параметр может принимать всего три значения: $\top$, *константа* и $\perp$
- ◇ Псевдокод второй фазы на следующем слайде.

12.3. Межпроцедурное распространение констант

12.3.5. Алгоритм

// Фаза 2: Выполнение итераций

while (Worklist $\neq \emptyset$;)

 pick parameter f from Worklist // выбрать параметр

 let p be the procedure declaring f

 // Обновить значение каждого параметра функции f

 for each call site s in p and parameter x such that

$f \in \text{Support}(\mathcal{I}_s^x)$

$t \leftarrow \text{Value}(x)$

$\text{Value}(x) \leftarrow \text{Value}(x) \wedge \mathcal{I}_s^x$ // Новое значение

 if ($\text{Value}(x) < t$)

 then $\text{Worklist} \leftarrow \text{Worklist} \cup \{x\}$

12.3. Межпроцедурное распространение констант

12.3.5. Алгоритм

```
// Post-process Val формирует множество CONSTANTS
for each procedure p
CONSTANTS(p)  $\leftarrow \emptyset$ ;
    for each formal parameter f to p
        if (Value(f) = Undef)
            then Value(f)  $\leftarrow$  NAC;
        if (Value(f)  $\neq$  NAC)
            then CONSTANTS(p)  $\leftarrow$ 
                CONSTANTS(p)  $\cup \{ (f, \text{Value}(f)) \}$ 
```

12.3. Межпроцедурное распространение констант

12.3.6. Реализация функций скачка

- ◇ Реализации функций скачка варьируются от простых статических приближений, которые не изменяются во время анализа (например, аннотации), либо небольших параметризованных моделей (например, клоны, или строки вызова), до более сложных схем, которые выполняют глубокий всесторонний анализ при каждом вычислении функции скачка.
- ◇ Для каждой из таких схем выполняется несколько простых принципов.
 - ◇ Если анализатор выясняет, что параметр $\mathbf{x}$ в точке вызова $\mathbf{s}$ равен известной константе $\mathbf{c}$, то $J_s^x = c$ и $Support(J_s^x) = \emptyset$
 - ◇ Если $y \in Support(J_s^x)$ и $Value(y) = Undefined$, то $J_s^x = Undefined$
 - ◇ Если анализатор выясняет, что значение J_s^x невозможно определить, то $J_s^x = NAC$

12.3. Межпроцедурное распространение констант

12.3.6. Реализация функций скачка

- ◇ Анализатор может реализовать J_s^x многими способами.
 - ◇ Самая простая реализация может распространять константу только тогда, когда x является SSA-именем формального параметра процедуры, вызываемой в точке s .
 - ◇ Более сложная реализация может строить выражения из SSA-имен формальных параметров и литеральных констант.
 - ◇ Наиболее точная, но дорогая реализация может для обновления значений функций скачка вызывать программу, выполняющую алгоритм глобального распространения констант.

12.3. Межпроцедурное распространение констант

12.3.7. Учет возвращаемых значений

- ◇ Рассмотренный алгоритм межпроцедурного распространения констант учитывает только фактические параметры вызываемых процедур.
- ◇ Его можно расширить очевидным образом, чтобы он учитывал и возвращаемые значения, а также значения глобальных переменных.
- ◇ Для этого необходимо определить *обратные функции скачка*, моделирующие передачу значений от вызываемой функции к вызывающей.
- ◇ Обратные функции скачка важны, когда анализируются функции, используемые для инициализации значений, а не для заполнения полей *Java*-класса

13. Выбор команд

13.1 Вводные замечания

13.12.1 Постановка задачи

- ◇ Оптимизатор работает над промежуточным представлением программы, и когда процесс оптимизации закончится, получится оптимизированная программа в промежуточном представлении.
- ◇ На следующем этапе необходимо перевести программу из промежуточного представления в код целевого процессора.
- ◇ Отображение *инструкций* промежуточного представления в *команды* целевого процессора называется *выбором команд*.
- ◇ Существует несколько подходов к решению этой задачи. Рассмотрим подход, использующий сопоставление шаблонов на абстрактном синтаксическом дереве, построенном по оптимизированному промежуточному представлению.

13.1 Вводные замечания

13.1. 2 Вход и выход генератора кода

- ◇ Входной поток генератора кода:
 - ◇ *промежуточное представление* исходной программы (последовательность трехадресных инструкций)
 - ◇ *таблица символов*, которая используется для определения адресов времени выполнения объектов данных, обозначаемых в промежуточном представлении именами.
- ◇ Выходной поток генератора кода: *объектный код* (последовательность команд *целевого процессора*)
- ◇ *Объектный модуль* – часть программы в объектном коде, соответствующая исходному модулю
- ◇ *Компоновщик* объединяет объектные модули и библиотеки в единую программу, которая и выполняется на целевом процессоре.

13.2 Объектный код

13.2.1 Общая характеристика

Предположения о целевом процессоре (ядре)

- ◇ *RISC*-архитектура
- ◇ Только целочисленная арифметика.
- ◇ Байтовая адресация памяти
- ◇ n регистров общего назначения $R_0, R_1, \dots, R_{n-1}$
- ◇ Команда: **op, dst, src1, src2**
- ◇ Операции:
 - ◇ загрузки и сохранения регистров,
 - ◇ вычислительные,
 - ◇ условные и безусловные переходы.

13.2 Объектный код

13.2.1 Общая характеристика

Предположения о целевом про

- ◇ *RISC*-архитектура
- ◇ Только целочисленная
- ◇ Байтовая адресация па
- ◇ n регистров общего наз
- ◇ Команда: **op**, **dst**,
- ◇ Операции:
 - ◇ загрузки и сохранения регистров,
 - ◇ вычислительные,
 - ◇ условные и безусловные переходы.

RISC (Reduced Instruction Set Computer).
Процессор с сокращенным набором команд.
Система команд имеет упрощенный вид:

- ◇ Все команды одинакового формата с простой кодировкой.
- ◇ Обращение к памяти происходит посредством команд загрузки (**LD**) и записи (**ST**).
- ◇ Остальные команды типа **RR**.
- ◇ Команда, поступающая в CPU, уже разделена по полям и не требует дополнительной дешифрации.

13.2 Объектный код

13.2.2 Система команд

LD, r, x	Загрузка значения из ячейки памяти x на регистр r
LD, r1, r2	Копирование значения регистра r1 на регистр r2
ST, x, r	Сохранение значения регистра r в ячейке памяти x
OP ¹ dst, src1, src2	Бинарная операция: dst = src1 OP src2
OP ² dst, src1	Унарная операция: dst = OP src1
BR L	Переход на команду с меткой L (goto L).
Bcond ³ r, L	Ветвление: переход на команду с меткой L , если значение на регистре r удовлетворяет условию cond : if (cond) goto L

¹ определены арифметические, булевы и поразрядные операции

² определены арифметические, булевы и поразрядные операции

³ формирование условий выполняется с помощью операций отношения и арифметических операций

13.2 Объектный код

13.2.3 Режимы адресации

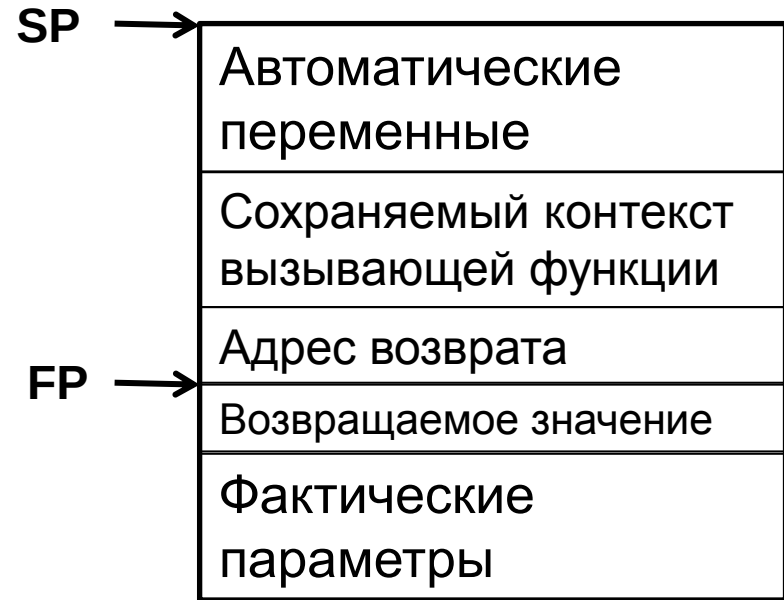
- ◇ *Режим прямой адресации* – адрес задается непосредственно значением **x**
- ◇ *Режим индексированной адресации* **a(r)**, **a** – переменная, **r** – регистр. Адрес **a(r)** вычисляется путем прибавления к **l**-значению **a** значения из регистра **r**.
Этот режим адресации полезен для обращения к массивам:
a – базовый адрес массива, **r** – смещение.
- ◇ *Режим косвенной адресации:*
***r** ссылка на ячейку памяти, находящуюся по адресу
contents(r) * c(r) ссылка на ячейку памяти, находящуюся по адресу **c + contents(r)**
(**c** – константа, **contents** – операция разыменования)
- ◇ *Режим адресации с использованием констант*,
для указания которых используется префикс **#**.
Например, команда **LD r, #n** загружает в регистр **r** целое число **n**

13.3 Основные фазы генерации кода

- ◇ Распределение памяти
- ◇ Выбор команд
- ◇ Распределение регистров
- ◇ Выбор оптимального порядка команд (планирование кода)

13.3 Основные фазы генерации кода

13.3.1. Распределение памяти



Статическая память состоит из «фреймов», динамически выделяемых в стэке при вызове процедуры и удаляемых при выходе из нее. Структура фрейма показана на правом рисунке.

13.4 Выбор команд

13.4.1 Постановка задачи

- ◇ Если не требовать эффективности целевой программы, выбор команд предельно прост:
 - ◇ для каждого типа трехадресных инструкций определяется шаблон целевого кода, генерируемого для таких инструкций;
 - ◇ каждая трехадресная инструкция заменяется соответствующим шаблоном
- ◇ Рассмотрим несколько простых примеров

13.4 Выбор команд

13.4.2. Примеры шаблонов

◇ **Пример 1.** При генерации кода для трехадресной инструкции $\mathbf{x} \leftarrow +, \mathbf{y}, \mathbf{z}$, где память под $\mathbf{x}$, $\mathbf{y}$ и $\mathbf{z}$ выделяется статически (**static**), можно использовать следующий шаблон:

```
LD    R0,  y           // загрузить y в R0
ADD   R0,  R0,  z       // сложить z и R0
ST    x,    R0         // сохранить R0 в x
```

Применим этот шаблон к последовательности из двух инструкций $\mathbf{a} \leftarrow +, \mathbf{b}, \mathbf{c}; \mathbf{d} \leftarrow +, \mathbf{a}, \mathbf{e};$

```
LD    R0,  b           // R0 ← b
ADD   R0,  R0,  c       // R0 ← +, R0, c
ST    a,    R0         // a ← R0
LD    R0,  a           // R0 ← a
ADD   R0,  R0,  e       // R0 ← R0 + e
ST    d,    R0         // d ← R0
```

13.4 Выбор команд

13.4.2. Примеры шаблонов

- ◇ **Пример 2.** Многие компьютеры под влиянием языка Си имеют команду **INC x**:
- ◇ Ей соответствует шаблон

```
INC    x           // x++
```

- ◇ Очевидно, что использование шаблона **INC** дает лучший код, чем использование шаблона из примера 1

```
LD    R0,    x           // загрузить x в  $R_0$ 
ADD   R0,    R0,    1     // сложить 1 и  $R_0$ 
ST    x,     R0           // сохранить  $R_0$  в x
```

13.4 Выбор команд

13.4.2. Примеры шаблонов

◇ Пример 3. Шаблон для **+, x, y, z** (**x, y, z** – адреса ячеек памяти)

```
LD    R1, y
LD    R2, z
ADD   R1, R1, R2
ST    x, R1
```

◇ Пример 4. Шаблон для **ifTrue(y < z) gotoL**

```
LD      R1, y
LD      R2, z
SUB     R1, R1, R2
BLTZ   R1, L
```

13.4 Выбор команд

13.4.2. Примеры шаблонов



Пример 5. Шаблон для

(a) **$b \leftarrow a[i]$** (**a** – массив 8-байтных значений)

LD R1, i

MUL R1, R1, #8

LD R2, a(R1)

// $R2 \leftarrow \text{contents}(a + \text{contents}(i))$

ST b, R2

(b) **$a[k] \leftarrow c$**

LD R1, c

LD R2, k

MUL R2, R2, #8

ST a(R2), R1

// $\text{contents}(a + \text{contents}(k)) \leftarrow R1$

13.4 Выбор команд

13.4.2. Примеры шаблонов

◇ Пример 6. Шаблоны для

(a) $x \leftarrow *p$

LD R1, p

LD R2, 0(R1)

ST x, R2

//R2 ← contents(0 + contents(R1))

(b) $*p \leftarrow y$

LD R1, p

LD R2, y

ST 0(R1), R2

//contents(0 + contents(R1)) ← R2

13.4 Выбор команд

13.4.3 Стоимость команд

- ◇ Каждая команда целевого языка имеет связанную с ней стоимость.
- ◇ Если считать стоимость команды равной единице плюс стоимости, связанные с режимами адресации операндов, то стоимость будет пропорциональна длине команды в словах:
 - ◇ Стоимость операнда на регистре равна 0
 - ◇ Стоимость операнда из ячейки памяти равна 1
 - ◇ Стоимость операнда-константы равна 1Такой подход обосновывается тем, что адреса ячеек памяти и константы хранятся в словах, следующих за командой
- ◇ Стоимость каждого разыменования равна 1

13.4 Выбор команд

13.4.3 Стоимость команд



Примеры.

- 1) Стоимость команды **LD R0, R1** равна 1.
- 2) Стоимость команды **LD R0, M** (**M** – адрес ячейки памяти) равна 2.
- 3) Стоимость команды **LD R0, *100(R1)**
($R_0 \leftarrow contents(contents(100 + contents(R_1)))$) равна 4.



Стоимость программы (на целевом языке) равна сумме стоимостей ее команд. Чем меньше стоимость программы, тем быстрее она выполняется



Алгоритм генерации кода должен минимизировать стоимость программы.

13.5 Метод переписывания дерева

13.5.1 Схема трансляции деревьев

- ◇ Рассмотрим инструкцию присваивания

$$\mathbf{a}[\mathbf{i}] \leftarrow +, \mathbf{b}, 1 \quad (1)$$

- ◇ Пусть

- ◆ массив **a** хранится в стеке времени выполнения
(адреса времени выполнения локальных переменных **a** и **i** заданы как смещения C_a и C_i относительно указателя на начало текущей *записи активации* **SP**
(адрес **SP** хранится на специальном регистре $\mathbf{R}_{\mathbf{SP}}$)
- ◆ переменная **b** хранится в глобальной ячейке памяти M_b .

- ◇ В целевом коде инструкции (1) будет соответствовать последовательность команд :

13.5 Метод переписывания дерева

13.5.1 Схема трансляции деревьев

- ◇ В целевом коде инструкции $\mathbf{a[i] \leftarrow +, b, 1}$ будет соответствовать последовательность команд :
- ```
LD R1, Mb // R1 = b
ADD R1, R1, #1 // R1 = b + 1
LD R2, Ci (RSP)
 // R2 = contents(Ci + contents(RSP))
MUL R2, R2, 8 // R2 = R2 * 8
ST *R2 (Ca (RSP)) , R1
 // R1 = contents(contents(Ca +
 contents(RSP)) + contents(R2))
```
- ◇ Введем операцию индексирования  $\mathbf{ind(index)}$ :
- $$\mathbf{ind(C_i + R_{SP}) =}$$
- $$\mathbf{contents(C_i + contents(R_{SP})) = contents(R2) = i}$$
- $$\mathbf{ind((C_a + R_{SP}) + ind(C_i + R_{SP})) =}$$
- $$\mathbf{contents(contents(C_a + contents(R_{SP})) + contents(R2))}$$
- $$\mathbf{= a[i]}$$

## 13.5 Метод переписывания дерева

### 13.5.1 Схема трансляции деревьев

$a[i] \leftarrow +, b, 1$

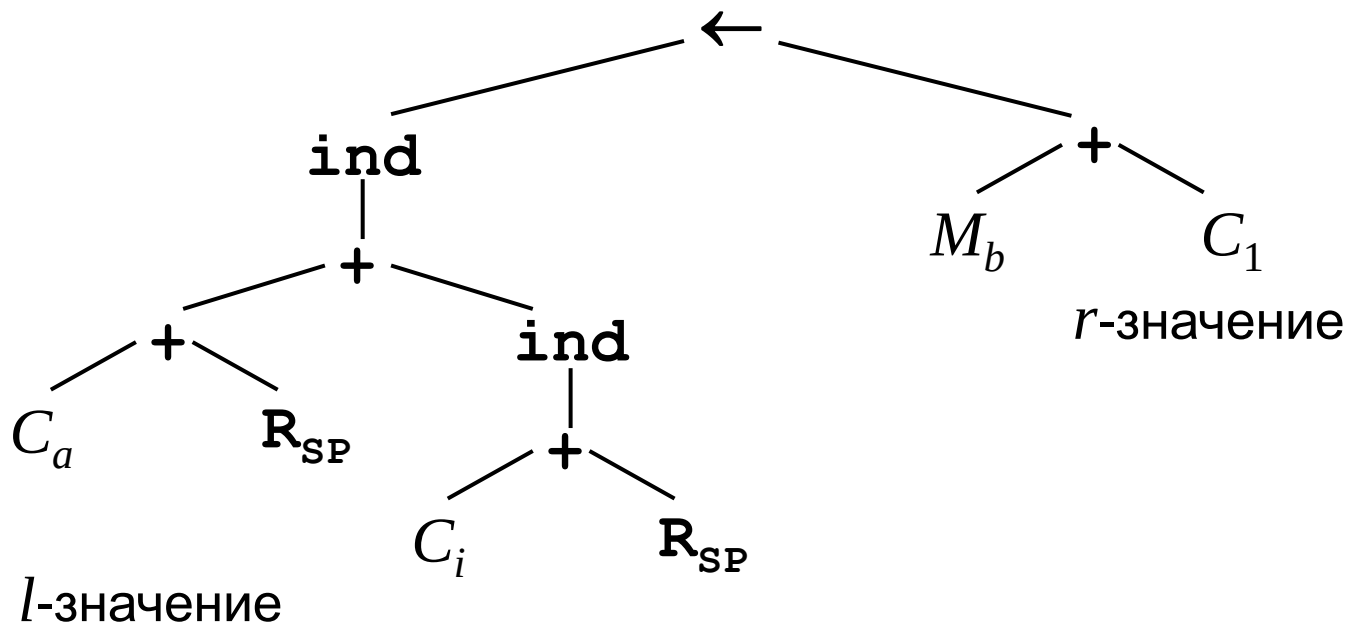
$\text{ind}(C_i + R_{SP}) =$

$\text{contents}(C_i + \text{contents}(R_{SP})) = \text{contents}(R_2) = i$

$\text{ind}((C_a + R_{SP}) + \text{ind}(C_i + R_{SP})) =$

$\text{contents}(\text{contents}(C_a + \text{contents}(R_{SP})) + \text{contents}(R_2)) = a[i]$

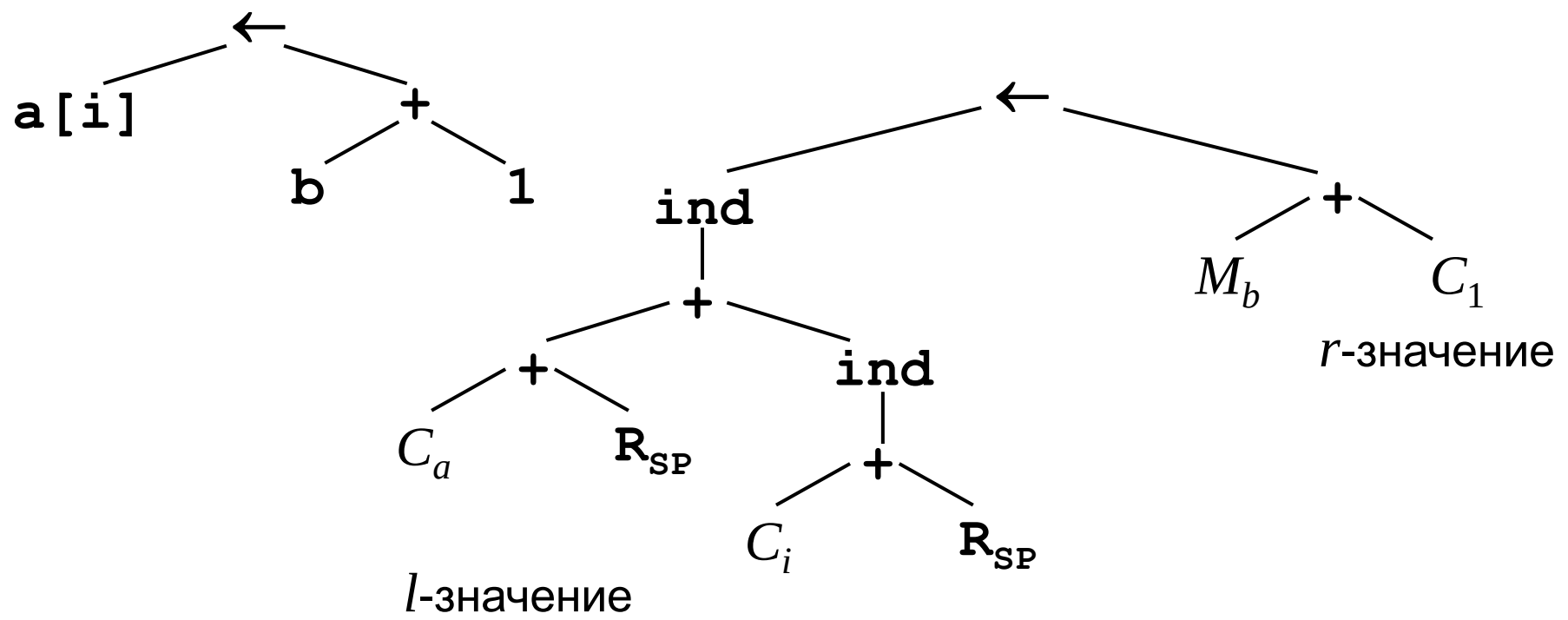
$+(M_b C_1) = +, b, \#1$



# 13.5 Метод переписывания дерева

## 13.5.1 Схема трансляции деревьев

Таким образом, исходное дерево (выражение) на левом рисунке должно быть преобразовано (переписано) в дерево на правом рисунке



## 13.5 Метод переписывания дерева

### 13.5.2 Описание метода

- ◇ Целевой код генерируется в процессе *свертки входного дерева в единый узел* путем последовательного применения *правил преобразования дерева*.
- ◇ Каждое *правило преобразования дерева* представляет собой инструкцию вида

$$replacement \leftarrow template \{action\}$$

где *replacement* – отдельный узел, *template* – шаблон (поддерево), *action* – действие (фрагмент генерируемого кода, соответствующий шаблону).

- ◇ Множество правил преобразования дерева именуется *схемой трансляции дерева*.
- ◇ Трансляция состоит из (возможно, пустой) последовательности машинных команд, которые генерируются действием (*action*), связанным с шаблоном (*template*).

## 13.5 Метод переписывания дерева

### 13.5.3 Схема трансляции дерева

- ◇ Схема трансляции дерева – удобный способ описания схемы выбора команд в генераторе кода.
- ◇ Схема трансляции задается набором *правил свертки*
- ◇ Каждое правило содержит:
  - ◇ поддерево, соответствующее рассматриваемому шаблону
  - ◇ метку узла, на который заменяется поддерево при свертке
  - ◇ команды, которые помещаются в объектную программу при свертке

**Пример правила:** правило для команды сложения одного регистра с другим имеет вид:



если входное дерево содержит поддерево, соответствующее данному шаблону, то это поддерево можно заменить одним узлом с меткой  $R_i$  и сгенерировать команду

**ADD Ri, Ri, Rj.**

Такая замена называется *замещением поддерева*.

## 13.5 Метод переписывания дерева

### 13.5.3 Схема трансляции дерева

- ◇ Листья как входного дерева, так и шаблона могут иметь атрибуты с индексами; иногда к значениям индексов применяется ряд ограничений – семантических предикатов, которым должен удовлетворять шаблон при установлении соответствия.  
**Пример предиката:** значение константы должно находиться в определенном диапазоне.
- ◇ Выбор команд ведется в следующих предположениях:
  - ◇ Любая инструкция промежуточного кода может быть реализована с помощью одной или нескольких машинных команд.
  - ◇ Имеется достаточно регистров для вычисления каждого узла.



## 13.5 Метод переписывания дерева

### 13.5.4 Пример

#### Правила свертки

На рисунке приведены правила свертки для некоторых команд целевой машины:

♦ правила 1) и 2) относятся к инструкциям загрузки (**LD**)

♦ правила 3) и 4) – инструкциям сохранения(**ST**)

♦ правила 5) и 6) – индексированным загрузкам  
♦ правила 7) и 8) сложениям (**ADD**)

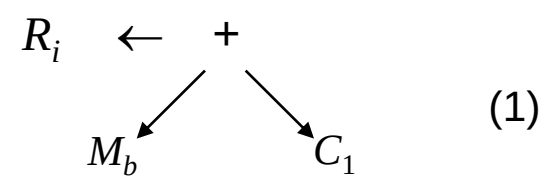
|    |                                                                                                                                                                                  |                     |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| 1) | $R_i \leftarrow C_a$                                                                                                                                                             | {LD Ri, #a}         |
| 2) | $R_i \leftarrow M_x$                                                                                                                                                             | {LD Ri, x}          |
| 3) | $M_x \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array}$                                                                                      | {ST x, Ri}          |
| 4) | $M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ \text{ind} \quad R_j \\   \\ R_i \end{array}$                                                                     | {ST *Ri, Rj}        |
| 5) | $R_i \leftarrow \begin{array}{c} \text{ind} \\   \\ + \\ \swarrow \quad \searrow \\ C_a \quad R_j \end{array}$                                                                   | {LD Ri, a(Rj)}      |
| 6) | $R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad \text{ind} \\ \quad   \\ \quad + \\ \quad \swarrow \quad \searrow \\ \quad C_a \quad R_j \end{array}$ | {ADD Ri, Ri, a(Rj)} |
| 7) | $R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad R_j \end{array}$                                                                                      | {ADD Ri, Ri, Rj}    |
| 8) | $R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad C_1 \end{array}$                                                                                      | {INC Ri}            |

# 13.5 Метод переписывания дерева

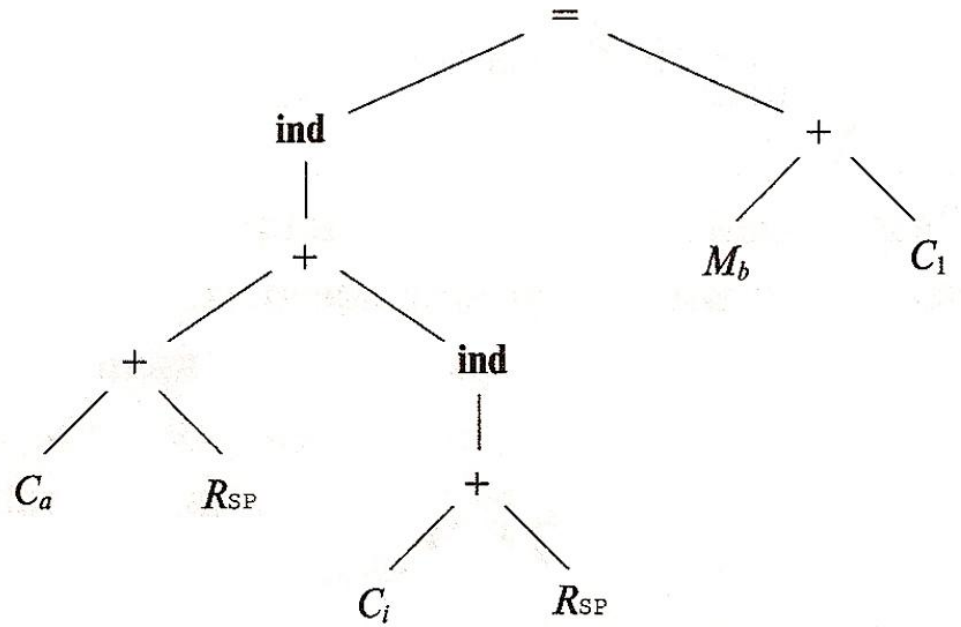
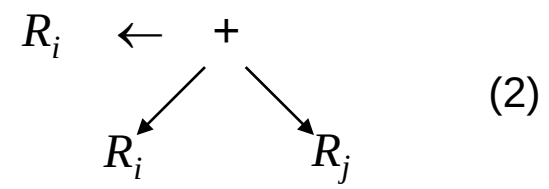
## 13.5.4 Пример

Начнем применять схему трансляции дерева к дереву из примера 13.5.1 слева направо.

Среди шаблонов нет шаблона



есть только шаблон



следовательно, чтобы свернуть шаблон (1) необходимо с помощью правил 1) и 2) привести его к виду (2)  
правило 1)  $R_i \leftarrow C_a \{LD Ri \#a\}$  позволяет заменить  $C_1$  на  $R_0$   
правило 2)  $R_i \leftarrow M_x \{LD Ri \ x\}$  позволяет заменить  $M_b$  на  $R_1$   
после этих замен шаблон принимает вид (2), что позволяет применить правило 7)

# 13.5 Метод переписывания дерева

## 13.5.4 Пример

Применим приведенную схему трансляции дерева для генерации кода для примера 13.5.1.

Правило 1) позволяет привести шаблон

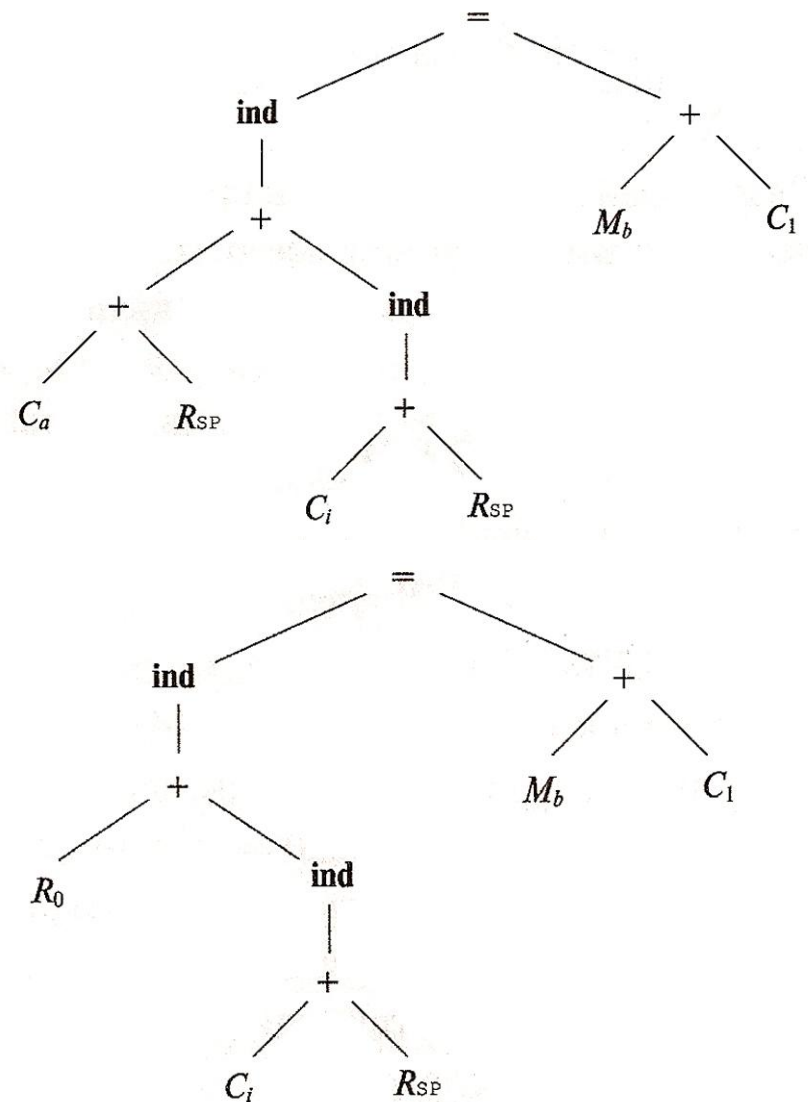


выдав команду  
**{LD R0 #a}**

Теперь можно с помощью правила 7)  
привести исходное дерево к виду  
(нижний рисунок) и выдать команду  
**{ADD R0 R0 SP}**

Итак, дерево приведено к виду (нижний  
рисунок) и выданы команды:

**LD R0 #a**  
**ADD R0 R0 SP**

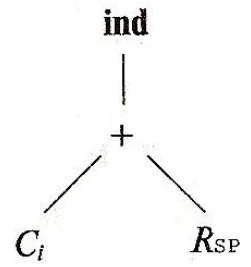


# 13.5 Метод переписывания дерева

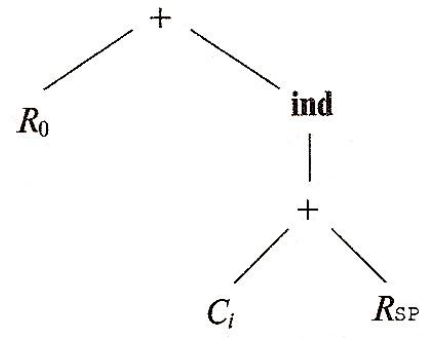
## 13.5.4 Пример

В левом поддереве дерева (верхний рисунок слева)

можно применить правило 5) к шаблону



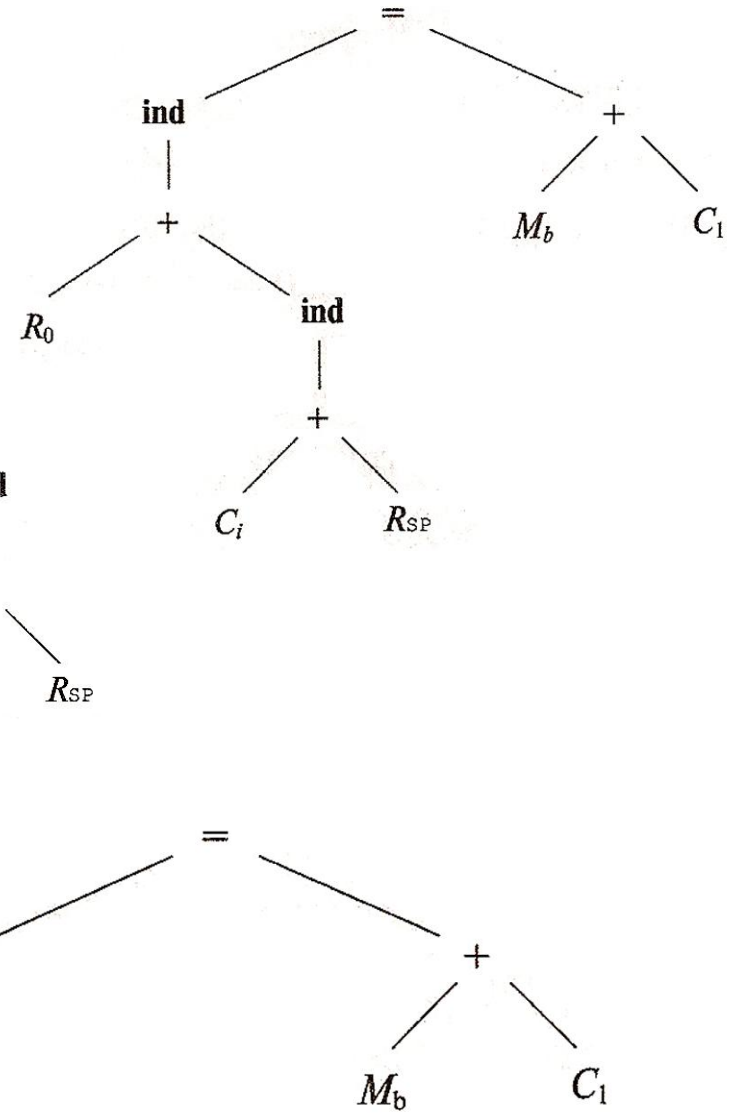
либо правило 6) к шаблону



Выгоднее применить правило 6), так как оно сворачивает большую часть дерева. После применения правила 6) дерево примет вид

при этом будет выдана команда

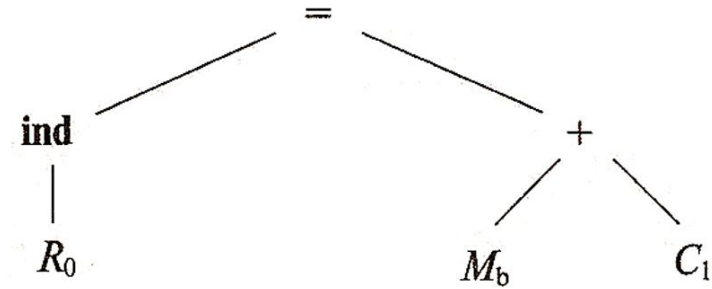
```
ADD R0 R0 i(SP)
```



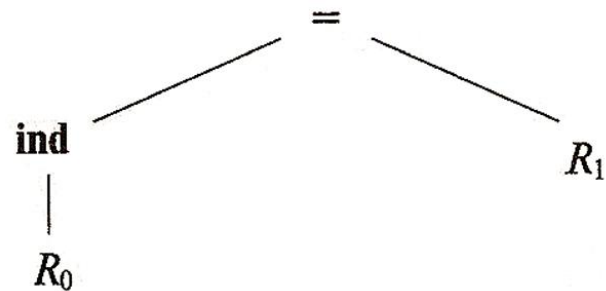
## 13.5 Метод переписывания дерева

### 13.5.4 Пример

Применив к правому поддереву дерева



сначала правило 2), а потом правило 8), получим дерево



Будет выдано две команды:

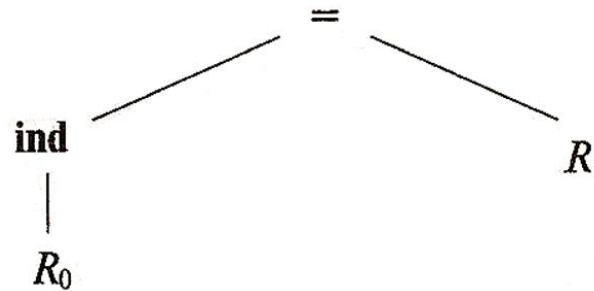
LD R1 b (правило 2))

INC R1 (правило 8))

## 13.5 Метод переписывания дерева

### 13.5.4 Пример

Последнее дерево



соответствует поддереву из правила 4); применение этого правила сворачивает дерево в единственный узел и выдает команду **ST \*R0 R1**

Таким образом в процессе свертки исходного дерева была сгенерирована последовательность команд:

```
LD R0 #a
ADD R0 R0 SP
ADD R0 R0 i (SP)
LD R1 b
INC R1
ST *R0 R1
```

## 13.5 Метод переписывания дерева

### 13.5.5 Проблемы

- ◇ Если на каком-нибудь шаге не будет найдено соответствия ни одному шаблону, процесс генерации кода блокируется.
- ◇ Для реализации процесса свертки, показанного на примере, необходимо решить два вопроса, связанных с поиском соответствия деревьев шаблону:
  - ◇ *Каким образом выполняется поиск соответствия?*  
Эффективность процесса генерации кода в процессе компиляции зависит от того, насколько эффективен применяющийся алгоритм поиска соответствий.
  - ◇ *Что делать, если оказалось, что можно применить более одного шаблона?*  
Эффективность сгенерированного кода может зависеть от порядка, в котором выявлялось соответствие шаблонам, так как различные последовательности соответствий будут приводить к разным последовательностям машинных команд, одни из которых более эффективны, чем другие.

# 13.5 Метод переписывания дерева

## 13.5.6 Поиск соответствий

- ◇ Правила преобразования дерева и входное дерево можно представить в виде строк, используя префиксную польскую запись: сначала записывается операция (корень поддерева), потом ее первый операнд (левое поддерево), потом второй операнд (правое поддерево).
- ◇ Для рассматриваемого примера правила преобразования дерева будут представлены в виде:

|     |                                              |                     |
|-----|----------------------------------------------|---------------------|
| 1)  | $R_i \rightarrow c_a$                        | { LD Ri #a }        |
| 2)  | $R_i \rightarrow M_x$                        | { LD Ri x }         |
| 3)  | $M \rightarrow = M_x R_i$                    | { ST x Ri }         |
| 4)  | $M \rightarrow = \text{ind } R_i R_j$        | { ST *Ri Rj }       |
| 5)  | $R_i \rightarrow \text{ind} + c_a R_j$       | { LD Ri a(Rj) }     |
| 6)  | $R_i \rightarrow + R_i \text{ind} + c_a R_j$ | { ADD Ri Ri a(Rj) } |
| 7)  | $R_i \rightarrow + R_i R_j$                  | { ADD Ri Ri Rj }    |
| 8)  | $R_i \rightarrow + R_i c_1$                  | { INC Ri }          |
| 9)  | $R \rightarrow \text{sp}$                    |                     |
| 10) | $M \rightarrow \text{m}$                     |                     |

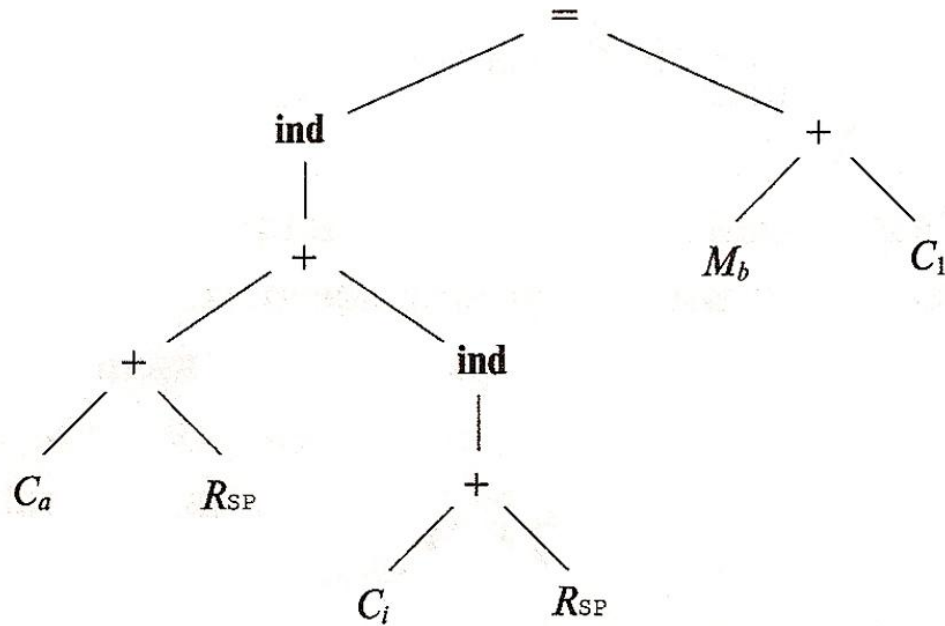


## 13.5 Метод переписывания дерева

### 13.5.6 Поиск соответствий

- ◇ Дерево из примера 13.5.4 в префиксном представлении будет задаваться следующей строкой:

$$= \mathbf{ind} + + C_a R_{SP} \mathbf{ind} + C_i R_{SP} + M_b C_1$$



## 13.5 Метод переписывания дерева

### 13.5.7 Поиск соответствий с помощью синтаксического анализа



Содержимое второго столбца таблицы можно рассматривать как продукции  $LR$ -грамматики. Поэтому соответствия можно искать с помощью  $LR$ -анализатора, используемого при синтаксическом анализе

|     |                                              |                     |
|-----|----------------------------------------------|---------------------|
| 1)  | $R_i \rightarrow c_a$                        | { LD Ri #a }        |
| 2)  | $R_i \rightarrow M_x$                        | { LD Ri x }         |
| 3)  | $M \rightarrow = M_x R_i$                    | { ST x Ri }         |
| 4)  | $M \rightarrow = \text{ind } R_i R_j$        | { ST *Ri Rj }       |
| 5)  | $R_i \rightarrow \text{ind} + c_a R_j$       | { LD Ri a(Rj) }     |
| 6)  | $R_i \rightarrow + R_i \text{ind} + c_a R_j$ | { ADD Ri Ri a(Rj) } |
| 7)  | $R_i \rightarrow + R_i R_j$                  | { ADD Ri Ri Rj }    |
| 8)  | $R_i \rightarrow + R_i c_1$                  | { INC Ri }          |
| 9)  | $R \rightarrow \text{sp}$                    |                     |
| 10) | $M \rightarrow \text{m}$                     |                     |

## 13.5 Метод переписывания дерева

### 13.5.7 Поиск соответствий с помощью синтаксического анализа

- ◇ На основе продукций схемы трансляции можно построить *LR*-анализатор: стековый автомат, который загоняет символы анализируемой строки в стек (*перенос*) пока в голове стека не получится строка, являющаяся правой частью одной из продукций. Когда это происходит выполняется *свертка*: правая часть найденной продукции удаляется из стека и вместо нее в стек помещается левая часть этой продукции. При каждой свертке генерируется машинная команда из соответствующей строки третьего столбца таблицы.
- ◇ Обычно грамматика генерации кода неоднозначна и поэтому должны быть предприняты меры по разрешению конфликтов. При отсутствии информации о стоимости общее правило состоит в предпочтении больших сверток меньшим. Это означает, что в случае конфликта «свертка – свертка» выбирается более длинная свертка, а при конфликте «перенос – свертка» – перенос. Такой подход обеспечивает выполнение большего числа операций с помощью одной машинной команды.

## **13.5 Метод переписывания дерева**

### **13.5.7 Поиск соответствий с помощью синтаксического анализа**

- ◇ Преимущества использования *LR*-анализа для генерации кода:
  - ◇ Методы синтаксического анализа эффективны и хорошо изучены, использование алгоритмов *LR*-анализа обеспечивает надежность и эффективность генератора кода.
  - ◇ Облегчается перенастройка генератора кода для другой целевой машины, так как создание генератора кода для новой машины сводится к разработке грамматики, описывающей ее команды.
  - ◇ Качество генерируемого кода может быть сделано весьма высоким за счет добавления продукций для особых случаев, чтобы использовать преимущества машинных идиом.

## **13.5 Метод переписывания дерева**

### **13.5.8 Проверка атрибутов**

- ◇ В схеме трансляции для генерации кода встречаются ограничения на значения атрибутов. Такие ограничения могут возникнуть в связи с машинными идиомами.
- ◇ Машинная команда может требовать, чтобы значения атрибута находились в определенном диапазоне или чтобы два значения атрибутов были связаны некоторым соотношением.  
  
Такие ограничения на значения атрибутов могут быть указаны как предикаты, проверяемые перед выполнением свертки. Использование предикатов может обеспечить большую гибкость и простоту описания по сравнению с чисто грамматической спецификацией генератора кода.
- ◇ Некоторые аспекты архитектуры целевой машины, например режимы адресации, могут быть выражены с помощью ограничений на атрибуты. Это позволяет сократить описание целевой машины.

## 13.5 Метод переписывания дерева

### 13.5.7 Проверка атрибутов

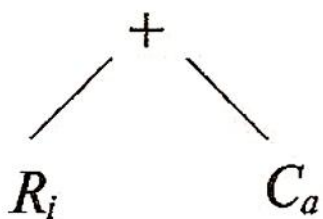
◇ В случае использования *обобщенных шаблонов* для выбора команд в частных случаях могут использоваться семантические проверки

◇ **Пример.**

С помощью одного обобщенного шаблона могут быть представлены два варианта команд сложения: **ADD** и **INC**.

Для выбора команды используется *семантическая проверка*.

$R_i \leftarrow$



Обобщенный шаблон

**if** ( $a = 1$ )

INC  $R_i$

**else**

ADD  $R_i, R_i, \#a$

Семантическая проверка

## 13.6 Генерация оптимального кода для выражений

### 13.6.1 Разметка деревьев выражений. Числа Ершова

- ◇ *Числа Ершова* назначаются каждому узлу дерева выражения и указывают, сколько регистров требуется для вычисления этого узла без сохранения временных переменных
- ◇ Числа Ершова назначаются по следующим правилам:
  1. Все листья имеют метку 1.
  2. Метка внутреннего узла с одним дочерним узлом равна метке дочернего узла.
  3. Метка внутреннего узла с двумя дочерними равна:
    - (а) если метки дочерних узлов различны – наибольшей из меток дочерних узлов;
    - (б) если метки дочерних узлов совпадают – метке дочернего узла, увеличенной на 1.

# 13.6 Генерация оптимального кода для выражений

## 13.6.1 Разметка деревьев выражений. Числа Ершова

◇ **Пример.** Трехадресный код, соответствующий выражению

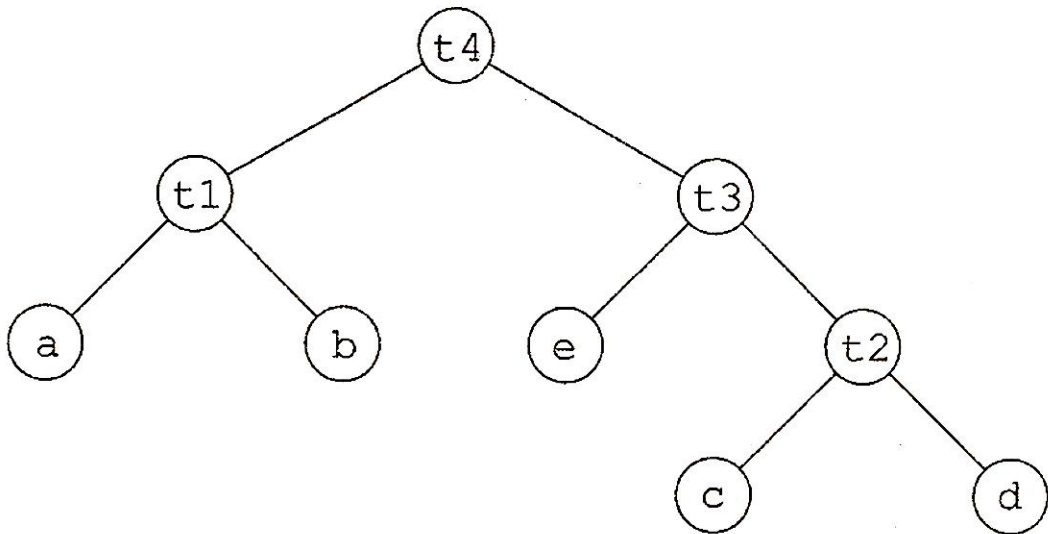
$(a - b) + e \times (c + d):$

**t1 = a - b**

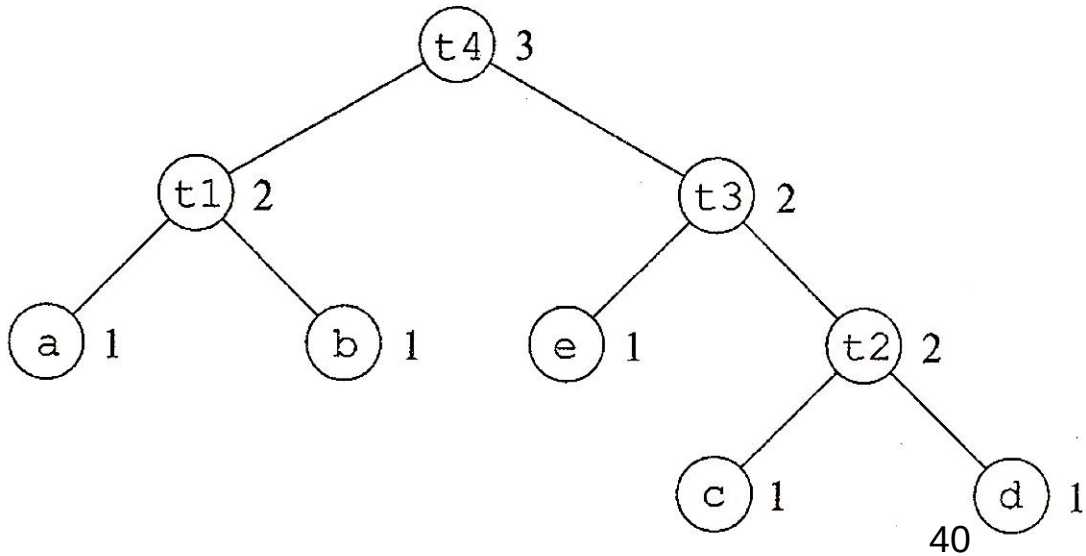
**t2 = c + d**

**t3 = e \* t2**

**t4 = t1 + t3** можно представить в виде дерева на верхнем рисунке



◇ Разметив узлы этого дерева в соответствии со сформулированными правилами, получим дерево на нижнем рисунке





## 13.6 Генерация оптимального кода для выражений

### 13.6.2 Алгоритм генерации кода для размеченных деревьев выражений

- ◇ **Вход:** размеченное дерево, в котором каждый операнд появляется по одному разу (т.е. отсутствуют общие подвыражения).
- ◇ **Выход:** оптимальная последовательность машинных команд для вычисления значения корня в регистре.
- ◇ **Метод:** *рекурсивный алгоритм* генерации машинного кода (на следующем слайде).
- ◇ **Замечание.** Если алгоритм применяется к узлу с меткой  $k$ , будут использованы ровно  $k$  регистров:  $R_b, R_{b+1}, \dots, R_{b+k-1}$ .  
Результат всегда будет помещаться в регистр  $R_{b+k-1}$ .

## 13.6 Генерация оптимального кода для выражений

### 13.6.2 Алгоритм генерации кода для размеченных деревьев выражений

1. Генерация машинного кода для внутреннего узла с меткой  $k$  и двумя дочерними узлами с **равными** метками (в этом случае метки обоих дочерних узлов равны  $k - 1$ ):
  - (a) Рекурсивная генерация кода для правого дочернего узла, используя регистры  $R_{b+1}, \dots, R_{b+k-1}$ . Результат правого дочернего узла помещается в регистр  $R_{b+k-1}$ .
  - (b) Рекурсивная генерация кода для левого дочернего узла, используя регистры  $R_b, \dots, R_{b+k-2}$ . Результат левого дочернего узла помещается в регистр  $R_{b+k-2}$ .
  - (c) Генерация команды
$$\text{ОР } R_{b+k-1}, R_{b+k-2}, R_{b+k-1},$$
где **ОР** – операция в узле с меткой  $k$ .

## 13.6 Генерация оптимального кода для выражений

### 13.6.2 Алгоритм генерации кода для размеченных деревьев выражений

2. Генерация кода для внутреннего узла с меткой  $k$  и двумя дочерними узлами с **разными** метками (в этом случае один из дочерних узлов («большой») имеет метку  $k$ , а второй («маленький») – метку  $m < k$ ):
  - (a) Рекурсивная генерация кода для **большого** узла, используя  $k$  регистров  $R_b, \dots, R_{b+k-1}$ ;  
результат получается в регистре  $R_{b+k-1}$ .
  - (b) Рекурсивная генерация кода для **маленького** узла, используя  $m$  регистров  $R_b, \dots, R_{b+m-1}$ ;  
результат находится в регистре  $R_{b+m-1}$ .  
 $m < k$ , следовательно при вычислениях не используются ни регистр  $R_{b+m}$ , ни какой-либо иной регистр с большим номером.
  - (c) Генерируем команду  
**ОР**  $R_{b+k-1}, R_{b+m-1}, R_{b+k-1}$  или **ОР**  $R_{b+k-1}, R_{b+k-1}, R_{b+m-1}$   
в зависимости от того, является большой узел правым или левым.

## ***13.6 Генерация оптимального кода для выражений***

### **13.6.2 Алгоритм генерации кода для размеченных деревьев выражений**

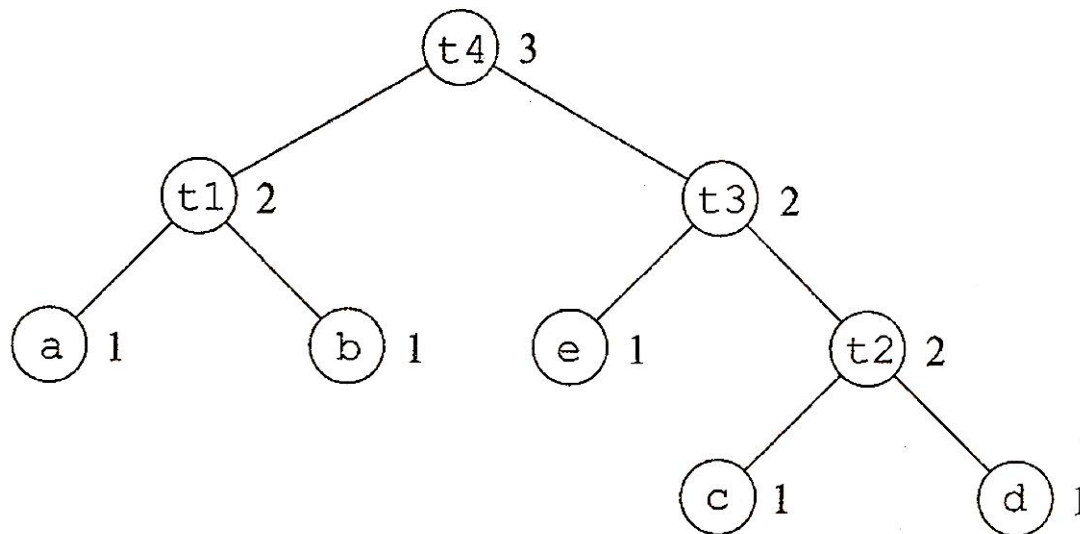
3. Для листа, представляющего операнд  $x$ , при использовании базы  $b$  генерируем команду **LD**  $R_b, x$ .

## 13.6 Генерация оптимального кода для выражений

### 13.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



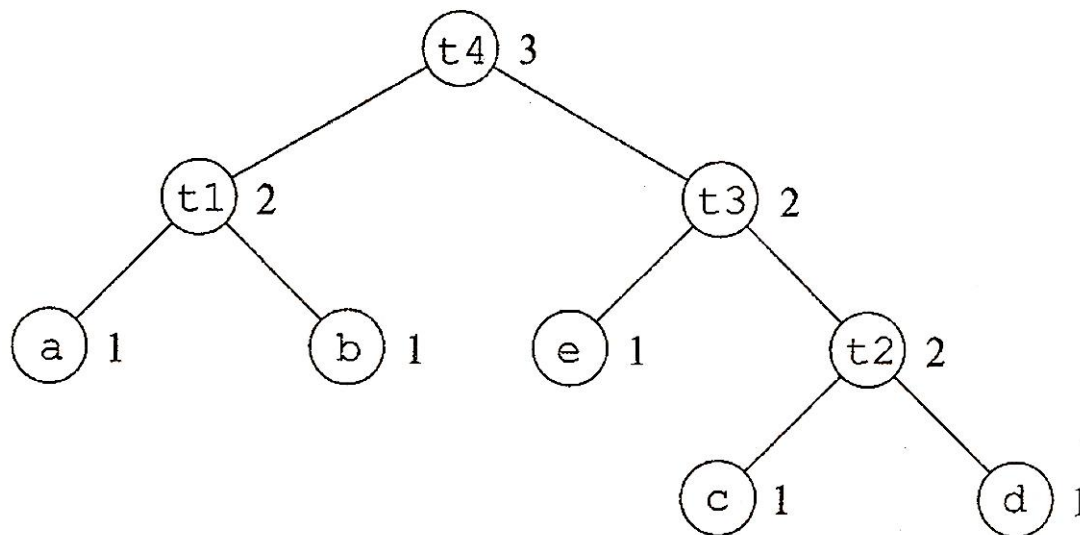
Применим описанный алгоритм к дереву на рисунке. Поскольку метка корня равна 3, результат получится в регистре R3, а использоваться при вычислениях будут только регистры R1, R2 и R3.



- 1) Рассматривается корень **t4**. У него два дочерних узла **t1** и **t3** с одинаковыми метками, следовательно должен работать п.1 алгоритма.

## 13.6 Генерация оптимального кода для выражений

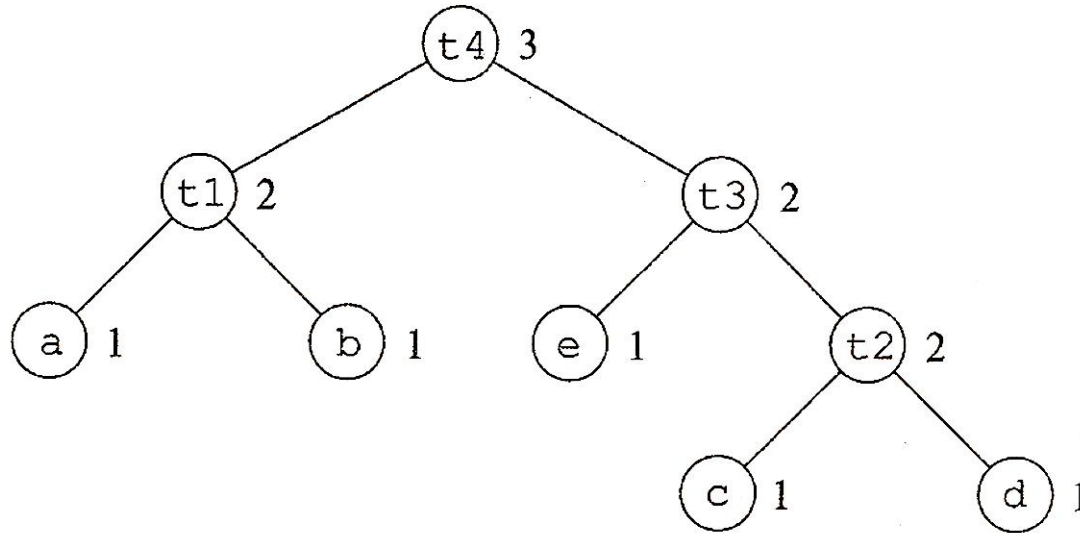
### 13.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



- 2) Согласно п.1 (а) сначала нужно сгенерировать код для правого поддерева (с корнем **t3**).  
У него правый узел большой, а левый – маленький, следовательно должен работать п.2 алгоритма.
- 3) Согласно п.2 (а) сначала генерируется код для большого поддерева (с корнем **t2**).

## 13.6 Генерация оптимального кода для выражений

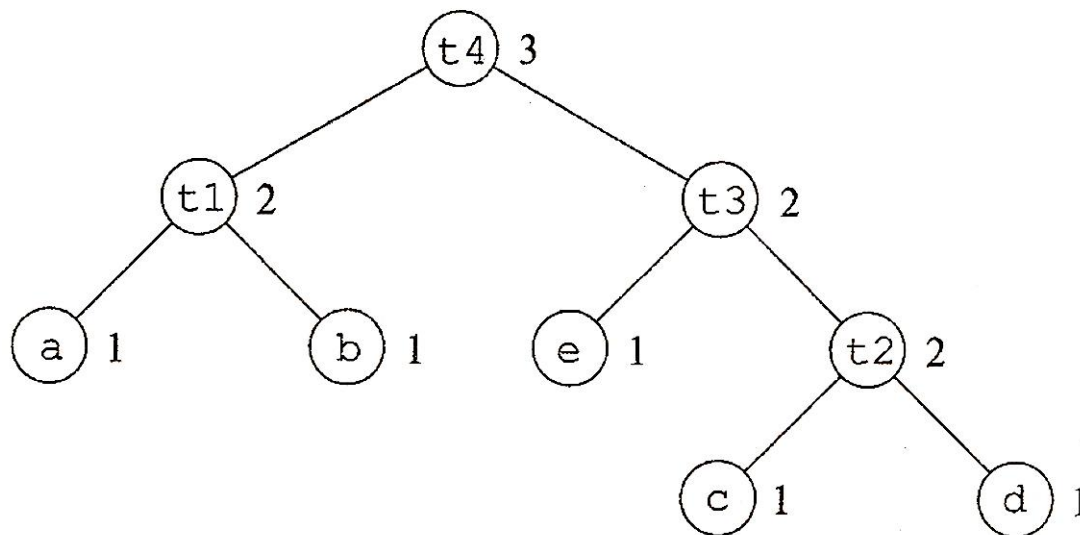
### 13.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



- 4) Рассматривается корень **t2**. У него два дочерних листовых узла с одинаковыми метками, следовательно должен работать п.1 алгоритма.

## 13.6 Генерация оптимального кода для выражений

### 13.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



- 5) Сначала согласно п.1 (а) нужно сгенерировать код для **правого** листа, потом согласно п.1 (b) нужно сгенерировать код для **левого** листа, потом согласно п.1 (с) нужно сгенерировать **команду**. В результате получатся следующие три команды:

**LD     R3, d**

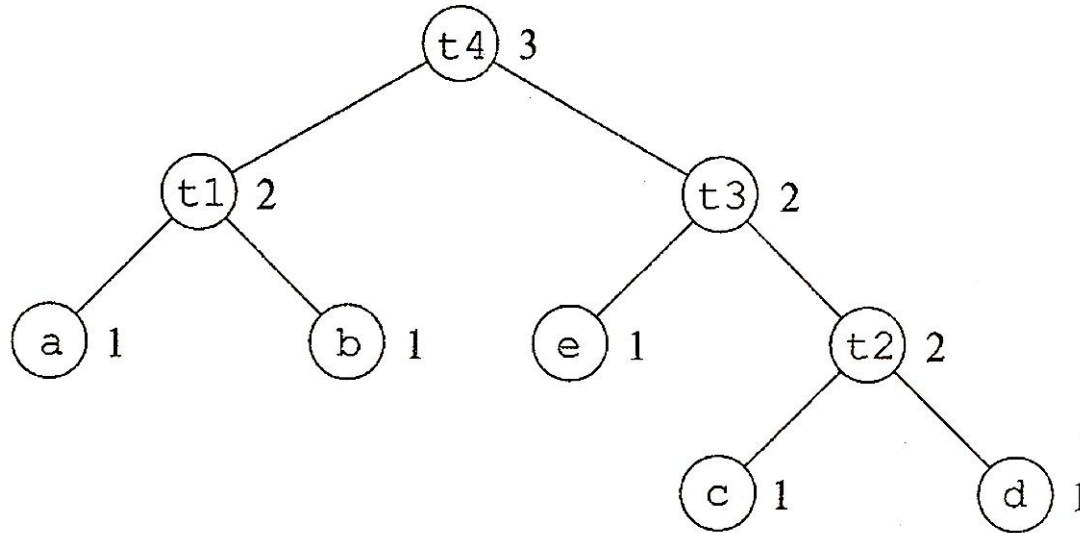
**LD     R2, c**

**ADD   R3, R2, R3**



## 13.6 Генерация оптимального кода для выражений

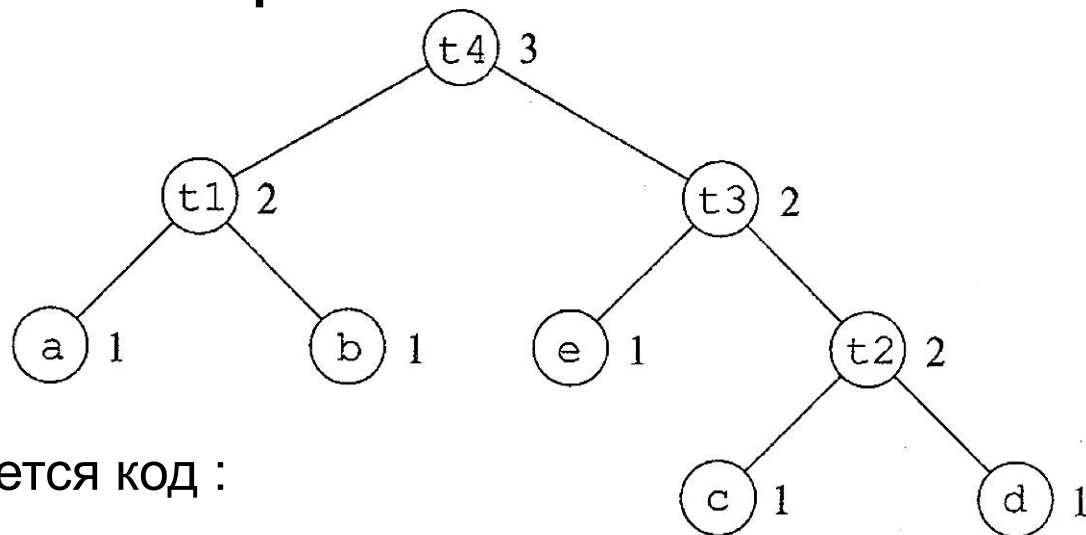
### 13.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



- 6) Далее генерируются команды для:  
левого поддеревья дерева с корнем **t3**,  
для корня **t3**,  
для левого поддеревья дерева с корнем **t4**  
и для всего дерева с корнем **t4**

## 13.6 Генерация оптимального кода для выражений

### 13.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



В результате генерируется код :

```
LD R3, d
LD R2, e
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

## **13.6 Генерация оптимального кода для выражений**

### **13.6.4 Вычисление выражений при недостаточном количестве регистров**



**Алгоритм генерации кода для размеченного дерева выражения (с учетом конечного числа доступных регистров)**

**Вход:** (1) размеченное дерево, в котором каждый операнд встречается по одному разу (т.е. отсутствуют общие подвыражения)  
(2) количество доступных регистров  $r \geq 2$

**Выход:** оптимальная последовательность машинных команд для вычисления значения корня дерева на регистре с использованием не более, чем  $r$  регистров  
(регистры  $R_1, R_2, \dots, R_r$ )

**Метод:** рекурсивный алгоритм генерации машинного кода

## **13.6 Генерация оптимального кода для выражений**

### **13.6.4 Вычисление выражений при недостаточном количестве регистров**



**Алгоритм генерации кода для размеченного дерева выражения (с учетом конечного числа доступных регистров)**

**Метод:** рекурсивный алгоритм генерации машинного кода

Пусть  $k$  – метка корня рассматриваемого дерева  $N$ .

Если  $k \leq r$ , алгоритм совпадает с алгоритмом 13.6.2

Если  $k > r$ , каждое поддерево обрабатывается отдельно, и результат большого поддерева (оно обрабатывается первым) сохраняется в памяти. Непосредственно перед вычислением корня  $N$  этот результат загружается из памяти на регистр  $R_r$  и используется вместе с результатом маленького поддерева (полученного на регистре  $R_{r-1}$ ) для вычисления результата корня  $N$  (на регистре  $R_r$ ).

## 13.6 Генерация оптимального кода для выражений

### 13.6.4 Вычисление выражений при недостаточном количестве регистров



Метод

- (1) **Выбор большого узла:** Узел  $N$  имеет как минимум один дочерний узел с меткой  $\geq r$ . Дочерний узел с большей меткой выбирается как «большой», а узел с меньшей меткой – как «маленький». Если метки обоих узлов одинаковы, то в качестве «большого» берется правый дочерний узел.
- (2) Рекурсивная генерация кода для большого дочернего узла с использованием регистров  $R_1, R_2, \dots, R_r$ . Результат вычислений получается на регистре  $R_r$ .
- (3) Генерация команды **ST**  $t_k$   $R_r$ , где  $t_k$  – временная переменная, используемая для вычисления узлов с меткой  $k$ .

## 13.6 Генерация оптимального кода для выражений

### 13.6.4 Вычисление выражений при недостаточном количестве регистров



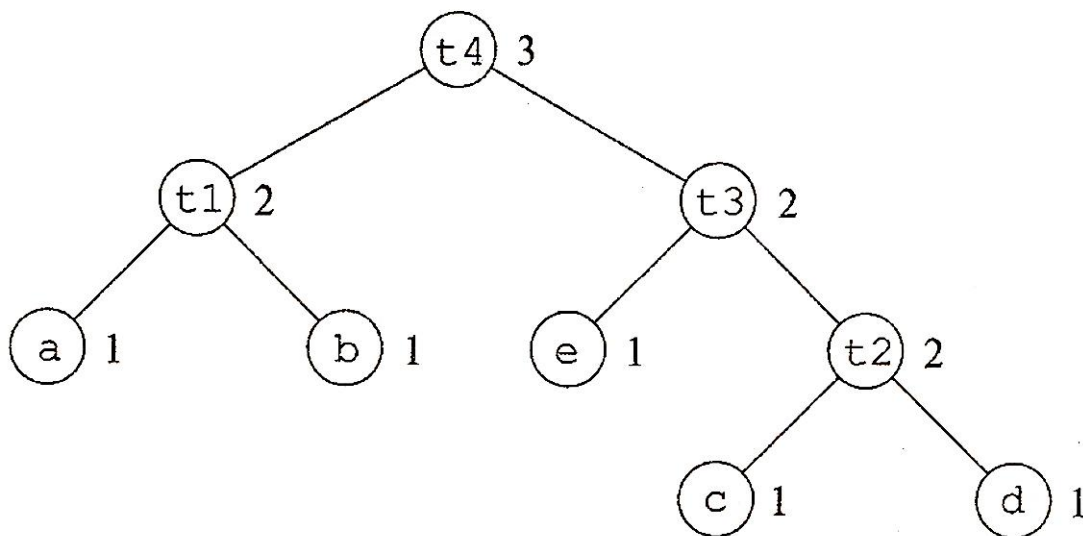
Метод

- (4) Рекурсивная генерация кода для маленького дочернего узла с использованием регистров  $R_{r-j}, R_{r-j+1}, \dots, R_r$ , где  $j \leq r$  – метка маленького узла. Результат вычислений получается на регистре  $R_r$ .
- (5) Генерация команды **LD**  $R_{r-1} \ t_k$ .
- (6) Генерация команды **OP**  $R_r \ R_r \ R_{r-1}$ , если большой узел является правым дочерним узлом корня  $N$ , либо команды **OP**  $R_r \ R_{r-1} \ R_r$ , если большой узел является левым дочерним узлом корня  $N$ .

## 13.6 Генерация оптимального кода для выражений

### 13.6.5 Пример

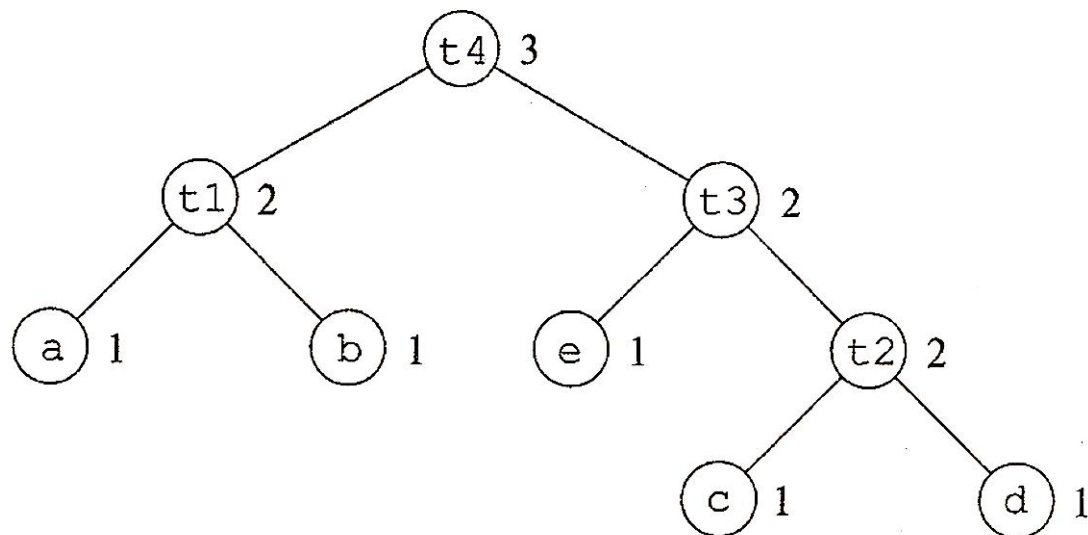
- ◇ Применим алгоритм 13.6.4 к дереву на рисунке, в предположении, что количество доступных регистров равно 2 ( $r = 2$ ), т.е. для хранения временных значений при вычислении выражения можно использовать только два регистра – **R1** и **R2**.



- ◇ Корень дерева имеет метку 3, большую, чем  $r = 2$ . Согласно п. (1) алгоритма 13.6.4 выбирается большой узел. Так как у обоих дочерних узлов метки одинаковы, выбирается правый узел.

## 13.6 Генерация оптимального кода для выражений

### 13.6.5 Пример



Рекурсивная генерация кода для поддерева с корнем **t3** выполняется алгоритмом 13.6.2, так как метка **t3** равна 2, т.е. регистров для его вычисления достаточно. Результат похож на результат примера 13.6.3, но вместо регистров **R2** и **R3** используются регистры **R1** и **R2** :

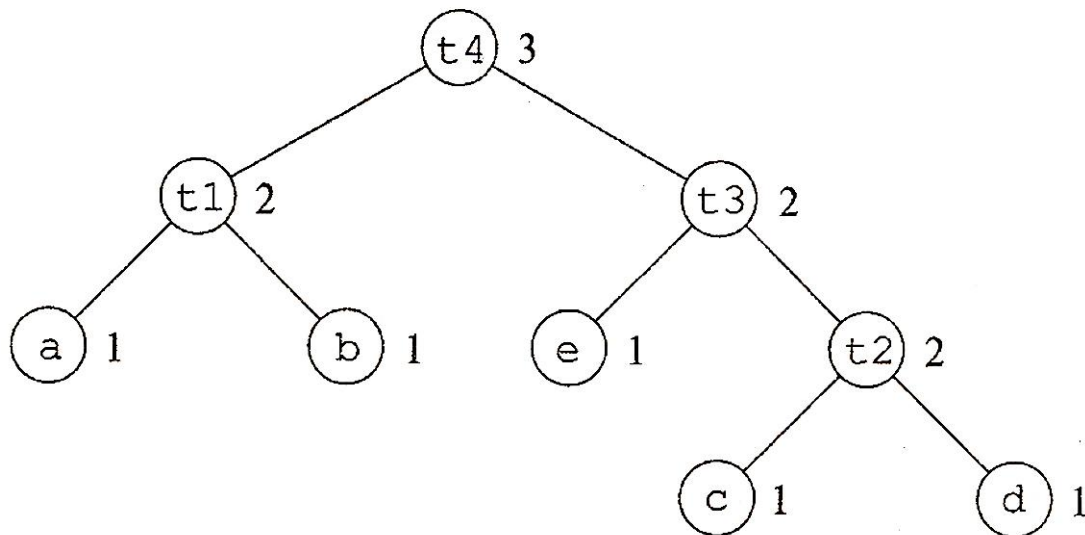
```
LD R2 d
LD R1 c
ADD R2 R1 R2
LD R1 e
MUL R2 R1 R2
```



## 13.6 Генерация оптимального кода для выражений

### 13.6.5 Пример

◇ Поскольку для вычисления левого дочернего узла корня нужны оба регистра, генерируется команда  
**ST t3 R2**



◇ Для левого дочернего узла регистров хватает для вычислений, так что алгоритмом 13.6.2 генерируется код

**LD R2 b**

**LD R1 a**

**SUB R2 R1 R2**

◇ На регистре R1 восстанавливается значение правого поддерева командой **LD R1 t3**

◇ Выполняется операция в корне дерева командой

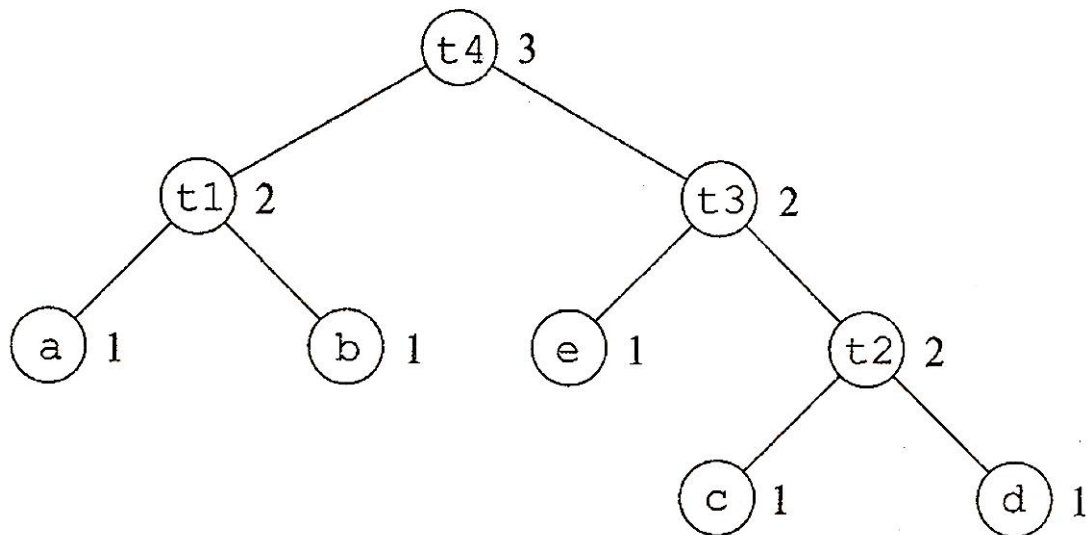
**ADD R2 R2 R1**

## 13.6 Генерация оптимального кода для выражений

### 13.6.5 Пример

В результате для дерева на рисунке генерируется код:

```
LD R2 d
LD R1 c
ADD R2 R1 R2
LD R1 e
MUL R2 R1 R2
ST t3 R2
LD R2 b
LD R1 a
SUB R2 R1 R2
LD R1 t3
ADD R2 R2 R1
```



Который вычисляет дерево,  
используя только два регистра

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.1 Постановка задачи

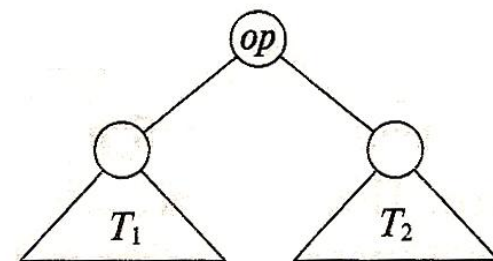
- ◇ Алгоритм 13.6.4 генерирует оптимальный код по дереву выражения за время, линейно зависящее от размера дерева. Этот алгоритм работает для машин, у которых все вычисления выполняются в регистрах, а команды состоят из операторов, применяемых к двум регистрам или к регистру и ячейке памяти.
- ◇ Для расширения класса машин, для которых возможно построение оптимального кода на основе деревьев выражений за линейно зависящее от размера дерева время, можно воспользоваться алгоритмом на основе динамического программирования.
- ◇ Алгоритм динамического программирования может использоваться для генерации кода для любой машины с  $r$  взаимозаменяемыми регистрами  $R_0, R_1, \dots, R_{(r-1)}$  и командами загрузки, сохранения и операций.

Для простоты сделано предположение, что все команды имеют одинаковую стоимость, равную единице, хотя алгоритм динамического программирования можно легко модифицировать для случая, когда<sup>59</sup> стоимости команд различны.

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.1 Постановка задачи

- ◇ Алгоритм динамического программирования – это эвристика, которая состоит в разбиении сложной задачи на подзадачи аналогичной структуры и поиске оптимального решения для каждой подзадачи: предполагается, что каждая подзадача проще основной задачи, и ее оптимальное решение найти проще.
- ◇ Генерация оптимального кода для выражения сводится к генерации оптимального кода для подвыражений этого выражения и последующей генерации кода для выражения в целом.
- ◇ Рассмотрим выражение  $E$  вида  $E_1 \text{ op } E_2$ .  
Алгоритм динамического программирования составляет оптимальную программу для  $E$ , выбирая оптимальный порядок выполнения программ для вычисления  $E_1$  и  $E_2$ , и помещая вслед за ними код для выполнения операции  $\text{op}$ .  
Подзадачи генерации оптимального кода для вычисления подвыражений  $E_1$  и  $E_2$  решаются аналогично.



## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.2 Последовательные вычисления

- ◇ По определению программа  $P$  вычисляет дерево  $T$  *последовательно*, если она вначале вычисляет поддеревья  $T_1$  и  $T_2$  дерева  $T$  ( $T$  либо в порядке  $T_1, T_2$  и затем корень, либо в порядке  $T_2, T_1$ ) и **запоминает их значения в памяти**. Затем  $P$  вычисляет корень.  
При необходимости **используются предварительно вычисленные значения, хранящиеся в памяти**.
- ◇ **Утверждение.** В случае регистровой машины для любой программы  $P$ , вычисляющей дерево выражения  $T$ , можно найти эквивалентную программу  $P'$ , такую, что
  - ◇ стоимость  $P'$  не больше стоимости  $P$ ;
  - ◇  $P'$  использует не больше регистров, чем  $P$ ;
  - ◇  $P'$  вычисляет дерево последовательно.
- ◇ **Следствие.** Каждое дерево выражения можно вычислить оптимальным образом с помощью последовательной программы.

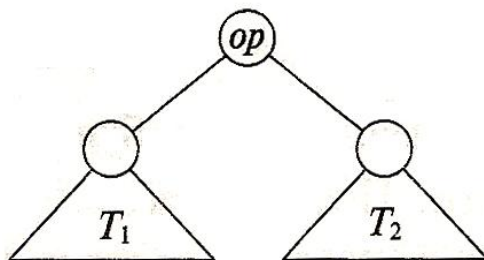
## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.2 Последовательные вычисления



Оптимальная программа, порожденная алгоритмом динамического программирования, имеет важное свойство, заключающееся в том, что она вычисляет выражение  $E = E_1 \text{ op } E_2$  «последовательно».

Синтаксическое дерево  $T$  для выражения  $E$ .



$T_1$  и  $T_2$  – синтаксические деревья для  $E_1$  и  $E_2$  соответственно.

## **13.7 Генерация кода с использованием алгоритма динамического программирования**

### **13.7.3 Алгоритм динамического программирования**



Алгоритм динамического программирования состоит из трех фаз.

- (1) В восходящем порядке для каждого узла  $n$  дерева выражения  $T$  вычисляется массив стоимости  $C$ ,  $i$ -й элемент которого  $C[i]$  представляет собой оптимальную стоимость вычисления поддеревья  $S$  с корнем  $n$  с при условии, что для вычисления имеется  $i$  ( $1 \leq i \leq r$ ) доступных регистров ( $r$  – число регистров целевой машины)
- (2) Обход дерева выражения  $T$  с использованием векторов стоимости для определения, какие из поддеревьев  $T$  должны быть вычислены в память.
- (3) Обход каждого дерева с использованием векторов стоимости и связанных с ними команд для генерации конечного целевого кода. Первым генерируется код для поддеревьев, вычисляемых в память.



Каждая из фаз может быть выполнена за время, линейно пропорциональное размеру дерева выражения.

## **13.7 Генерация кода с использованием алгоритма динамического программирования**

### **13.7.3 Алгоритм динамического программирования**



Стоимость вычисления узла  $n$  включает загрузки и сохранения, необходимые для вычисления  $S$  с данным количеством регистров и стоимость выполнения операции в корне  $S$ . Нулевой компонент вектора стоимости – оптимальная стоимость вычисления поддерева  $S$  с сохранением результата в памяти.

Свойство последовательного вычисления гарантирует, что оптимальная программа для вычисления поддерева  $S$  может быть сгенерирована путем рассмотрения комбинаций только оптимальных программ для поддеревьев дерева с корнем  $S$ . Сформулированное ограничение сокращает количество случаев, которые необходимо рассмотреть.



## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.3 Алгоритм динамического программирования



Для вычисления стоимости  $C[i]$  в узле  $n$  будем рассматривать команды как правила преобразования дерева.

Для каждого шаблона  $E$ , соответствующего входному дереву в узле  $n$ :



Изучая векторы стоимости соответствующих наследников  $n$ , определить стоимости вычисления операндов в листьях  $E$ .



Для операндов  $E$ , являющихся регистрами, рассмотреть все возможные порядки вычисления поддеревьев  $T$  в регистры.

Для каждого из рассматриваемых порядков вычисления первое поддерево, соответствующее регистровому операнду, можно вычислить с использованием  $i$  доступных регистров, второе – с помощью  $i - 1$  доступных регистров и т.д.



Добавить стоимость команды, связанной с корнем  $E$ .

Значение  $C[i]$  представляет собой минимальную стоимость среди всех возможных порядков вычисления.

## **13.7 Генерация кода с использованием алгоритма динамического программирования**

### **13.7.3 Алгоритм динамического программирования**



Векторы стоимости для всего дерева  $T$  могут быть вычислены в восходящем порядке (снизу вверх) за линейное время. Команды, соответствующие наилучшей стоимости  $C[i]$  для каждого значения  $i$ , удобно хранить в узлах дерева; при этом наименьшая стоимость корневого узла  $T$  соответствует минимальной стоимости вычисления дерева  $T$ .

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.4 Пример



Пусть машина имеет два регистра  $R_0$  и  $R_1$  и следующие команды, стоимость каждой из которых равна единице:

| <i>Команда</i>            | <i>Семантика команды</i>    |
|---------------------------|-----------------------------|
| <b>LD</b> $R_i, M_j$      | $R_i = M_j$                 |
| <b>op</b> $R_i, R_i, R_j$ | $R_i = R_i \text{ op } R_j$ |
| <b>op</b> $R_i, R_i, M_j$ | $R_i = R_i \text{ op } M_j$ |
| <b>LD</b> $R_i, R_j$      | $R_i = R_j$                 |
| <b>ST</b> $M_i, R_j$      | $M_i = R_j$                 |

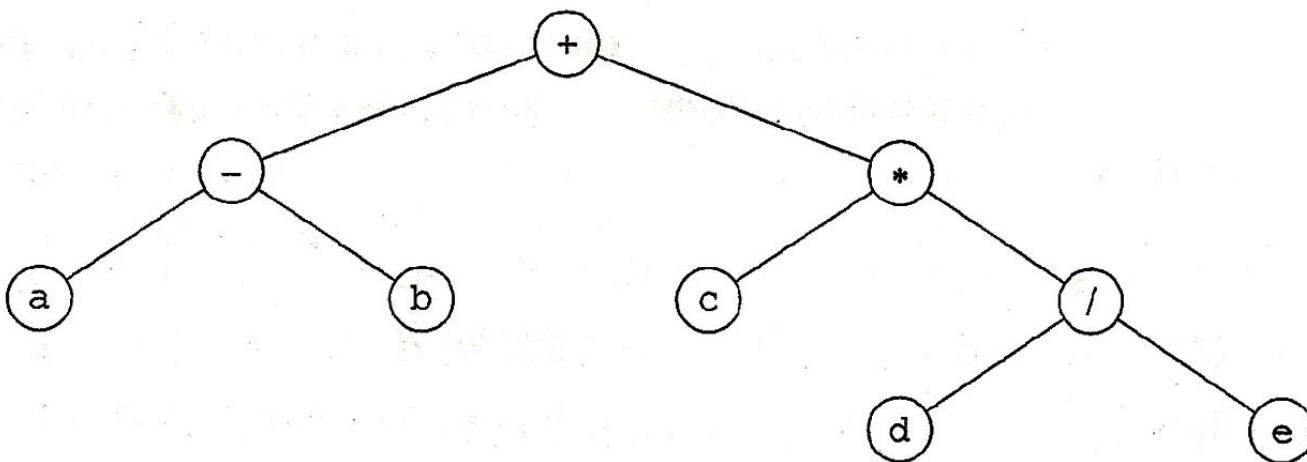


$R_i$  представляет собой либо  $R_0$ , либо  $R_1$ , а  $M_j$  – адрес в памяти. Значок *op* соответствует знаку арифметической операции.

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.4 Пример

- ◇ Применим алгоритм динамического программирования для генерации оптимального кода для синтаксического дерева (см. рисунок).

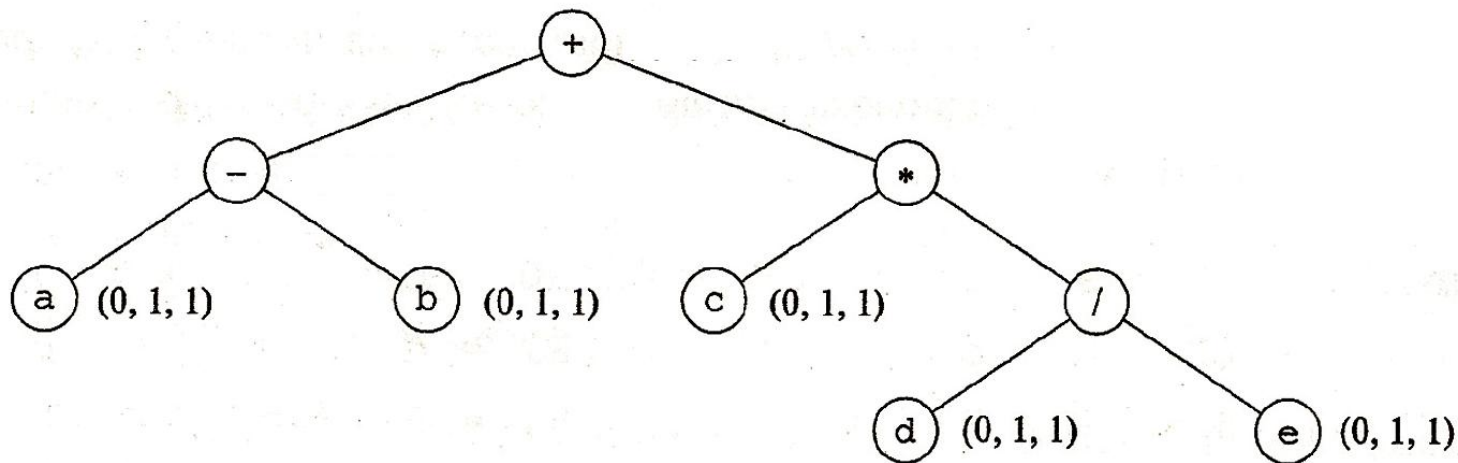


#### (1) Вычисление векторов стоимости.

- ◇ Вычисление вектора стоимости для листа **a**:
- ◆ Стоимость вычисления **a** в память, равна нулю, так как **a** и так находится в памяти:  $C[0] = 0$ .
  - ◆ Стоимость  $C[1]$  вычисления **a** на регистр равна 1, так как **a** можно загрузить на регистр одной командой `LD R0, a`.
  - ◆ Стоимость  $C[2]$  загрузки **a** на регистр при двух доступных регистрах тоже равна 1.
- Таким образом, вектор стоимости в листе **a** равен (0, 1, 1).

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.4 Пример

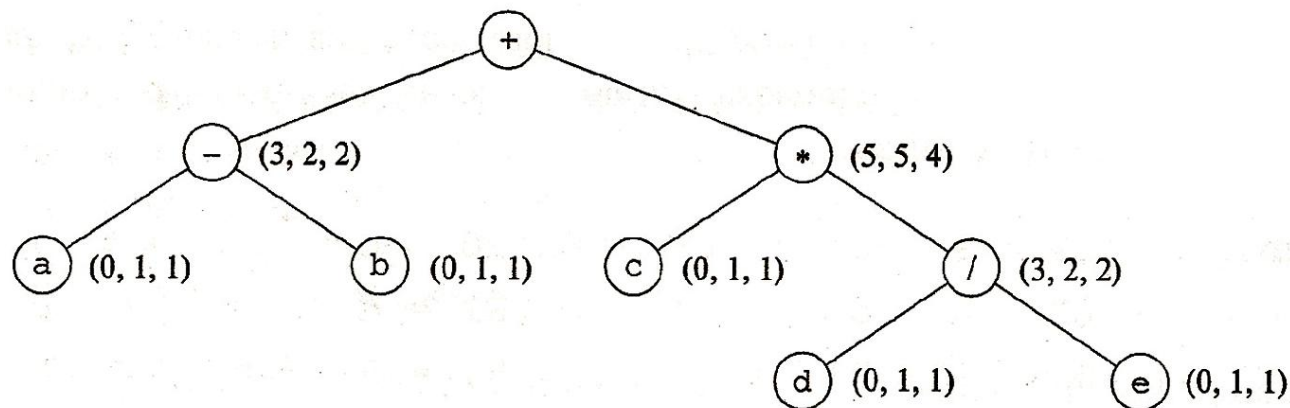


(1) Вычисление векторов стоимости.

- ◆ Векторы стоимости для листьев  $b$   $c$   $d$   $e$  вычисляются аналогично.
- ◆ Вычисление вектора стоимости для узла «-» :
  - ◆ Оба операнда в памяти  $C[0] = 3$  так как оба операнда нужно загрузить на регистры
  - ◆ Один из операндов на регистре  $C[1] = C[2] = 2$
- ◆ Вектор стоимости для для узла «/» вычисляется аналогично.
- ◆ Вектор стоимости для для узла «\*» вычисляется аналогично.

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.4 Пример

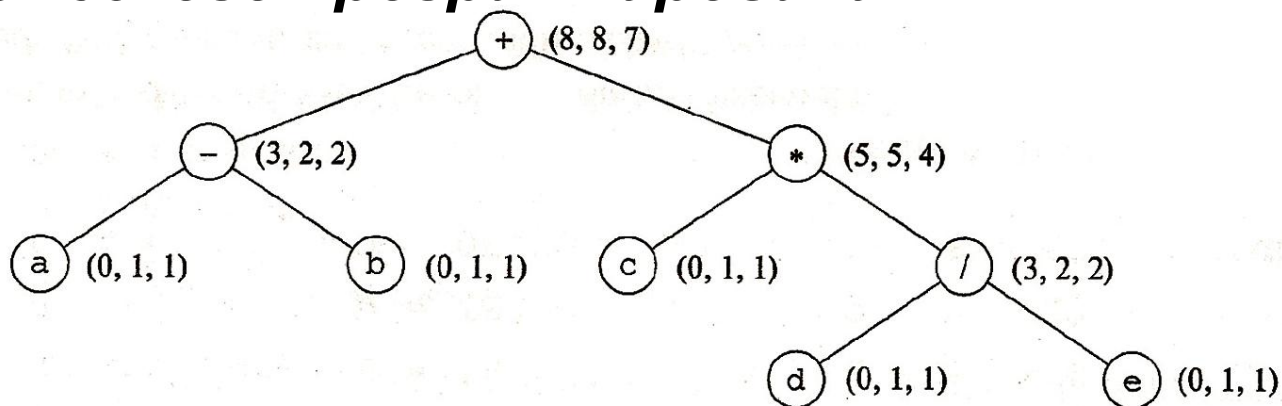


#### (1) Вычисление векторов стоимости.

- ◆ Вычисление вектора стоимости для корня дерева. Различные варианты вычисления корня
  - ◆ (1) (2 свободных регистра) левое поддереве  $\Rightarrow R0$  (стоимость 2)  
(2) (1 свободный регистр) правое поддереве  $\Rightarrow R1$  (стоимость 5),  
(3) команда **ADD**  $R0, R0, R1$  (стоимость 1). **Итого  $2 + 5 + 1 = 8$**
  - ◆ (1) правое поддереве  $\Rightarrow M$  (стоимость 5)  
(2) (2 свободных регистра) левое поддереве  $\Rightarrow R0$  (стоимость 2),  
(3) команда **ADD**  $R0, R0, M$  (стоимость 1). **Итого  $5 + 2 + 1 = 8$**
  - ◆ (1) (2 свободных регистра) правое поддереве  $\Rightarrow R1$  (стоимость 4)  
(2) (1 свободный регистр) левое поддереве  $\Rightarrow R0$  (стоимость 2),  
(3) команда **ADD**  $R0, R0, R1$  (стоимость 1). **Итого  $4 + 2 + 1 = 7$**

## 13.7 Генерация кода с использованием алгоритма динамического программирования

### 13.7.4 Пример



(2)

Обход дерева и генерация кода

Имея векторы стоимости можно построить код путем обхода дерева. Для рассматриваемого дерева в предположении доступности двух регистров оптимальный код имеет следующий вид:

|     |     |     |          |                |
|-----|-----|-----|----------|----------------|
| LD  | R0, | c   | //R0 = c |                |
| LD  | R1, | d   | //R1 = d |                |
| DIV | R1, | R1, | e        | //R1 = R1 / e  |
| MUL | R0, | R0, | R1       | //R0 = R0 * R1 |
| LD  | R1, | a   | //R1 = a |                |
| SUB | R1, | R1, | b        | //R1 = R1 - b  |
| ADD | R1, | R1, | R0       | //R1 = R1 + R0 |

## **14. Распределение и назначение регистров**



## 14.1 Постановка задачи

### 14.1.1 Модель памяти

- ◇ В промежуточном представлении с каждой переменной связывается ячейка памяти для хранения ее значений.
- ◇ Эта ячейка называется *абстрактной*, так как с ней связывается *символический адрес*, который может указывать на регистры, стек, кучу, статическую память, причем каждому классу памяти соответствует бесконечно много символических адресов
- ◇ В компиляторах часто используется модель памяти *«регистр – регистр»*.  
В этой модели компилятор старается расположить все данные на *символических регистрах* (или *псевдорегистрах*).  
В отличие от физических регистров, число которых невелико, псевдорегистров бесконечно много.  
При распределении памяти часть псевдорегистров придется отобразить не на регистры, а в память.

## 14.1 Постановка задачи

### 14.1.2 Распределение и назначение регистров

- ◇ *Распределение регистров*  
отображает неограниченное множество имен (псевдорегистров) на конечное множество физических регистров целевой машины, Это NP-полная задача
- ◇ *Назначение регистров*  
отображает множество распределенных имен регистров на физические регистры целевого процессора. Для решения этой задачи известно несколько алгоритмов *полиномиальной сложности*.
- ◇ Во время назначения регистров предполагается, что распределение регистров уже было выполнено, так что при генерации каждой команды требуется не более  $n$  регистров ( $n$  – число физических регистров).

## 14.2 Локальное распределение регистров

### 14.2.1 Постановка задачи



Применения регистров:



На регистры помещаются **операнды и результаты операций**

(при выполнении операции необходимо, чтобы ее операнды находились на регистрах, результат получается на регистре).



Регистры – **временные переменные**

(на регистры помещаются промежуточные результаты при вычислении выражений

если удастся, на них размещаются все переменные, использующиеся в пределах только одного базового блока).



Регистры используются **для хранения глобальных значений**.



Регистры используются для помощи в управлении памятью времени выполнения (например для управления стеком времени выполнения, включая поддержку указателя стека).

## 14.2 Локальное распределение регистров

### 14.2.1 Постановка задачи

- ◇ Рассмотрим алгоритм, распределяющий только те регистры, которые предназначены для операндов и временных переменных (остальные регистры зарезервированы).
- ◇ Предположения:
  - ◇ Базовый блок уже оптимизирован (все «лишние» вычисления удалены).
  - ◇ Для каждой операции **OP** существует команда вида  
**OP reg, reg, reg**  
(операнды и результат – на регистрах)
  - ◇ В набор команд входят команды:  
**LD reg, mem** (загрузка из памяти на регистр)  
**ST mem, reg** (сохранение значения регистра)
- ◇ Необходимо, чтобы генератор кода минимизировал количество операций **LD** и **ST** в целевом коде

## 14.2 Локальное распределение регистров

### 14.2.2 Дескрипторы регистров и переменных

- ◇ *Дескриптор  $DR[r]$  регистра  $r$  указывает, значение какой переменной содержится на регистре  $r$  (на каждом регистре могут храниться значения одного или нескольких имен)*
- ◇ *Дескриптор  $DA[a]$  переменной  $a$  указывает адрес текущего значения  $a$ . Это может быть регистр, адрес памяти, указатель стека*
- ◇ Пусть определена *функция  $getReg(I)$* , имеющая доступ ко всем дескрипторам регистров и адресов, а также ко всем атрибутам объектов, хранящимся в таблице символов, которая назначает регистры для операндов и результата команды  $I$ .
- ◇ Функция  *$getReg(I)$*  позволяет назначать регистры во время выбора команд

## 14.2 Локальное распределение регистров

### 14.2.3 Выбор команд для базового блока



Выбор команд для вычислительной трехадресной инструкции  
 $x \leftarrow op, y, z$

1. С помощью функции *getReg()* выбираются регистры  $R_x$ ,  $R_y$  и  $R_z$  для  $x$ ,  $y$  и  $z$ .
2. Если  $DR[R_y] \neq y$  ( $y$  не находится на  $R_y$ ), генерируется команда **LD  $R_y, y'$** ,  
где  $y' = DA[y]$  (местоположение  $y$  в памяти).
3. Если  $DR[R_z] \neq z$ , а  $DA[z] = z'$ , генерируется команда **LD  $R_z, z'$** .
4. Генерируется команда **OP  $R_x, R_y, R_z$** .

## 14.2 Локальное распределение регистров

### 14.2.3 Выбор команд для базового блока

- ◇ Выбор команды для инструкции копирования  $x = y$   
Функция *getReg()* всегда выбирает для  $x$  и  $y$  одни и те же регистры.  
Если  $DR[R_y] \neq y$ , генерируется команда **LD  $R_y, y'$** .  
Если  $DR[R_y] = y$ , ничего не генерируется  
Во всех случаях обновляется  $DR[R_y]$ :  $x$  становится одним из значений, находящихся на  $R_y$ .
- ◇ Генерация команды запоминания значений переменных, остающихся живыми после выхода из блока.  
Если переменная  $x$  жива на выходе из блока,  
и если в конце блока оказывается, что  $DA[x] = R$  (а не  $x$ ),  
требуется генерация команды **ST  $x, R$** .

## 14.2 Локальное распределение регистров

### 14.2.3 Выбор команд для базового блока

- ◇ Правила обновления  $DR$  и  $DA$  после генерации команды
  - ◇ Для команды **LD  $R$ ,  $x$** :
    - ◆ изменяем  $DR[R]$ : в  $R$  хранится только  $x$ ;
    - ◆ изменяем  $DA[x]$ , добавляя ссылку на  $R$
  - ◇ Для команды **ST  $x$ ,  $R$** 
    - ◆ изменяем  $DA[x]$ , добавляя ссылку на  $x$
  - ◇ Для команды: **ADD  $R_x$ ,  $R_y$ ,  $R_z$** :
    - ◆ изменяем  $DR[R_x]$ : в  $R_x$  хранится  $x$ ;
    - ◆ изменяем  $DA[x]$ :  $x$  – только на  $R_x$
    - ◆ удаляем  $R_x$  из  $DA$  всех переменных, кроме  $x$ .
  - ◇ Для команды  $x = y$ 
    - ◆ если  $DA[y] \neq R_y$  добавляем команду  $LD R_y, y$ ;
    - ◆ изменяем  $DA[x]$  так, чтобы он указывал только на  $R_y$



## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

1  $t \leftarrow -, a, b$

2  $u \leftarrow -, a, c$

3  $v \leftarrow +, t, u$

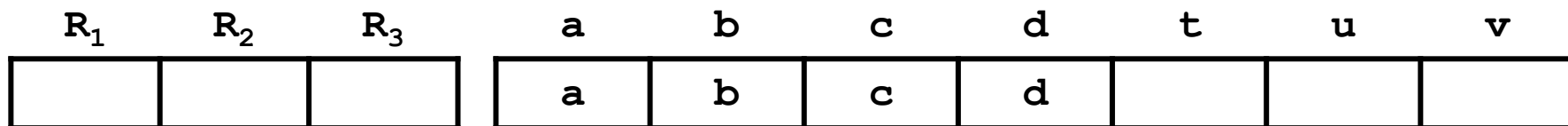
4  $a \leftarrow d$

5  $d \leftarrow +, v, u$

$t, u, v$  – временные переменные,  
локальные для блока,  
 $a, b, c, d$  – переменные, живые  
на выходе из блока.

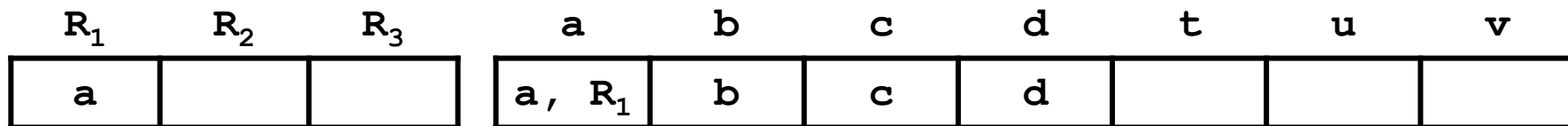
Для инструкции 1:  $t \leftarrow -, a, b$  необходимо сгенерировать три команды:

- ♦ загрузка регистра  $R_a$
- ♦ загрузка регистра  $R_b$
- ♦ вычитание (результат на регистре  $R_t$ )



LD  $R_1, a$

$getReg()$  выдает  $R_1$  для  $R_a$



## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

1  $t \leftarrow -, a, b$

2  $u \leftarrow -, a, c$

3  $v \leftarrow +, t, u$

4  $a \leftarrow d$

5  $d \leftarrow +, v, u$

$t, u, v$  – временные переменные,  
локальные для блока,  
 $a, b, c, d$  – переменные, живые при  
выходе из блока.

Для инструкции 1:  $t \leftarrow -, a, b$  необходимо сгенерировать три команды:

- ♦ загрузка регистра  $R_a$
- ♦ загрузка регистра  $R_b$
- ♦ вычитание (результат на регистре  $R_t$ )

| $R_1$ | $R_2$ | $R_3$ | $a$      | $b$      | $c$ | $d$ | $t$ | $u$ | $v$ |
|-------|-------|-------|----------|----------|-----|-----|-----|-----|-----|
| a     | b     |       | a, $R_1$ | b, $R_2$ | c   | d   |     |     |     |

LD  $R_1, a$

LD  $R_2, b$

SUB  $R_2, R_1, R_2$

*getReg()* выдает  $R_2$  сначала для  $R_b$ , а потом и для  $R_t$

| $R_1$ | $R_2$ | $R_3$ | $a$      | $b$ | $c$ | $d$ | $t$   | $u$ | $v$ |
|-------|-------|-------|----------|-----|-----|-----|-------|-----|-----|
| a     | t     |       | a, $R_1$ | b   | c   | d   | $R_2$ |     |     |

## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

```
1 t ← -, a, b
2 u ← -, a, c
3 v ← +, t, u
4 a ← d
5 d ← +, v, u
```

t, u, v – временные переменные,  
локальные для блока,  
a, b, c, d – переменные, живые на  
выходе из блока.

Для инструкции 2:  $u \leftarrow -, a, c$  необходимо сгенерировать команды:

- ♦ загрузка регистра  $R_c$
- ♦ вычитание (результат на регистре  $R_u$ )

| $R_1$ | $R_2$ | $R_3$ | a        | b | c | d | t     | u | v |
|-------|-------|-------|----------|---|---|---|-------|---|---|
| a     | t     |       | a, $R_1$ | b | c | d | $R_2$ |   |   |

```
LD R3, c
SUB R1, R1, R2
```

*getReg()* выдает  $R_3$  для  $R_c$  и  $R_1$  для  $R_u$

| $R_1$ | $R_2$ | $R_3$ | a | b | c        | d | t     | u     | v |
|-------|-------|-------|---|---|----------|---|-------|-------|---|
| u     | t     | c     | a | b | c, $R_3$ | d | $R_2$ | $R_1$ |   |

## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

```
1 t ← -, a, b
2 u ← -, a, c
3 v ← +, t, u
4 a ← d
5 d ← +, v, u
```

t, u, v – временные переменные,  
локальные для блока,  
a, b, c, d – переменные, живые на  
выходе из блока.

Для инструкции 3:  $v \leftarrow +, t, u$  необходимо сгенерировать команду:  
♦ сложение (результат на регистре  $R_v$ )

| $R_1$ | $R_2$ | $R_3$ | a | b | c        | d | t     | u     | v |
|-------|-------|-------|---|---|----------|---|-------|-------|---|
| a     | t     | c     | a | b | c, $R_3$ | d | $R_2$ | $R_1$ |   |

ADD R3, R2, R1

*getReg()* выдает  $R_3$  для  $R_v$

| $R_1$ | $R_2$ | $R_3$ | a | b | c | d | t     | u     | v     |
|-------|-------|-------|---|---|---|---|-------|-------|-------|
| u     | t     | v     | a | b | c | d | $R_2$ | $R_1$ | $R_3$ |

## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

```
1 t ← -, a, b
2 u ← -, a, c
3 v ← +, t, u
4 a ← d
5 d ← +, v, u
```

t, u, v – временные переменные,  
локальные для блока,  
a, b, c, d – переменные, живые при  
выходе из блока.

Для инструкции копирования 4:  $a \leftarrow d$  необходимо сгенерировать одну команду:

♦ загрузка регистра  $R_d$

| $R_1$ | $R_2$ | $R_3$ | a | b | c | d | t     | u     | v     |
|-------|-------|-------|---|---|---|---|-------|-------|-------|
| u     | t     | v     | a | b | c | d | $R_2$ | $R_1$ | $R_3$ |

LD  $R_2$ , d

*getReg()* выдает  $R_2$  для  $R_d$

| $R_1$ | $R_2$ | $R_3$ | a | b | c | d        | t | u     | v     |
|-------|-------|-------|---|---|---|----------|---|-------|-------|
| u     | a, d  | v     | a | b | c | d, $R_2$ |   | $R_1$ | $R_3$ |

В регистре  $R_2$  теперь хранятся и  $d$ , и  $a$  (ведь у них одинаковые значения).

t пропало, так как его не запомнили

## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

```
1 t ← -, a, b
2 u ← -, a, c
3 v ← +, t, u
4 a ← d
5 d ← +, v, u
```

t, u, v – временные переменные,  
локальные для блока,  
a, b, c, d – переменные, живые при  
выходе из блока.

Для инструкции 5:  $d = v + u$  необходимо сгенерировать одну команду:

♦ сложение, результат на регистр  $R_d$

| $R_1$ | $R_2$ | $R_3$ | a | b | c | d        | t | u     | v     |
|-------|-------|-------|---|---|---|----------|---|-------|-------|
| u     | a, d  | v     | a | b | c | d, $R_2$ |   | $R_1$ | $R_3$ |

ADD  $R_1$ ,  $R_3$ ,  $R_1$

*getReg()* выдает  $R_1$  для  $R_d$

| $R_1$ | $R_2$ | $R_3$ | a     | b | c | d     | t | u | v     |
|-------|-------|-------|-------|---|---|-------|---|---|-------|
| d     | a     | v     | $R_2$ | b | c | $R_1$ |   |   | $R_3$ |

После этой команды

- ♦ d хранится только на  $R_1$ , но не в ячейке памяти для d.
- ♦ a хранится только на  $R_2$ , но не в ячейке памяти для a.

## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

```
1 t ← -, a, b
2 u ← -, a, c
3 v ← +, t, u
4 a ← d
5 d ← +, v, u
```

**t, u, v** – временные переменные,  
локальные для блока,  
**a, b, c, d** – переменные, живые при  
выходе из блока.

В заключение необходимо сохранить живые переменные **a** и **d**, значения которых есть только на регистрах

| R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | a              | b | c | d              | t | u | v              |
|----------------|----------------|----------------|----------------|---|---|----------------|---|---|----------------|
| d              | a              | v              | R <sub>2</sub> | b | c | R <sub>1</sub> |   |   | R <sub>3</sub> |

ST     a,     R2  
ST     d,     R1

| R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | a                 | b | c | d                 | t | u | v              |
|----------------|----------------|----------------|-------------------|---|---|-------------------|---|---|----------------|
| d              | a              | v              | a, R <sub>2</sub> | b | c | d, R <sub>1</sub> |   |   | R <sub>3</sub> |

## 14.2 Локальное распределение регистров

### 14.2.4 Пример

Генерация кода для базового блока:

```
1 t ← -, a, b
2 u ← -, a, c
3 v ← +, t, u
4 a ← d
5 d ← +, v, u
```

**t**, **u** и **v** – временные переменные, локальные для блока,  
**a**, **b**, **c** и **d** – переменные, живые на выходе из блока.

Сгенерированный код:

```
LD R1, a
LD R2, b
SUB R2, R1, R2
LD R3, c
SUB R1, R1, R3
ADD R3, R2, R1
LD R2, d
ADD R1, R3, R1
ST a, R2
ST d, R1
```

Код содержит  
4 команды **LD**  
2 команды **ST**

Все эти команды связаны с множествами  $In(B)$  и  $Out(B)$  переменных живых при входе в блок  $B$  и при выходе из него

6 команд из 10 связаны с обращениями к памяти



## 14.2 Локальное распределение регистров

### 14.2.5 Реализация функции *getReg*

- ◇ Генерация команды для инструкции  $I$   
$$\mathbf{x} \leftarrow \mathbf{op}, \mathbf{y}, \mathbf{z}$$
- ◇ выбор регистров для операндов  $\mathbf{y}$  и  $\mathbf{z}$
- ◇ выбор регистра для результата  $\mathbf{x}$

## 14.2 Локальное распределение регистров

### 14.2.5 Реализация функции *getReg*

- ◇ Выбор регистра  $R_y$  для операнда  $y$   
(регистр  $R_z$  для операнда  $z$  выбирается аналогично) .
  - ◇ Если  $DA[y]$  ссылается на регистр  $R$ , то полагаем  $R_y = R$
  - ◇ Если  $DA[y]$  не содержит ссылок на регистры, но имеется регистр  $R$ , для которого  $D[R]$  не содержит ссылок ни на одну переменную, то полагаем  $R_y = R$
  - ◇ Если  $DA[y]$  не содержит ссылок на регистры и не имеется ни одного регистра  $R$ , для которого  $DR[R]$  не содержит ссылок ни на одну переменную, то  $R$  можно использовать в качестве  $R_y$ , если для каждой переменной  $v$ , ссылка на которую содержится  $DR[R]$ , выполняется одно из следующих условий:
    - ◆  $DA[v]$  содержит ссылку не только на  $R$ , но и на адрес  $v$ ,
    - ◆  $v$  это переменная  $x$ , вычисляемая командой  $I$ , и  $x$  не является одновременно одним из операндов команды  $I$ .
    - ◆ переменная  $v$  после команды  $I$  больше не используется.
  - ◇ Если ни одна из перечисленных выше ситуаций не имеет места, то прежде чем использовать  $R$  в качестве  $R_y$ , необходимо выполнить сброс регистра, т.е. команду **ST  $v$ ,  $R$**

## 14.2 Локальное распределение регистров

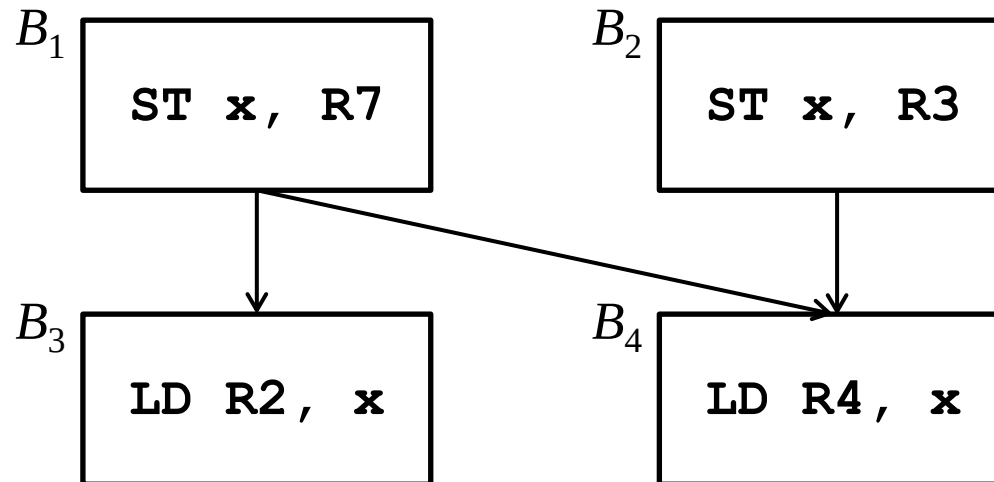
### 14.2.5 Реализация функции *getReg*

- ◇ Выбор регистра  $R_x$  для результата  $\mathbf{x}$ 
  - ◇ Если  $DR[R]$  ссылается только на  $x$ , то полагаем  $R_x = R$   
Это можно делать даже тогда, когда  $x$  является одним из  $y$  или  $z$ , так как в одной машинной команде допускается совпадение двух регистров.
  - ◇ Если  $y$  не используется после команды  $I$  и если  $DR[R_y]$  ссылается только на  $y$ , то  $R_y$  может использоваться в роли  $R_x$ .
  - ◇ Если  $z$  не используется после команды  $I$  и если  $DR[R_z]$  ссылается только на  $z$ , то  $R_z$  может использоваться в роли  $R_x$ .
- ◇ Генерация команд для инструкции  $I$   
 $\mathbf{x} \leftarrow \mathbf{y}$   
Сначала выбирается  $R_y$ , так же, как и для операнда инструкции  $\mathbf{x} \leftarrow \mathbf{op}, \mathbf{y}, \mathbf{z}$ , после чего полагается  $R_x = R_y$ .

## 14.2 Локальное распределение регистров

### 14.2.6 Ограничения

- ◇ В примере на рисунке независимое назначение регистров в базовых блоках привело к тому, что для одной и той же переменной **x** в каждом блоке используются разные регистры
- ◇ Если бы *getReg* блока  $B_3$  знала, что в блоке  $B_1$  значение **x** было получено на **R7**, то выделила бы для **x** регистр **R7**, что позволило бы исключить команду загрузки на регистр в блоке  $B_3$
- ◇ Наличие блоков  $B_2$  и  $B_4$  еще больше усложняет проблему, так как возникают различные требования на разных путях



## 14.3 Глобальное распределение и назначение регистров

### 14.3.1 Интервалы жизни

- ◇ *Интервалом жизни (ИЖ)* значения  $w$  в переменной  $v$  называется множество команд программы, начиная с команды, в которой переменная  $v$  определяется со значением  $w$ , и кончая последней командой, в которой переменная  $v$  используется с этим значением.

|    |     |    |         |  |       |        |
|----|-----|----|---------|--|-------|--------|
| 0  | LD  | R0 | ...     |  | $R_0$ | [0,10] |
| 1  | LD  | R1 | R0 (0)  |  | $R_1$ | [1,6]  |
| 2  | LD  | R2 | 2       |  | $R_1$ | [6,7]  |
| 3  | LD  | R3 | R0 (@x) |  | $R_1$ | [7,8]  |
| 4  | LD  | R4 | R0 (@y) |  | $R_1$ | [8,9]  |
| 5  | LD  | R5 | R0 (@z) |  | $R_1$ | [9,10] |
| 6  | MUL | R1 | R1 R2   |  | $R_2$ | [2,6]  |
| 7  | MUL | R1 | R1 R3   |  | $R_3$ | [3,7]  |
| 8  | MUL | R1 | R1 R4   |  | $R_4$ | [4,8]  |
| 9  | MUL | R1 | R1 R5   |  | $R_5$ | [5,9]  |
| 10 | ST  | v  | R0 (0)  |  |       |        |

## 14.3 Глобальное распределение и назначение регистров

### 14.3.2 Построение интервалов жизни

- ◇ **Построение множеств переменных, живых на выходе из каждого блока.** Это задача анализа потока данных. Методом итераций решается система уравнений

$$LiveOut[B] = \bigcap_{s \in Succ(B)} (use_s \cup (LiveOut[S] - VarKill_s)) \quad (1)$$

где

$LiveOut(B)$  – множество переменных, живых на выходе из блока  $B$ ,

$use_B$  – множество переменных блока  $B$ , которые используются в  $B$  **до** их переопределения в  $B$ ,

$VarKill_B$  – множество переменных блока  $B$ , которые переопределяются в  $B$  до их использования (в анализе живых переменных это множество обозначается как  $def_B$ ).

- ◇ Решив методом итераций систему (1),  
**получим  $LiveOut(B)$  для всех  $B$ .**

## 14.3 Глобальное распределение и назначение регистров

### 14.3.2 Построение интервалов жизни

- ◇ Построение множеств переменных, живых на выходе из каждого блока. Это задача анализа потока данных. Методом итераций решается система уравнений

$$LiveOut[B] = \bigcap_{s \in Succ(B)} (use_s \cup (LiveOut[S] - VarKill_s)) \quad (1)$$

Не удивительно, что система уравнений (1) – это не что иное, как система, решаемая при анализе живых переменных; в ней просто изменены обозначения, чтобы повысить эмоциональность: например, множество  $VarKill_B$  в анализе живых переменных обозначалось  $def_B$ .

переопределяются в  $B$  до их использования.

- ◇ Решив методом итераций систему (1), получим  $LiveOut(B)$  для всех  $B$ .

## **14.3 Глобальное распределение и назначение регистров**

### **14.3.2 Построение интервалов жизни**

- ◇ **Исследование отношений между различными определениями и использованиями.** Нужно построить множества всех определений, достигающих одного и того же использования (*UD-цепочки*), и всех использований, которые достигаются одним и тем же определением (*DU-цепочки*).
  - ◇ Такое построение удобно проводить в *SSA*-форме, так как в ней
    - ◇ каждое имя определяется только один раз,
    - ◇ каждое использование ссылается на единственное имя, а объединение имен обеспечивается с помощью  $\phi$ -функций.
- Для программы в *SSA*-форме требуемая группировка имен достигается за один просмотр ее текста.
- ◇ Алгоритм объединения имен анализирует каждую  $\phi$ -функцию и строит объединение множеств, связанных с каждым ее параметром, и множества, связанного с определяемой ею переменной. Это объединение и является интервалом жизни.
  - ◇ После обработки всех  $\phi$ -функций строится отображение *SSA*-имен на имена ИЖ.



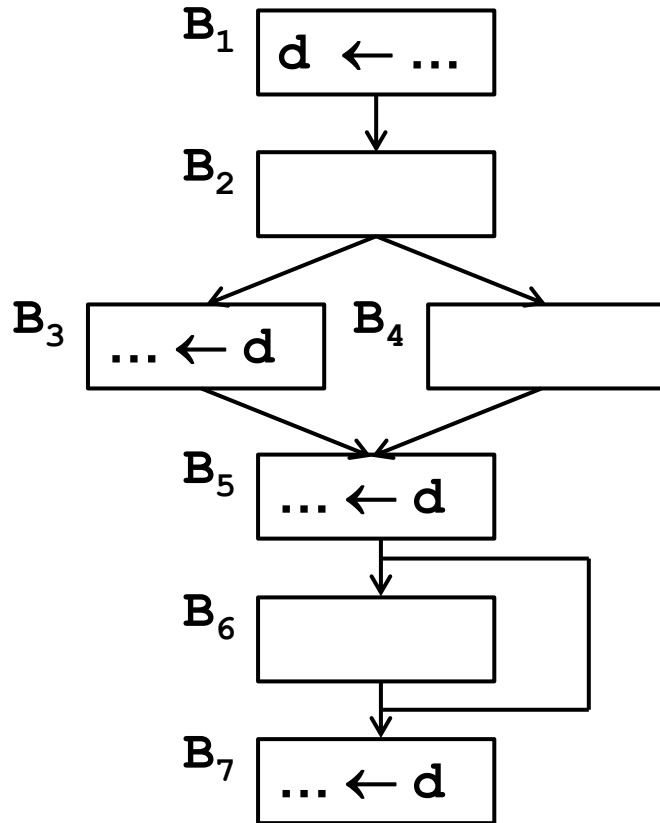
## 14.3 Глобальное распределение и назначение регистров

### 14.3.2 Построение интервалов жизни

- ◇ **Исследование отношений между различными определениями и использованиями.** Нужно построить множества всех определений, достигающих одного и того же использования (*UD-цепочки*), и всех использований, которые достигаются одним и тем же определением (*DU-цепочки*).
- ◇ Такое построение удобно проводить в *SSA*-форме, так как в ней
  - ◇ каждое имя определяется только один раз,
  - ◇ каждое использование ссылается на единственное имя, а объединение имен обеспечивается с помощью  $\phi$ -функций.Для программы в *SSA*-форме требуемая группировка имен достигается за один просмотр ее текста.
- ◇ Для ускорения работы просматриваются не инструкции, а множества переменных, живых на выходе из базового блока; если переменная входит в такое множество, то весь базовый блок включается в ее ИЖ интервалом жизни.
- ◇ После обработки всех  $\phi$ -функций строится отображение *SSA*-имен на имена ИЖ.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.2 Построение интервалов жизни. Простой пример.



Du-цепочка псевдорегистра (переменной)  $d$ :

$\{B_1, B_3, B_5, B_7\}$



Интервал жизни  $LR_d$  псевдорегистра (переменной)  $d$ :

$\{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$

(блоки перечислены в топологическом порядке).



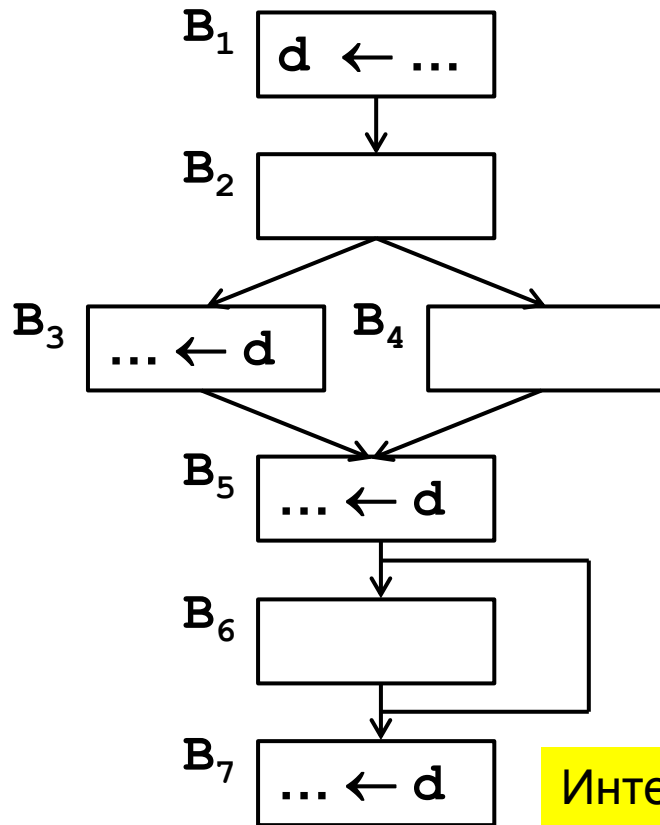
В течение всего интервала жизни на регистре будет находиться значение  $d$ , определенное в блоке  $B_1$ .



Переменная входит в множества **Input** каждого из блоков  $B_i$ , кроме первого и в множества **Output** каждого из блоков  $B_i$ , кроме последнего

# 14.3 Глобальное распределение и назначение регистров

## 14.3.2 Построение интервалов жизни. Простой пример.



Du-цепочка псевдорегистра  
(переменной)  $d$ :

$\{B_1, B_3, B_5, B_7\}$



Интервал жизни  $LR_d$  псевдорегистра  
(переменной)  $d$ :

$\{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$

(блоки перечислены в топологическом  
порядке).

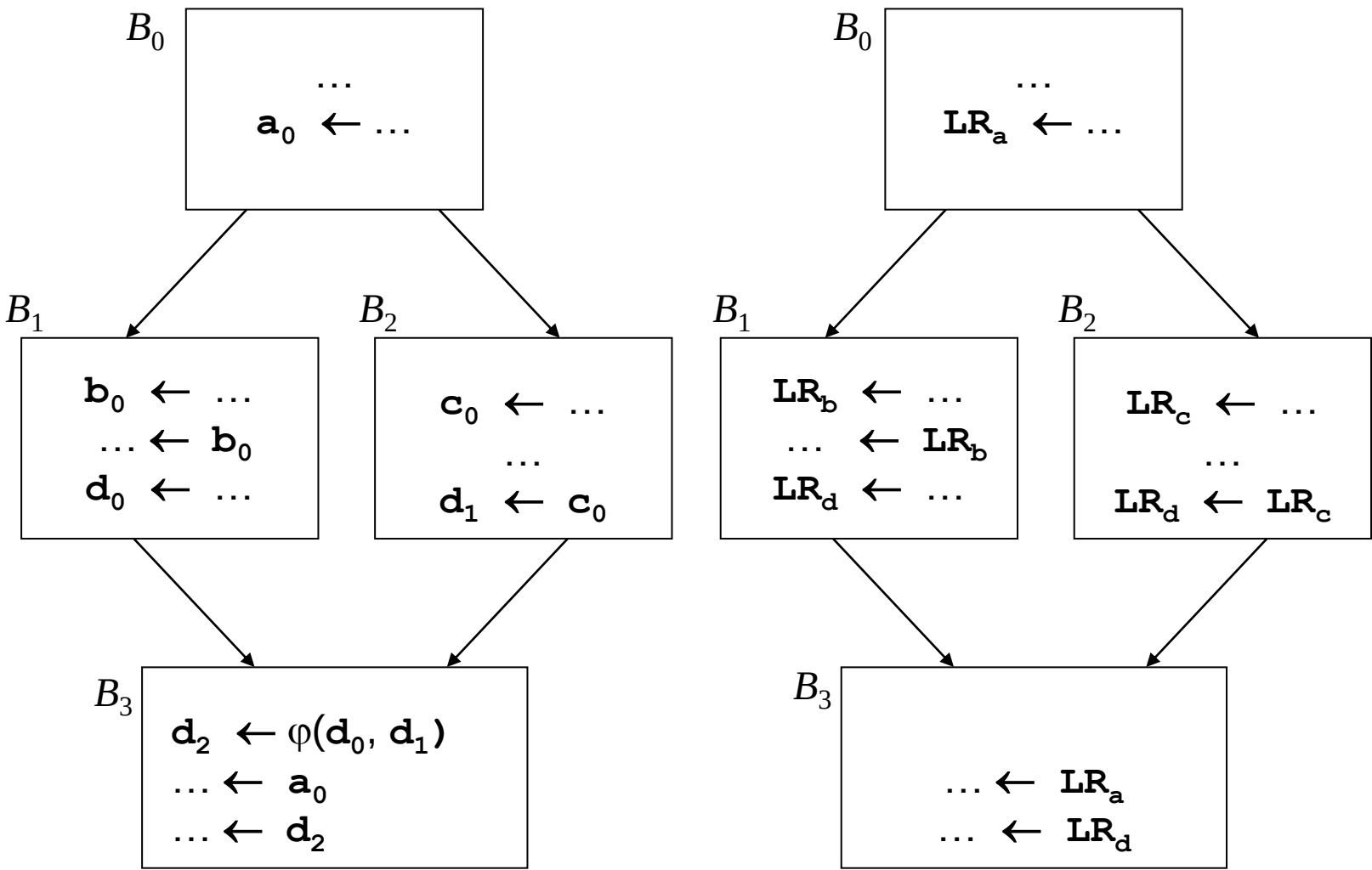


В течение всего интервала жизни на  
регистре будет находиться  
значение  $d$ , определенное в блоке  $B_1$ .

Интервал жизни переменной является пересечением множества базовых блоков (вершин ГПУ), в которых она жива, и множества базовых блоков (вершин ГПУ), которых она может достичь.

# 14.3 Глобальное распределение и назначение регистров

## 14.3.3 Построение интервалов жизни. Пример

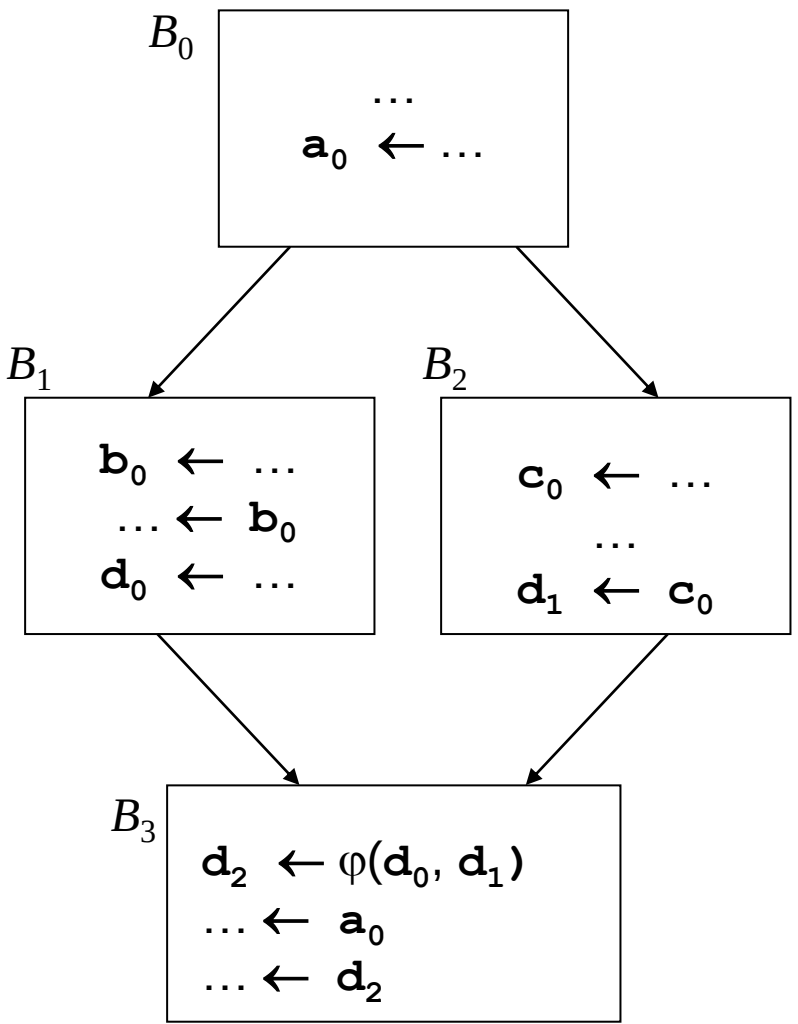


Фрагмент кода  
в SSA-форме

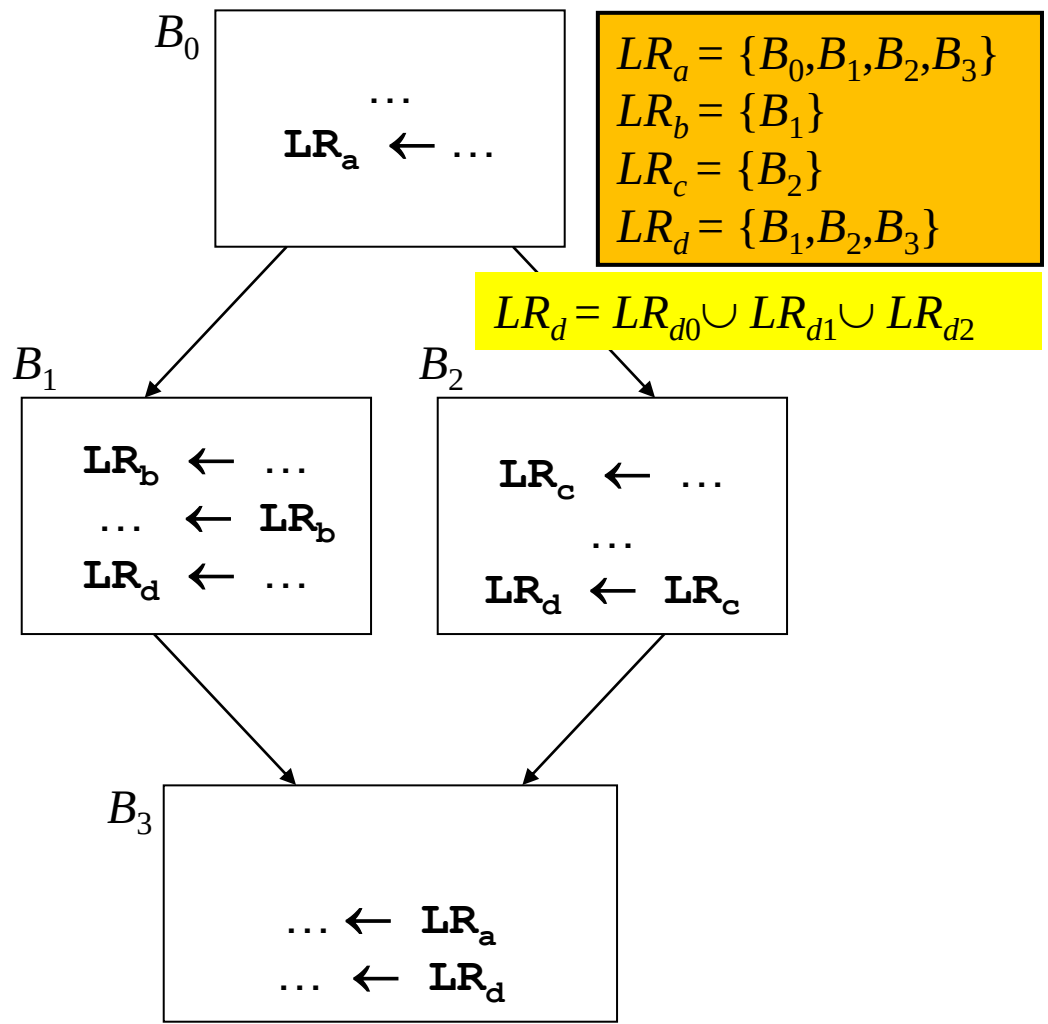
Тот же фрагмент, переписанный  
в терминах ИЖ

# 14.3 Глобальное распределение и назначение регистров

## 14.3.3 Построение интервалов жизни. Пример



Фрагмент кода  
в SSA-форме



Тот же фрагмент, переписанный  
в терминах ИЖ

## 14.3 Глобальное распределение и назначение регистров

### 14.3.4 Оценка стоимости сброса

- ◇ Стоимость сброса складывается из следующих трех компонент:
  - ◇ стоимость вычисления адресов при сбросе
  - ◇ стоимость операций доступа к памяти
  - ◇ оценка частоты выполнения
- ◇ Для хранения сброшенных значений во фрейме процедуры выделяется специальная область.  
Это позволяет свести к минимуму стоимость вычисления адресов при сбросе, исключив использование косвенной адресации и дополнительных регистров для вычисления адреса сбрасываемого значения.
- ◇ ИЖ, который содержит только команды загрузки регистра и его сохранения может иметь ***отрицательную стоимость сброса*** в случае, когда обе команды обращаются к одному и тому же адресу памяти. Такие ситуации могут возникнуть вследствие исключения избыточных вычислений.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.4 Оценка стоимости сброса

- ◇ Чтобы учитывать частоту выполнения базовых блоков в графе потока управления, каждый базовый блок снабжается *аннотацией*, содержащей *оценку стоимости* его выполнения (для ее вычисления используется профилирование).
- ◇ Для получения грубой оценки стоимости частоты выполнения используется *простая эвристика*:  
делается допущение, что каждый цикл выполняется 10 раз; тогда стоимость каждой загрузки внутри одного цикла оценивается как 10, внутри гнезда из двух циклов – как 100 и т.д.; стоимость каждой ветви непредсказуемого **if-then-else** оценивается как 0.5.  
На практике из этой эвристики, в частности, следует, что сброс выгоднее выполнять в более внешнем цикле.
- ◇ Аннотации могут вычисляться заранее (и тогда потребуются дополнительный просмотр программы), либо во время первого обращения

## 14.3 Глобальное распределение и назначение регистров

### 14.3.5 Конфликтные ситуации и граф конфликтов

- ◇ При распределении регистров моделируется состязание за место на регистрах целевой машины.  
Рассмотрим два различных ИЖ  $LR_i$  и  $LR_j$ . Если в программе есть команды, во время которых и  $LR_i$ , и  $LR_j$  актуальны, они не могут занимать один и тот же регистр.  
В таком случае говорят, что  $LR_i$  и  $LR_j$  находятся в конфликте.
- ◇ **Определение.** Интервалы жизни  $LR_i$  и  $LR_j$  *находятся в конфликте* если один из них актуален при определении другого и они имеют различные значения.
- ◇ Граф, узлы которого соответствуют отдельным ИЖ, а дуги соединяют ИЖ, находящиеся в конфликте, называется *графом конфликтов* (ГК).  
Этот граф не является направленным, так как отношение нахождения в конфликте симметрично.
- ◇ Таким образом, если два узла ГК являются смежными (соединены дугой), то им должны соответствовать различные регистры.



## 14.3 Глобальное распределение и назначение регистров

### 14.3.8 Построение графа конфликтов

- ◇ После построения глобальных ИЖ и множеств *LiveOut* для каждого базового блока можно за один просмотр программы от *Exit* к *Entry* построить ГК.
- ◇ Внутри каждого базового блока алгоритм поддерживает множество *LiveNow* переменных, живых в текущей точке блока. При входе в блок *B* полагают  $LiveNow = LiveOut(B)$ . Затем просматривают каждую команду блока (*op*  $LR_a, LR_b, LR_c$ ) и узел ГК, соответствующий переменной  $LR_a$ , определяемой в этой команде, соединяют дугами со всеми узлами, соответствующими переменным из *LiveNow* (это как раз переменные, живые во время определения другой переменной).
- ◇ Для повышения эффективности ГК задается левой (нижней) половиной своей матрицы смежности.
- ◇ Операции копирования  $\mathbf{LR}_i \leftarrow \mathbf{LR}_j$  не порождают конкуренции между  $LR_i$  и  $LR_j$ , так как эти значения могут занимать один и тот же регистр.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.8 Построение графа конфликтов

◇ Более точно алгоритм можно выразить следующим образом:

```
for each LR_i do
 create a node $n_i \in N$;
for each basic block b do{
 LiveNow = LiveOut(b)
 for $I_n, I_{n-1}, \dots, I_1$ in b do{
 || I_k имеет вид $op_k LR_a, LR_b, LR_c$
 for each $LR_i \in \text{LiveNow}$ {
 add (LR_a, LR_i) to E
 remove LR_a from LiveNow
 add LR_b to LiveNow
 add LR_c to LiveNow
 }
 }
}
```

## 14.3 Глобальное распределение и назначение регистров

### 14.3.9 Слияние интервалов жизни

- ◇ Рассмотрим команду копирования  $LR_i = LR_j$ . Если ИЖ  $LR_i$  и  $LR_j$  не находятся в конфликте (не являются смежными узлами ГК), то команду можно исключить и все ссылки на  $LR_j$  заменить ссылками на  $LR_i$ . В результате ИЖ  $LR_i$  и  $LR_j$  как бы *сольются*.
- ◇ *Слияние* ИЖ приносит следующие выгоды:
  - ◇ Исключается команда копирования (код становится меньше и тем самым потенциально быстрее)
  - ◇ Снижается степень каждого узла (ИЖ), который был в конфликте либо с  $LR_i$ , либо с  $LR_j$ .
  - ◇ Множество ИЖ сокращается (в литературе приводится пример, когда в результате слияния ИЖ удалось исключить свыше 30% всех ИЖ).

## 14.3 Глобальное распределение и назначение регистров

### 14.3.9 Слияние интервалов жизни

◇ **Пример.** Рассмотрим фрагмент программы:

|       |                                                           |   |
|-------|-----------------------------------------------------------|---|
| ADD   | <b>LR<sub>a</sub></b> , LR <sub>t</sub> , LR <sub>u</sub> | a |
| ..... |                                                           |   |
| LD    | <b>LR<sub>b</sub></b> , LR <sub>a</sub>                   | b |
| LD    | <b>LR<sub>c</sub></b> , <b>LR<sub>a</sub></b>             | c |
| ..... |                                                           |   |
| ADD   | LR <sub>x</sub> , <b>LR<sub>b</sub></b> , LR <sub>w</sub> |   |
| ADD   | LR <sub>z</sub> , <b>LR<sub>c</sub></b> , LR <sub>y</sub> |   |

Отрезки справа от кода отмечают ИЖ **LR<sub>a</sub>** **LR<sub>b</sub>** **LR<sub>c</sub>**  
(красным показано определение переменной, зеленым – ее последнее использование).

Несмотря на то, что ИЖ **LR<sub>a</sub>** пересекается и с **LR<sub>b</sub>**, и с **LR<sub>c</sub>**, он не находится в конфликте ни с тем, ни с другим, так как источник и приёмник соответствующих команд копирования не находятся в конфликте.

**LR<sub>b</sub>** и **LR<sub>c</sub>** находятся в конфликте, так как **LR<sub>b</sub>** жив при определении **LR<sub>c</sub>**  
ИЖ из обеих операций копирования являются кандидатами на слияние.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.9 Слияние интервалов жизни

◇ После слияния  $LR_a$  и  $LR_b$ :

|       |           |           |        |    |
|-------|-----------|-----------|--------|----|
| ADD   | $LR_{ab}$ | $LR_t$    | $LR_u$ | ab |
| ..... |           |           |        |    |
| LD    | $LR_c$    | $LR_{ab}$ |        | c  |
| ..... |           |           |        |    |
| ADD   | $LR_x$    | $LR_{ab}$ | $LR_w$ |    |
| ADD   | $LR_z$    | $LR_c$    | $LR_y$ |    |

◇ После слияния ИЖ  $LR_a$  и  $LR_b$  получаем новый ИЖ  $LR_{ab}$ .

◇ Теперь ИЖ  $LR_c$  получается из  $LR_{ab}$  командой копирования, и следовательно  $LR_c$  и  $LR_{ab}$  не находятся в конфликте, так что слияние  $LR_a$  и  $LR_b$  в  $LR_{ab}$  понизило степень  $LR_c$ .  
Слияние ИЖ может или уменьшить степени смежных узлов (ИЖ), или оставить их без изменения.

◇ Команда копирования  $LD\ LR_c, LR_{ab}$  позволяет продолжить процесс слияния ИЖ, заменив  $LR_{ab}$  на  $LR_{abc}$ .

## ***14.3 Глобальное распределение и назначение регистров***

### **14.3.12 Перестройка графа конфликтов**

- ◇ Перестройка ГК достигается помимо слияния ИЖ (п. 14.3.9) с помощью *слива* переменных (п. 14.3.13) и *расщепления* ИЖ (п.14.3.14).

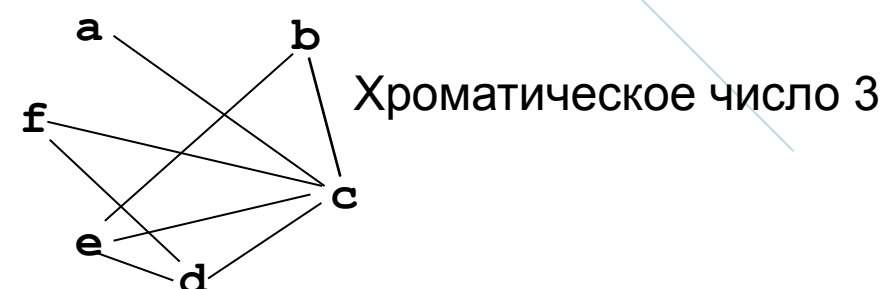
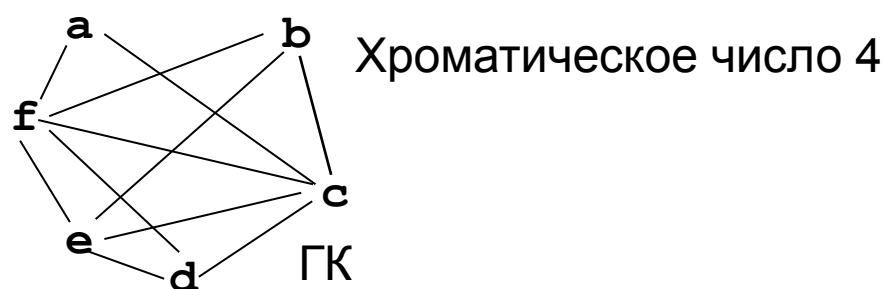
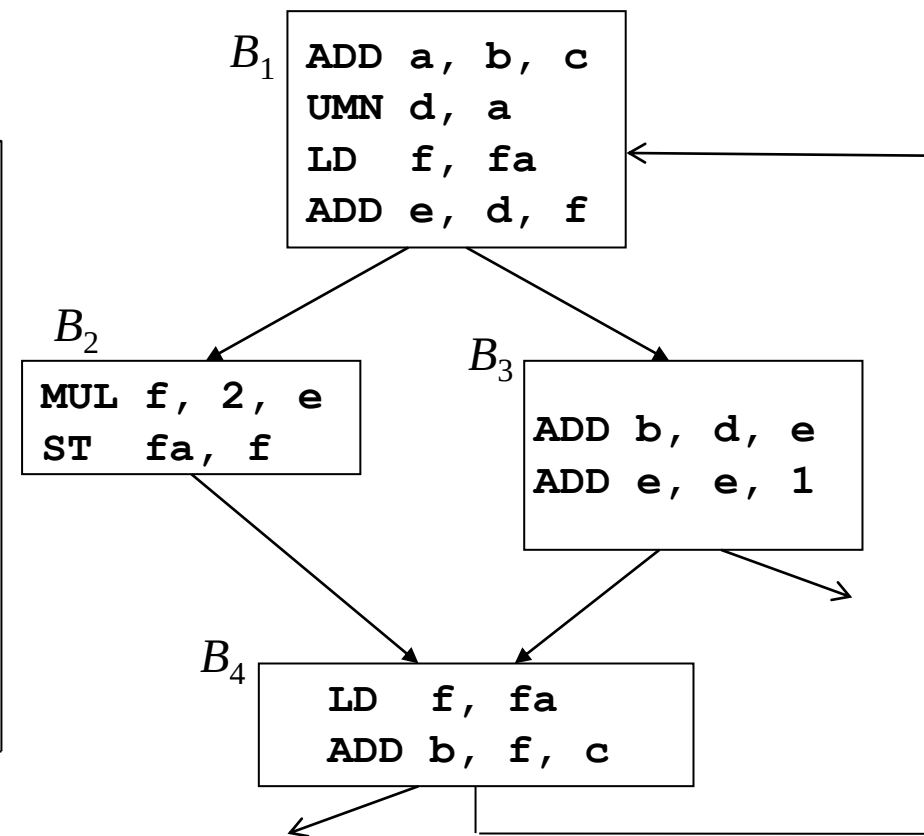
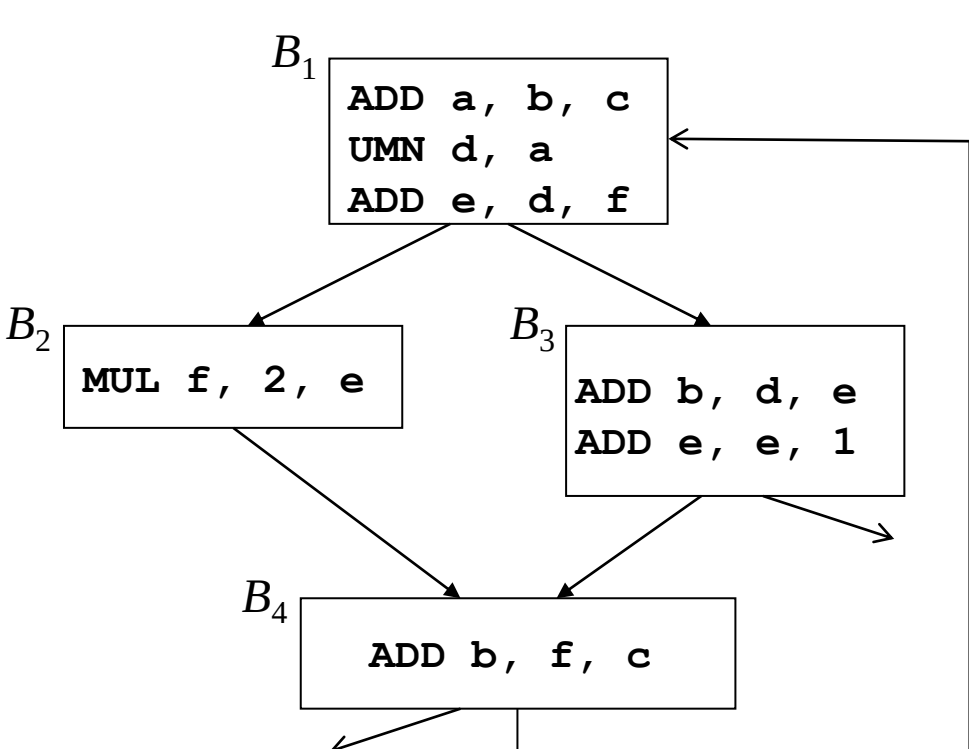
## 14.3 Глобальное распределение и назначение регистров

### 14.3.13 Слив переменных

- ◇ *Слив* временной переменной **t** состоит в следующем:
  - ◇ в «автоматической» памяти процедуры выделить ячейку **ta** для хранения значений **t** (эту ячейку называют «дом» **t**);
  - ◇ перед каждой командой, использующей **t**, вставить команду **LD t, ta**;
  - ◇ после каждой команды, определяющей **t**, вставить команду **ST ta, t**;
  - ◇ просматривают текст полученной процедуры с целью выявить и удалить лишние команды **LD** и **ST** (иногда это удается).

# 14.3 Глобальное распределение и назначение регистров

## 14.3.13 Слив переменных





## 14.3 Глобальное распределение и назначение регистров

### 14.3.14 Расщепление интервалов жизни

```
ADD a, m, 1
ADD b, k, 4
```

```
ADD a, m, 1
ST mem, a
ADD b, k, 4
```

◇ На левом рисунке  $LR_a$  и  $LR_b$  полностью пересекаются. Расщепив  $LR_a$  на два  $LR$ , удастся устранить конфликт.

```
ADD r, b, m
```

```
ADD r, b, m
```

◇ При этом команды **ST** и **LD** применяются только здесь, а не перед всеми **a**.

```
LD a, mem
```

```
ADD a, m, 1
```

```
ADD a, m, 1
```

**a** и **b** в конфликте

Конфликт между  
**a** и **b** устранен

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

- ◇ Пусть доступно  $R$  регистров и пусть составлен список из  $n$  ИЖ. Алгоритм *Linear scan* должен просмотреть список ИЖ и распределить на регистры как можно больше ИЖ таким образом, чтобы никакие пересекающиеся ИЖ не были распределены на один и тот же регистр.
- ◇ Если  $n > R$  и ИЖ пересекаются в некоторой точке, то по крайней мере  $n - R$  ИЖ должны быть размещены в памяти.
- ◇ Количество пересекающихся ИЖ изменяется только в начале и в конце одного из ИЖ.
- ◇ Все ИЖ помещаются в список, отсортированный по возрастанию их начал. Тогда алгоритм может быстро просмотреть все ИЖ, перескакивая с одного начала ИЖ к следующему. На каждом шагу алгоритм поддерживает список активных ИЖ, которые пересекаются в данной точке и уже размещены на регистрах. Список активных ИЖ поддерживается отсортированным по возрастанию концов входящих в него ИЖ.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

- ◇ Пусть доступно  $R$  регистров и пусть составлен список из  $n$  ИЖ. Алгоритм *Linear scan* должен просмотреть список ИЖ и распределить на регистры как можно больше ИЖ таким образом, чтобы никакие пересекающиеся ИЖ не были распределены на один и тот же регистр.
- ◇ Если  $n > R$  и ИЖ пересекаются в некоторой точке, то по крайней
- ◇ М Отметим, что алгоритм *Linear scan* обходит не ГПУ, а список ИЖ, так что подробностей ГПУ он просто не замечает.
- ◇ Все ИЖ помещаются в список, отсортированный по возрастанию их начал. Тогда алгоритм может быстро просмотреть все ИЖ, перескакивая с одного начала ИЖ к следующему. На каждом шагу алгоритм поддерживает список активных ИЖ, которые пересекаются в данной точке и уже размещены на регистрах. Список активных ИЖ поддерживается отсортированным по возрастанию концов входящих в него ИЖ.

### 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◆ На псевдокоде

```
active ← {}
```

for each ИЖ  $i$  в порядке возрастания начал

```
ExpireOldIntervals(i) //окончание
 старых интервалов
```

**if length(active) = R then**

**SpillAtInterval(i)** //слив интервала i

**else**

**register[i] ← регистр, удаляемый из  
пула свободных регистров**

**добавить i в active, отсортированный  
по возрастанию концов ИЖ**

## ***14.3 Глобальное распределение и назначение регистров***

### **14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)**



На псевдокоде

```
ExpireOldIntervals(i)
```

```
foreach interval j in active, in order of
increasing end point {
```

```
 if endpoint[j] ≤ startpoint[i] then
```

```
 remove j from active
```

```
 add register[j] to pool of free
 registers
```

```
}
```

```
return
```

## **14.3 Глобальное распределение и назначение регистров**

### **14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)**



На псевдокоде

**SpillAtInterval(i)**

**spill**  $\leftarrow$  last interval in active

**if** endpoint[spill] > end point[i] **then**

**register**[i]  $\leftarrow$  **register**[spill]

**location**[spill]  $\leftarrow$  new stack location

**remove** spill from active

**add** i to active, sorted by increasing  
        end point

**else**

**location**[i]  $\leftarrow$  new stack location

## ***14.3 Глобальное распределение и назначение регистров***

### **14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)**

- ◇ При обнаружении каждого нового ИЖ алгоритм просматривает список **active** с начала до конца. Во время просмотра из списка удаляются все закончившиеся ИЖ и делает соответствующие регистры доступными для распределения.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◇ Пример. Пусть доступно всего два регистра **r1** и **r2**.

|                                             | <b>v1</b> | <b>v2</b> | <b>v3</b> | <b>v4</b> | <b>v5</b> |
|---------------------------------------------|-----------|-----------|-----------|-----------|-----------|
| (1) <b>v1</b> = 10                          |           |           |           |           |           |
| (2) <b>v2</b> = 20                          |           |           |           |           |           |
| (3) <b>v3</b> = <b>v1</b> + <b>v2</b>       |           |           |           |           |           |
| (4) <b>v4</b> = <b>v2</b> + <b>v3</b>       |           |           |           |           |           |
| (5) <b>v1</b> = <b>v3</b> + <b>v4</b>       |           |           |           |           |           |
| (6) <b>v5</b> = <b>v4</b> + <b>v1</b>       |           |           |           |           |           |
| (7) <b>return</b> ( <b>v1</b> + <b>v5</b> ) |           |           |           |           |           |

◇ ИЖ обрабатываются в порядке **v1**, **v2**, **v3**, **v4**, **v5**.

Сначала список **Active** пуст.

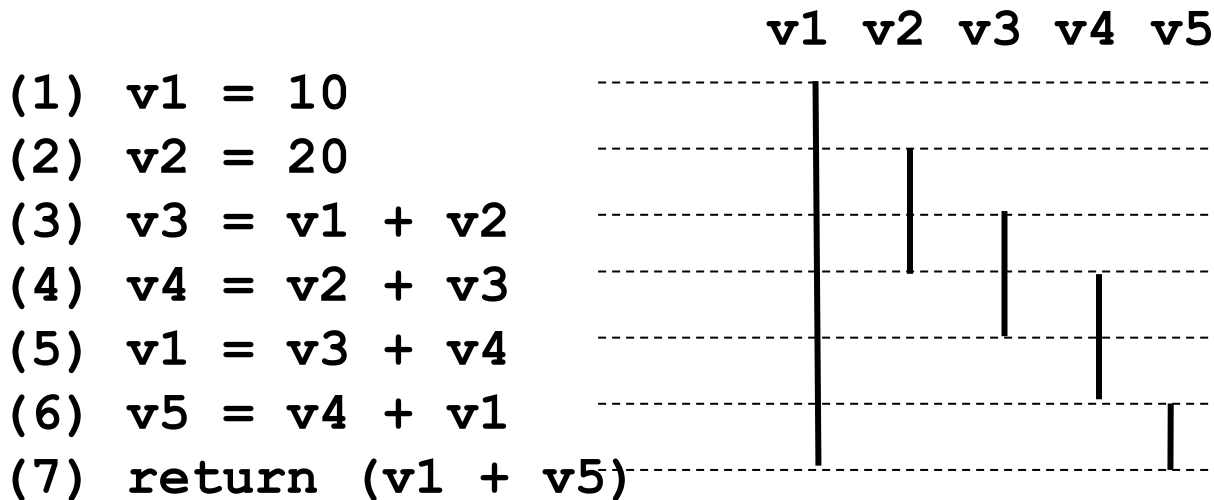
На первом шаге обрабатывается ИЖ **v1**. Так как список **Active** пуст, регистр **r1** выделяется для **v1**, и **v1** заносится в список **Active**.



## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◇ Пример. Пусть доступно всего два регистра **r1** и **r2**.

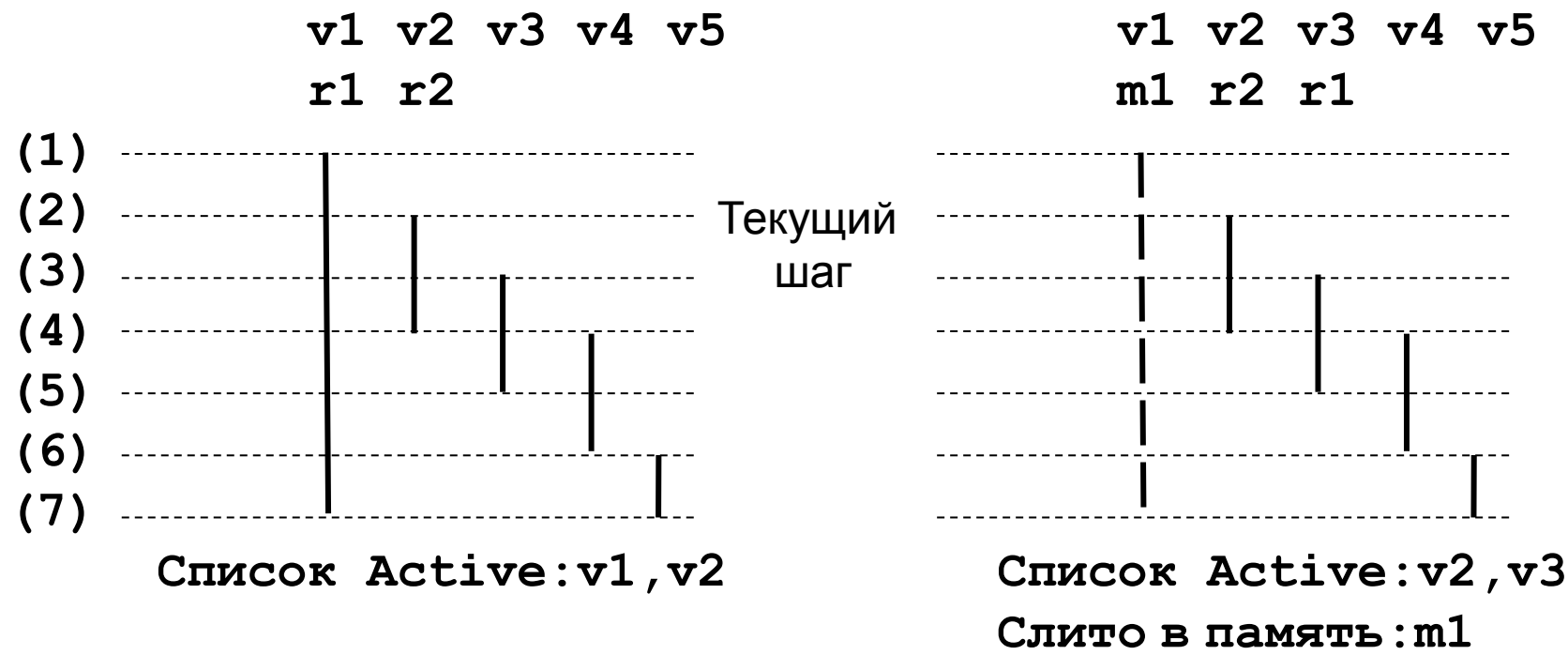


◇ ИЖ обрабатываются в порядке **v1**, **v2**, **v3**, **v4**, **v5**.  
После первого шага список **Active** содержит ИЖ **v1**.  
На втором шаге обрабатывается ИЖ **v2**. Так как **v1** продолжает оставаться активным, **v1** не исключается из списка **Active**, и для ИЖ **v2** выделяется единственный свободный регистр **r2**, а **v2** заносится в список **Active**.

# 14.3 Глобальное распределение и назначение регистров

## 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◇ Пример. Пусть доступно всего два регистра r1 и r2.



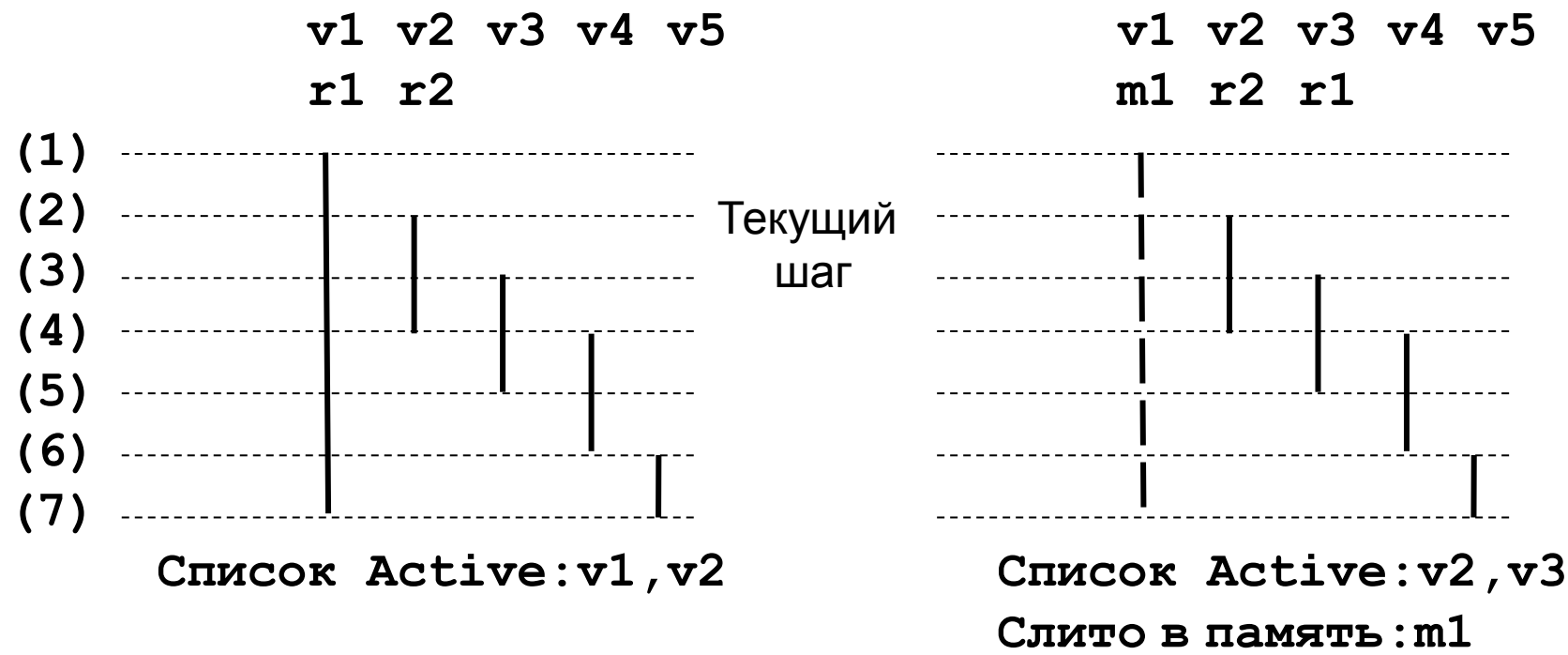
Текущий шаг

◇ На рисунках показано состояние до и после третьего шага.

# 14.3 Глобальное распределение и назначение регистров

## 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◇ **Пример.** Пусть доступно всего два регистра **r1** и **r2**.



На третьем шаге пересекаются три ИЖ: еще не закончившиеся ИЖ **v1** и **v2** и новый ИЖ **v3**. Значит один из ИЖ нужно слить. Согласно алгоритму слит будет ИЖ, который закончится позже, т.е. ИЖ **v1**.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◇ Пример. Пусть доступно всего два регистра  $r1$  и  $r2$ .

|                                 |                                 |
|---------------------------------|---------------------------------|
| (1) $v1 = 10$                   | (1) $m1 = 10$                   |
| (2) $v2 = 20$                   | (2) $r2 = 20$                   |
| (3) $v3 = v1 + v2$              | (3) $r1 = m1 + r2$              |
| (4) $v4 = v2 + v3$              | (4) $r2 = r2 + r1$              |
| (5) $v1 = v3 + v4$              | (5) $m1 = r1 + r2$              |
| (6) $v5 = v4 + v1$              | (6) $r1 = r2 + m1$              |
| (7) <b>return</b> ( $v1 + v5$ ) | (7) <b>return</b> ( $m1 + r1$ ) |

◇ Полученное распределение регистров показано на рисунке. Метод раскраски графа конфликтов обеспечивает для рассмотренного примера лучшее распределение регистров (без  $m1$ ).

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

- ◇ **Пример.** Пусть доступно всего два регистра `r1` и `r2`.
- ◇ Метод раскраски графа конфликтов обеспечивает для рассмотренного примера лучшее распределение регистров.

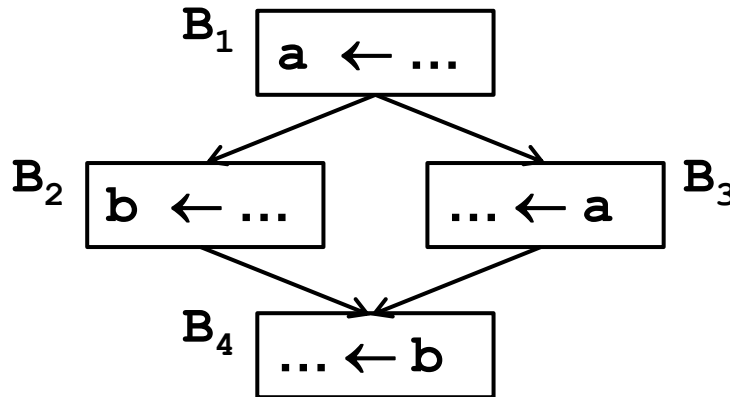
|                                   |                                   |
|-----------------------------------|-----------------------------------|
| (1) <code>v1 = 10</code>          | (1) <code>r2 = 10</code>          |
| (2) <code>v2 = 20</code>          | (2) <code>r1 = 20</code>          |
| (3) <code>v3 = v1 + v2</code>     | (3) <code>r2 = r2 + r1</code>     |
| (4) <code>v4 = v2 + v3</code>     | (4) <code>r1 = r1 + r2</code>     |
| (5) <code>v1 = v3 + v4</code>     | (5) <code>r2 = r2 + r1</code>     |
| (6) <code>v5 = v4 + v1</code>     | (6) <code>r1 = r1 + r2</code>     |
| (7) <code>return (v1 + v5)</code> | (7) <code>return (r2 + r1)</code> |

- ◇ Но метод раскраски графа конфликтов имеет асимптотическую временную сложность  $O(n^2)$ .  
Метод линейного сканирования реализует распределение регистров за один просмотр текста программы и имеет асимптотическую временную сложность  $O(n)$ .

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◇ Пример. Пусть доступно всего два регистра  $r1$  и  $r2$ .



ИЖ  $a = \{B_1, B_3\}$

ИЖ  $b = \{B_2, B_4\}$

ИЖ не пересекаются.

Значит для  $a$  и  $b$  может быть выделен один и тот же регистр.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

- ◇ В отличие от метода раскраски графа конфликтов, имеющего асимптотическую временную сложность  $O(n^2)$ , метод линейного сканирования реализует распределение регистров всего за один просмотр текста программы и имеет асимптотическую временную сложность  $O(n)$ .
- ◇ Разработчики JIT-компиляторов (например, для языка *Java* или для языка *Python*) предпочитают метод линейного сканирования.
- ◇ В последнее время используется модификация метода линейного сканирования, называемая *Second Chance Binpacking*. Во многих случаях она позволяет увеличить точность линейного сканирования, сохраняя асимптотическую временную сложность  $O(n)$ . Рассмотрим кратко этот метод.

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – модифицированный метод линейного сканирования (Second Chance Binpacking)

- ◇ Одним из основных недостатков метода линейного сканирования является тот факт, что он не рассматривает возможных «дыр» в ИЖ.  
В то же время анализ ГПУ реальных программ показывает, что такие дыры нередко возникают в связи с многочисленными ветвлениями. Даже простой пример, рассмотренный выше, подтверждает это.
- ◇ В самом деле ИЖ для  $v1$  имеет дыру от инструкции (3) до инструкции (5): значение  $v1 == 10$  после инструкции (3) не используется, а в инструкции (5) переопределяется.

```
(1) v1 = 10
(2) v2 = 20
(3) v3 = v1 + v2
(4) v4 = v2 + v3
(5) v1 = v3 + v4
(6) v5 = v4 + v1
(7) return (v1 + v5)
```



## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – модифицированный метод линейного сканирования (Second Chance Binpacking)



В самом деле ИЖ для  $v1$  имеет дыру от инструкции (3) до инструкции (5): значение  $v1 == 10$  после инструкции (3) не используется, а в инструкции (5) вообще переопределяется.

Методу линейного сканирования, который игнорирует этот факт, приходится и после инструкции (3) хранить в памяти (так как регистров не хватает) мертвое значение  $v1$ .



В модифицированном методе реализована возможность расщепления ИЖ.

Эта возможность позволяет расщепить ИЖ  $v1$  на два ИЖ –  $v11$  и  $v12$ . В итоге  $v1$  получает второй шанс занять регистр.

Отметим, что если бы использовалась SSA-форма, расщепления бы не потребовалось, оно бы произошло автоматически.

(1)  $v1 = 10$

(2)  $v2 = 20$

(3)  $v3 = v1 + v2$

(4)  $v4 = v2 + v3$

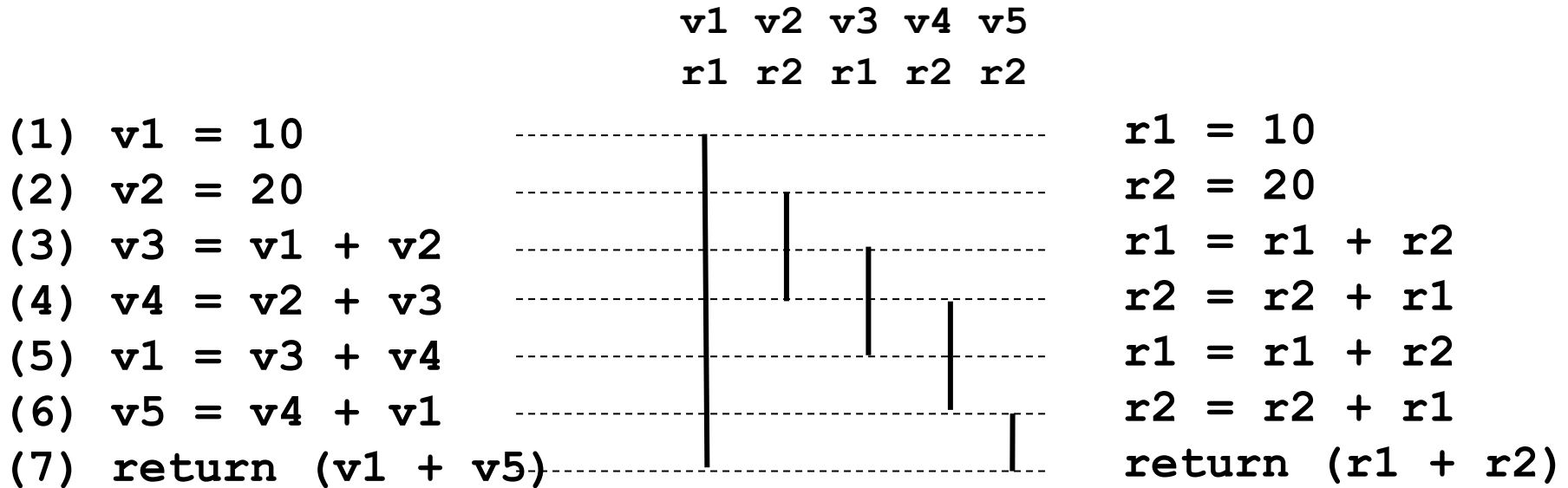
(5)  $v1 = v3 + v4$

(6)  $v5 = v4 + v1$

(7) **return** ( $v1 + v5$ )

## 14.3 Глобальное распределение и назначение регистров

### 14.3.11 Быстрый алгоритм распределения регистров – модифицированный метод линейного сканирования (Second Chance Binpacking)



В модифицированном алгоритме ИЖ **v1** сначала помещается в список **Active**, на третьем шаге переводится в список **Inactive**, а на пятом шаге снова помещается в список **Active**. В итоге получается распределение регистров, показанное на рисунке.

## **14.3 Глобальное распределение и назначение регистров**

### **14.3.11 Быстрый алгоритм распределения регистров – модифицированный метод линейного сканирования (Second Chance Binpacking)**

- ◇ Модифицированный алгоритм имеет **примерно** такую же скорость, как и базовый алгоритм линейного сканирования, и при этом генерирует код **примерно** такого же качества, что и алгоритм раскраски графа конфликтов.

# **15. Планирование кода**

## **15.1 Постановка задачи**

### **15.1.1 Цель планирования кода**

- ◇ Цель планирования кода – выбрать такую последовательность команд, которая, *не меняя семантики программы*, обеспечит *близкое к оптимальному использование особенностей архитектуры целевого процессора*
- ◇ Прежде всего необходимо обеспечить использование возможностей параллельного выполнения команд, реализованных в аппаратуре

## 15.1 Постановка задачи

### 15.1.2 Сохранение семантики программы

- ◇ Требование сохранения семантики программы проще всего выразить в форме ограничений, которым должна удовлетворять целевая программа.  
**Эти ограничения должны гарантировать, что оптимизированная программа будет давать такие же результаты, что и исходная.**
- ◇ На планирование кода накладывается следующие **три типа ограничений**:
  - ◇ *Ограничения управления.* Все операции, выполняемые в исходной программе, должны выполняться и в оптимизированной программе.
  - ◇ *Ограничения данных.* Операции в оптимизированной программе должны выдать те же результаты, что и соответствующие операции в исходной программе
  - ◇ *Ограничения ресурсов.* Планирование кода не должно требовать чрезмерного количества ресурсов.

## 15.1 Постановка задачи

### 15.1.2 Сохранение семантики программы

- ◇ Требование сохранения семантики программы проще всего выразить в форме ограничений, которым должна удовлетворять целевая программа.  
**Эти ограничения должны гарантировать, что оптимизированная программа будет давать такие же результаты, что и исходная.**
- ◇ На планирование кода накладывается следующие **три типа ограничений**:
  - ◇ *Ограничения управления.* Все операции, выполняемые в исходной программе, должны выполняться и в оптимизированной программе.
  - ◇ *Ограничения данных.* Операции в оптимизированной программе должны выдать те же результаты, что и соответствующие операции в исходной программе
  - ◇ *Ограничения ресурсов.* Планирование кода не должно требовать чрезмерного количества ресурсов.

## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.1 Конвейер обработки команд

◇ Каждый современный процессор (ядро) использует конвейер (обработки) команд.

◇ При использовании конвейера выполнение каждой команды производится поэтапно.

◇ На рисунке показан простой пятиэтапный конвейер команд: на первом этапе выполняется выборка команды (**IF**), затем – ее декодирование (**ID**), выполнение (**EX**), обращение к памяти (**MEM**) и запись результата (**WB**).

◇ На рисунке показано, как одновременно могут выполняться команды  $i$ ,  $i + 1$ ,  $i + 2$ ,  $i + 3$  и  $i + 4$ .

Каждая строка соответствует такту системных часов, а каждый столбец на рисунке определяет, на каком такте выполняются этапы каждой из команд.

После разгона конвейера параллельно **может** выполняться пять команд.

|        | $i$ | $i + 1$ | $i + 2$ | $i + 3$ | $i + 4$ |
|--------|-----|---------|---------|---------|---------|
| 1. IF  |     |         |         |         |         |
| 2. ID  | IF  |         |         |         |         |
| 3. EX  | ID  | IF      |         |         |         |
| 4. MEM | EX  | ID      | IF      |         |         |
| 5. WB  | MEM | EX      | ID      | IF      |         |
| 6.     | WB  | MEM     | EX      | ID      |         |
| 7.     |     | WB      | MEM     | EX      |         |
| 8.     |     |         | WB      | MEM     |         |
| 9.     |     |         |         | WB      |         |



## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.1 Конвейер обработки команд

- ◇ Большинство процессоров общего назначения динамически определяют зависимости между последовательными командами и автоматически приостанавливают выполнение команды в случае недоступности ее операндов.
- ◇ Некоторые процессоры, оставляют проверку зависимостей программному обеспечению по соображениям простоты и снижения энергопотребления процессора.  
В этом случае компилятор отвечает за вставку в код команд операций *NOP* («нет операции») там, где необходимо гарантировать доступность результатов предыдущей операции.
- ◇ Многие процессоры могут обрабатывать сразу несколько команд. Параллельное выполнение на таких машинах тоже может управляться как аппаратно, так и программно. Машины с программной обработкой параллельности известны под аббревиатурой *VLIW* (*Very Long Instruction Word*); машины с аппаратной обработкой известны под названием *суперскалярных*.

## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.2 Современные процессоры

- ◇ Процессоры *VLIW*, как следует из их названия, имеют команды большого размера, которые содержат несколько *операций*. Компилятор решает, какие операции будут выполняться параллельно, и явно указывает эту информацию в машинном коде.
- ◇ Суперскалярные процессоры автоматически обнаруживают зависимости между операциями и выполняют операции, как только их операнды становятся доступными.
- ◇ **Простые аппаратные планировщики** выполняют операции в том порядке, в котором выполняется их выборка. Если планировщик обнаруживает зависимую операцию, команда, содержащая эту операцию, и все последующие команды должны дождаться, пока зависимости не будут разрешены (то есть пока не станут доступны все требующиеся результаты выполнения предыдущих команд). Очевидно, что таким процессорам выгодна работа со статическим планировщиком, который размещает независимые операции одна за другой в порядке выполнения.<sup>7</sup>

## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.2 Современные процессоры

- ◇ Процессоры *VLIW*, как следует из их названия, имеют команды большого размера, которые содержат несколько *операций*. Компилятор решает, какие операции будут выполняться параллельно, и явно указывает эту информацию в машинном коде.
- ◇ Суперскалярные процессоры автоматически обнаруживают зависимости между операциями и выполняют операции, как только их операнды становятся доступными.
- ◇ **Простые аппаратные планировщики** выполняют операции в том порядке, в котором выполняется их выборка. Если планировщик обнаруживает зависимую операцию, команда, содержащая **Поток команд приостанавливается** и команды должны дожждаться, пока зависимости не будут разрешены (то есть пока не станут доступны все требующиеся результаты выполнения предыдущих команд). Очевидно, что таким процессорам выгодна работа со статическим планировщиком, который размещает независимые операции одна за другой в порядке выполнения.

## **15.2 Аппаратные возможности параллельного выполнения команд**

### **15.2.2 Современные процессоры**

- ◇ Более сложные планировщики могут выполнять команды «не по порядку» (*Out of Order*). Отдельные операции независимо приостанавливаются и не выполняются до тех пор, пока не будут получены все значения, от которых эти операции зависят. Но статические планировщики могут помочь в работе даже столь интеллектуальным планировщикам, поскольку аппаратные планировщики ограничены объемом памяти, в котором могут храниться отложенные операции. Статический же планировщик может разместить независимые операции поближе одна к другой с тем, чтобы более эффективно использовать аппаратные возможности процессора.
- ◇ Следует отметить, что независимо от возможностей аппаратного планировщика, он не в состоянии работать с командами, которые еще не были выбраны. Компилятор может повысить производительность динамического планировщика, обеспечивая возможность параллельного выполнения вновь выбираемых команд.

## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.2 Современные процессоры

- ◇ Более сложные планировщики могут выполнять команды «не по порядку» (*Out of Order*). **Отдельные операции независимо приостанавливаются** и не выполняются до тех пор, пока не будут получены результаты операций, от которых они зависят. **Поток команд не приостанавливается**. Но статические планировщики могут помочь в работе даже столь интеллектуальным планировщикам, поскольку аппаратные планировщики ограничены объемом памяти, в котором могут храниться отложенные операции. Статический планировщик может разместить независимые операции поближе одна к другой с тем, чтобы более эффективно использовать аппаратные возможности процессора.
- ◇ Следует отметить, что независимо от возможностей аппаратного планировщика, он не в состоянии работать с командами, которые еще не были выбраны. Компилятор может повысить производительность динамического планировщика, обеспечивая возможность параллельного выполнения вновь выбираемых команд.

## **15.2 Аппаратные возможности параллельного выполнения команд**

### **15.2.3 Параллелизм на уровне команд**

- ◇ Параллелизм на уровне команд был впервые использован в компьютерной архитектуре как средство повышения скорости выполнения обычного последовательного кода.
- ◇ Компиляторы компьютеров, не содержащих средств параллельного выполнения команд, переупорядочивали команды таким образом, чтобы минимизировать количество используемых регистров.
- ◇ При применении таких компиляторов на новых компьютерах они, экономя регистры, одновременно минимизировали и возможности параллельного выполнения команд.  
В результате в последовательном коде оставалось очень мало возможностей для обеспечения параллелизма на уровне команд.
- ◇ Одной из первых мер по обеспечению параллельного выполнения команд (операций) стала возможность аппаратного переименования регистров для отмены результатов компилятора по оптимизации использования регистров.

## **15.2 Аппаратные возможности параллельного выполнения команд**

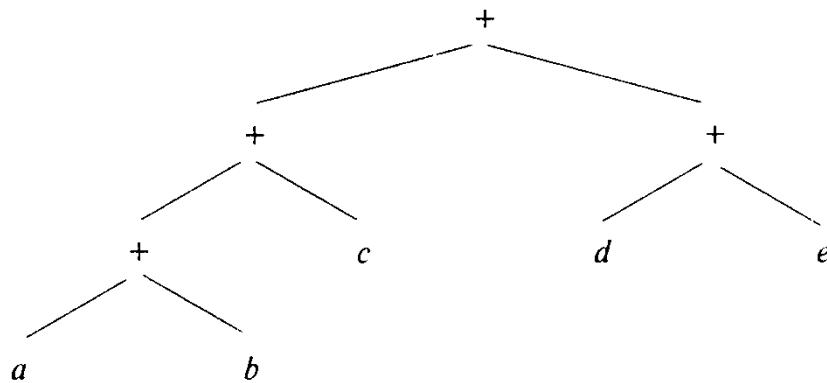
### **15.2.4 Аппаратное переименование регистров**

- ◇ **Аппаратное переименование регистров** динамически изменяет назначение регистров в процессе выполнения программы.  
Оно интерпретирует машинный код, сохраняет значения, предназначенные для одного и того же регистра, в различных внутренних регистрах, и изменяет все их использования так, чтобы обращения к соответствующим регистрам выполнялись корректно.
- ◇ Поскольку, в первую очередь, ограничения, связанные с зависимостями регистров, возникают благодаря работе компилятора, они могут быть устранены путем использования алгоритма распределения регистров, который учитывает параллелизм на уровне команд.

## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.4 Аппаратное переименование регистров

- ◇ **Пример.** Вычисление выражения  $(a + b) + c + (d + e)$  (см. дерево на рисунке) можно выполнить с использованием трех регистров (числа Ершова):



```
LD r1, a // r1 = a
LD r2, b // r2 = b
ADD r1, r1, r2 // r1 = r1+r2
LD r2, c // r2 = c
ADD r1, r1, r2 // r1 = r1+r2
LD r2, d // r2 = d
LD r3, e // r3 = e
ADD r2, r2, r3 // r2 = r2+r3
ADD r1, r1, r2 // r1 = r1+r2
```

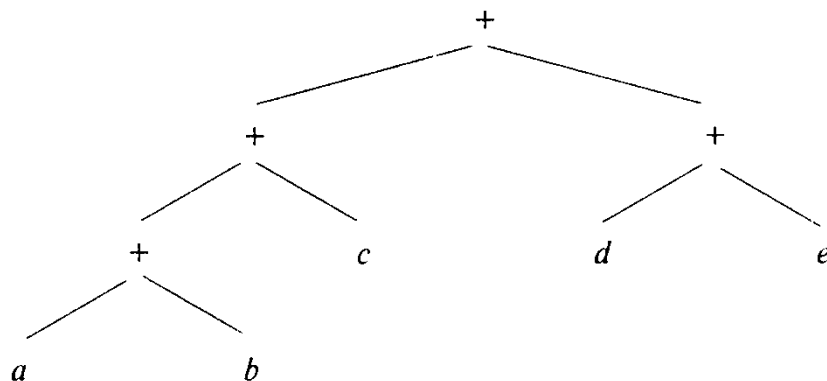
- ◇ Как видно из текста процедуры, для этого регистры **r1** и **r2** необходимо использовать несколько раз. Повторное использование регистров требует последовательного вычисления.
- ◇ Единственные операции, которые могут быть выполнены параллельно: загрузки значений из ячеек **a** и **b** и загрузки значений из ячеек **d** и **e**. Таким образом, всего для полного вычисления выражения с максимальным использованием параллельности требуется 7 шагов вместо 9.



## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.4 Аппаратное переименование регистров

- ◇ **Пример.** Вычисление выражения  $(a + b) + c + (d + e)$  (см. дерево на рисунке) можно выполнить с использованием трех регистров (числа Ершова):



```
LD r1, a // r1 = a
LD r2, b // r2 = b
ADD r1, r1, r2 // r1 = r1+r2
LD r2, c // r2 = c
ADD r1, r1, r2 // r1 = r1+r2
LD r2, d // r2 = d
LD r3, e // r3 = e
ADD r2, r2, r3 // r2 = r2+r3
ADD r1, r1, r2 // r1 = r1+r2
```

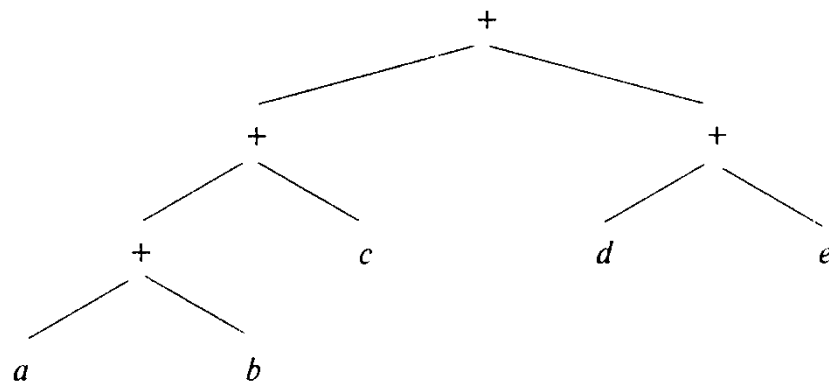
- ◇ Если использовать различные регистры для каждой промежуточной суммы (то есть **использовать не 3, а 5 регистров**), то выражение можно вычислить всего за 4 шага, что равно высоте дерева выражения.

|            |            |        |        |        |
|------------|------------|--------|--------|--------|
| r1 = a     | r2 = b     | r3 = c | r4 = d | r5 = e |
| r6 = r1+r2 | r7 = r4+r5 |        |        |        |
| r8 = r6+r3 |            |        |        |        |
| r9 = r8+r7 |            |        |        |        |

## 15.2 Аппаратные возможности параллельного выполнения команд

### 15.2.4 Аппаратное переименование регистров

- ◇ **Пример.** Вычисление выражения  $(a + b) + c + (d + e)$  (см. дерево на рисунке) можно выполнить с использованием трех регистров (числа Ершова):



```
LD r1, a // r1 = a
LD r2, b // r2 = b
ADD r1, r1, r2 // r1 = r1+r2
LD r2, c // r2 = c
ADD r1, r1, r2 // r1 = r1+r2
LD r2, d // r2 = d
LD r3, e // r3 = e
ADD r2, r2, r3 // r2 = r2+r3
ADD r1, r1, r2 // r1 = r1+r2
```

- ◇ Регистр **r1** переименовывается в **r3**,  
**r2** – в **r4**, **r3** – в **r5** (каждой промежуточной  
команде), то выражение  
с помощью специальных команд процессора строится по схеме дерева выражения.

|            |            |        |        |        |
|------------|------------|--------|--------|--------|
| r1 = a     | r2 = b     | r3 = c | r4 = d | r5 = e |
| r6 = r1+r2 | r7 = r4+r5 |        |        |        |
| r8 = r6+r3 |            |        |        |        |
| r9 = r8+r7 |            |        |        |        |

## 15.3 Модель процессора

### 15.3.1 Базовая модель процессора

- ◇ Многие процессоры можно описать, используя следующую модель: процессор представляется парой  $P = \langle R, T \rangle$ ,
  - ◇  $T$  – множество типов операций, таких как загрузки (на регистры), сохранения (в память), арифметические операции и т.п.;
  - ◇  $R = [r_1, r_2, \dots]$  – вектор, представляющий аппаратные ресурсы:  $r_i$  – количество доступных единиц  $i$ -го ресурса.
- ◇ **Примеры аппаратных ресурсов:**
  - ◇ модуль обращения к памяти (МОП),
  - ◇ арифметико-логическое устройство (АЛУ),
  - ◇ функциональное устройство (ФУ),
  - ◇ модуль для работы с числами с плавающей точкой (МПТ).
- ◇ **Примеры функциональных устройств:** сумматор, умножитель, ...

## 15.3 Модель процессора

### 15.3.1 Базовая модель процессора

- ◇ *Команда* может содержать одну или более *операций*.  
Для начала выполнения операции необходимо наличие:
  - ◇ входных операндов,
  - ◇ ресурсов, требующихся для выполнения операции.
- ◇ С каждым *входным операндом* связана *входная задержка*, указывающая, через сколько тактов после начала выполнения операции должно быть доступно его значение.  
Типичные входные операнды имеют нулевую задержку, означающую, что их значения нужны в момент начала выполнения операции.
- ◇ С каждым *выходным операндом* связана *выходная задержка*, указывающая, через сколько тактов после начала выполнения операции становится доступен *результат*.

## 15.4 Анализ зависимостей

### 15.4.1 Зависимости по данным

- ◇ **Определение.** Две команды называются *зависимыми по данным*, если изменение порядка их выполнения может привести к изменению результата вычислений, выполняемых программой.
- ◇ Виды зависимостей:
  - ◇ *Истинная зависимость: чтение после записи.* Если команда  $C_1$  записывает значение в некоторую ячейку памяти, а команда  $C_2$  считывает это значение, то команды  $C_1$  и  $C_2$  зависимы.
  - ◇ *Антизависимость: запись после чтения.* Если команда  $C_1$  считывает значение из некоторой ячейки памяти, а команда  $C_2$  записывает в эту ячейку новое значение, то команды  $C_1$  и  $C_2$  зависимы.
  - ◇ *Зависимость по выходу: запись после записи.* Если команды  $C_1$  и  $C_2$  записывают значения в одну и ту же ячейку памяти, то команды  $C_1$  и  $C_2$  зависимы.

## 15.4 Анализ зависимостей

### 15.4.1 Зависимости по данным

- ◇ **Замечание.** *Неустранимой является только истинная зависимость*, а два других вида зависимостей (их называют *зависимостями, связанными с хранением*) могут быть устранены путем использования в командах  $C_1$  и  $C_2$  разных ячеек памяти для разных значений.
- ◇ В общем случае задача выявления зависимостей алгоритмически неразрешима: компилятор обязан предполагать, что две команды могут обращаться к одним и тем же ячейкам, если он не может установить обратное (консервативность).

## 15.4 Анализ зависимостей

### 15.4.1 Зависимости по данным

◇ Пример: Рассмотрим код

```
1. a = 1
2. *p = 2
15. b = a
```

В этом коде сразу можно обнаружить истинную зависимость между командами 1 и 3. Больше зависимостей как будто нет, но это только в том случае, когда *компилятор может установить, что указатель **p** не может указывать на **a**.*

Если это не так, компилятор обязан считать, что указатель **p** может указывать на **a**, и тогда возникают еще две зависимости:

- ◇ истинная зависимость между командами 2 и 3
- ◇ зависимость по записи между командами 1 и 2.

◇ Пример показывает, что выявление зависимостей связано с *межпроцедурным анализом алиасов.*

## 15.4 Анализ зависимостей

### 15.4.2 Зависимости по данным между итерациями цикла

- ◇ Анализ зависимостей по данным для массива должен выяснить, анализируя индексные выражения обращений к массиву, могут ли два обращения ссылаться на один и тот же элемент массива при каких-нибудь значениях индексных переменных.

- ◇ **Пример:** Рассмотрим цикл

```
for (i=0; i<n; ++i)
 A[2*i] = A[2*i+1];
```

Обращения к массиву **A** в левой и правой частях оператора присваивания независимы, так как ни при каком значении индекса **i** значение индекса в левой части (четное число) не станет равным значению индекса в правой части (нечетное число).

Пример не должен вводить в заблуждение: анализ зависимостей по данным для итераций цикла, содержащих обращения к элементам массивов – сложная задача.



## ***15.4 Анализ зависимостей***

### **15.4.2 Зависимости по данным между итерациями цикла**



В случае, когда все индексы массивов и все границы циклов являются аффинными выражениями задача сводится к задаче целочисленного линейного программирования.

## 15.4 Анализ зависимостей

### 15.4.3 Зависимости по управлению

- ◇ Команда **s1** называется зависящей по управлению от команды **s2**, если выполнение команды **s2** определяет, будет ли выполняться команда **s1**.

#### Простые примеры

- ◇ в конструкции **if (c) s1; else s2; s1** и **s2** зависят по управлению от **c**
- ◇ в конструкции **while (c) s; s** зависит по управлению от **c**

## 15.4 Анализ зависимостей

### 15.4.2 Зависимости по управлению

◇ Команда **s1** называется зависящей по управлению от команды **s2**, если выполнение команды **s2** определяет, будет ли выполняться команда **s1**.

◇ **Пример.** Во фрагменте кода

```
 if (a > t)
 b = a * a;
 d = a + c;
```

◇ команды, соответствующие инструкциям **b = a \* a** и **d = a + c**, не зависят по данным от других частей фрагмента.

◇ команда, соответствующая инструкции **b = a \* a**, зависит по управлению от сравнения **a > t**

◇ команда, соответствующая инструкции **d = a + c**, не зависит от сравнения **a > t** ни по данным, ни по управлению

## **15.5 Планирование базовых блоков**

### **15.5.1 Граф зависимостей по данным**

- ◇ Граф зависимостей по данным  $G = (N, E)$ , содержит
  - ◇ множество узлов  $N$ , представляющих операции в командах базового блока,
  - ◇ множество ориентированных ребер  $E$ , представляющих зависимости по данным между операциями.
- ◇ Граф зависимостей по данным отражает все зависимости по данным, имеющиеся в блоке.

## 15.5 Планирование базовых блоков

### 15.5.1 Граф зависимостей по данным

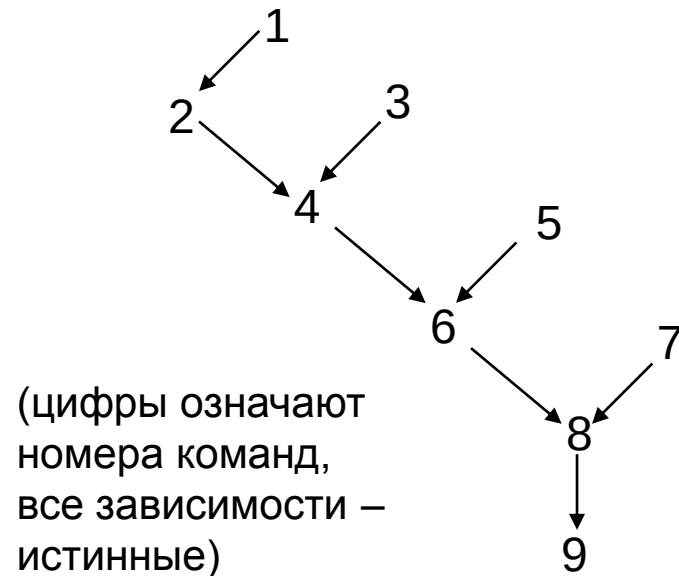
- ◇ Узлы и ребра графа  $G$  строятся следующим образом:
  - ◇ Каждой операции  $n \in N$  ставится в соответствие таблица резервирования ресурсов  $(RT_n)$ , определяемая в соответствии с типом операции  $n$ .  
Каждое ребро  $e \in E$  помечается *задержкой*  $d_e$ , указывающей, что целевой узел может быть выполнен не ранее, чем через  $d_e$  тактов после выполнения исходного узла.
  - ◇ Пусть за операцией  $n_1$  следует операция  $n_2$ , причем обе обращаются к одной и той же ячейке памяти с запаздываниями  $l_1$  и  $l_2$  соответственно (значение в ячейке создается через  $l_1$  тактов после начала первой команды и требуется второй команде через  $l_2$  тактов после ее начала: типичными являются значения  $l_1 = 1$  и  $l_2 = 0$ ).  
Тогда в множество  $E$  необходимо включить ребро  $n_1 \rightarrow n_2$ , помеченное задержкой  $l_1 - l_2$ .

## 15.5 Планирование базовых блоков

### 15.5.1 Граф зависимостей по данным

◇ Пример: Для кода

```
1. LD R1, #0
2. ADD R1, R1, R1
3. LD R2, #8
4. MUL R1, R1, R2
5. LD R2, #16
6. MUL R1, R1, R2
7. LD R2, #24
8. MUL R1, R1, R2
9. ST ..., R1
```



граф зависимостей по данным показан на рисунке

◇ Самый длинный путь в графе зависимостей называется *критическим*. В графе, показанном на рисунке, критическим является путь 1-2-4-6-8-9.

◇ Построение графа зависимостей состоит в обходе программы снизу вверх, построении узла для каждого нового значения и соединения этого узла со всеми узлами, использующими соответствующее значение.

## 15.5 Планирование базовых блоков

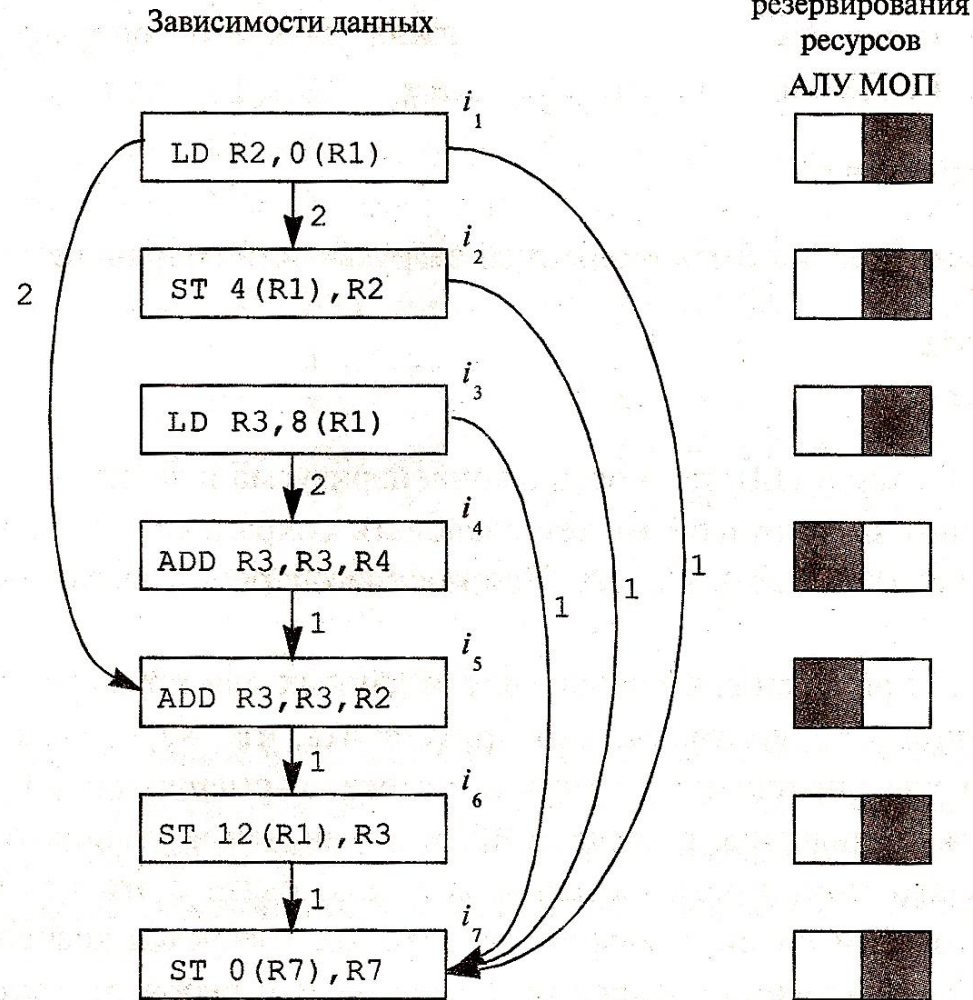
### 15.5.2 Пример

- ◇ Рассмотрим простой процессор (ядро), который может выполнять в каждый момент времени две операции:
  - ◇ первая операция должна быть
    - ◆ либо операцией АЛУ **OP dst, src1, src2,**
    - ◆ либо операцией ветвления
  - ◇ вторая операция должна быть
    - ◆ либо операцией загрузки **LD dst, addr**
    - ◆ либо операцией сохранения **ST addr, src**
- ◇ Операции загрузки (**LD**) и сохранения (**ST**) полностью конвейеризуемы и занимают два такта. Все прочие операции выполняются за один такт.
- ◇ Пусть **R1** – указатель стека данных, причем **d(R1)** обозначает адрес по стеку со смещением **d** относительно **R1** используемый для обращения к данным в стеке со смещениями 0 и 12.

# 15.5 Планирование базовых блоков

## 15.5.2 Пример

- ◇ Еще один пример графа зависимостей по данным.
  - ◇ Таблицу резервирования ресурсов удобно изображать в виде сетки с закрашенными и пустыми ячейками. Каждый столбец соответствует одному из ресурсов, а каждая строка – одному из тактов выполнения операции.
- Пусть никакая операция не требует более одной единицы одного ресурса. Тогда можно представить единицы закрашенными квадратами, а нули – пустыми.





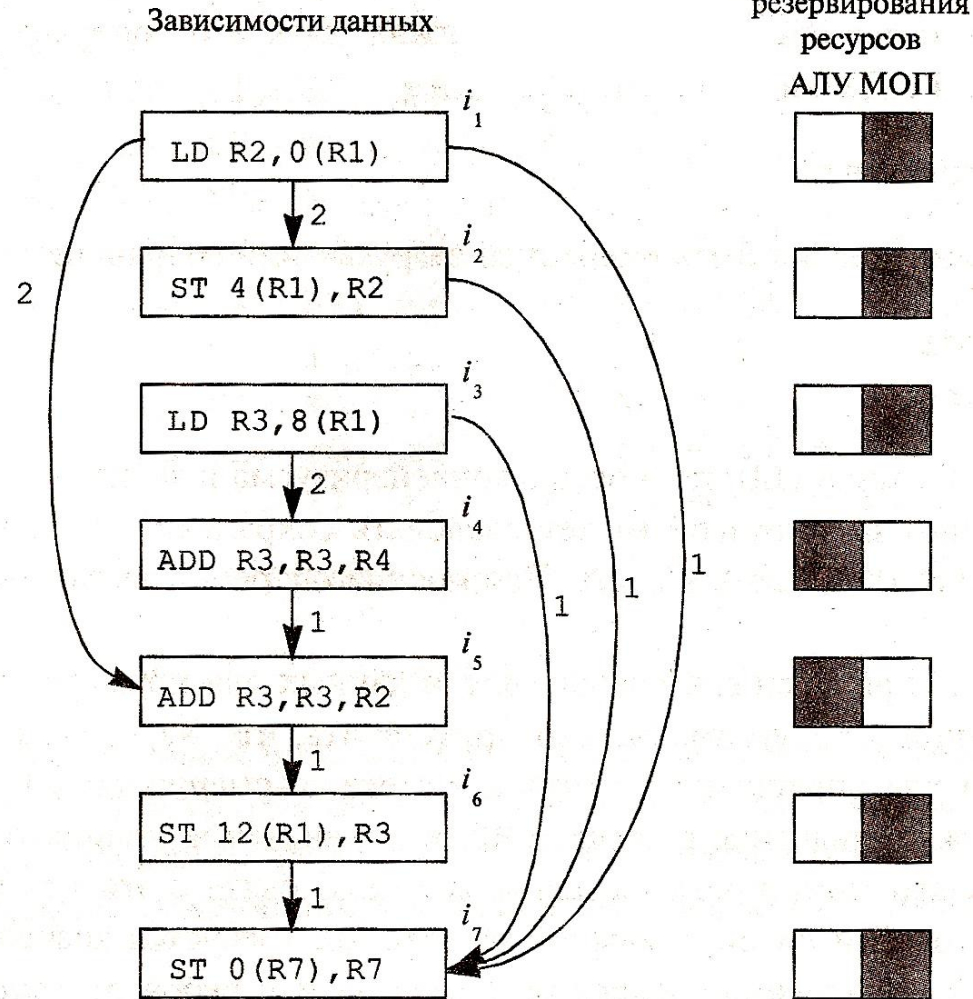
# 15.5 Планирование базовых блоков

## 15.5.2 Пример

Загрузка регистра занимает 2 такта; следовательно, операция, одним из операндов которой является регистр, загружаемый командой (операцией) LD, может начаться не раньше, чем через 2 такта после начала операции загрузки регистра LD.

из ресурсов, а каждая строка – одному из тактов выполнения операции.

Пусть никакая операция не требует более одной единицы одного ресурса. Тогда можно представить единицы закрашенными квадратами, а нули – пустыми.

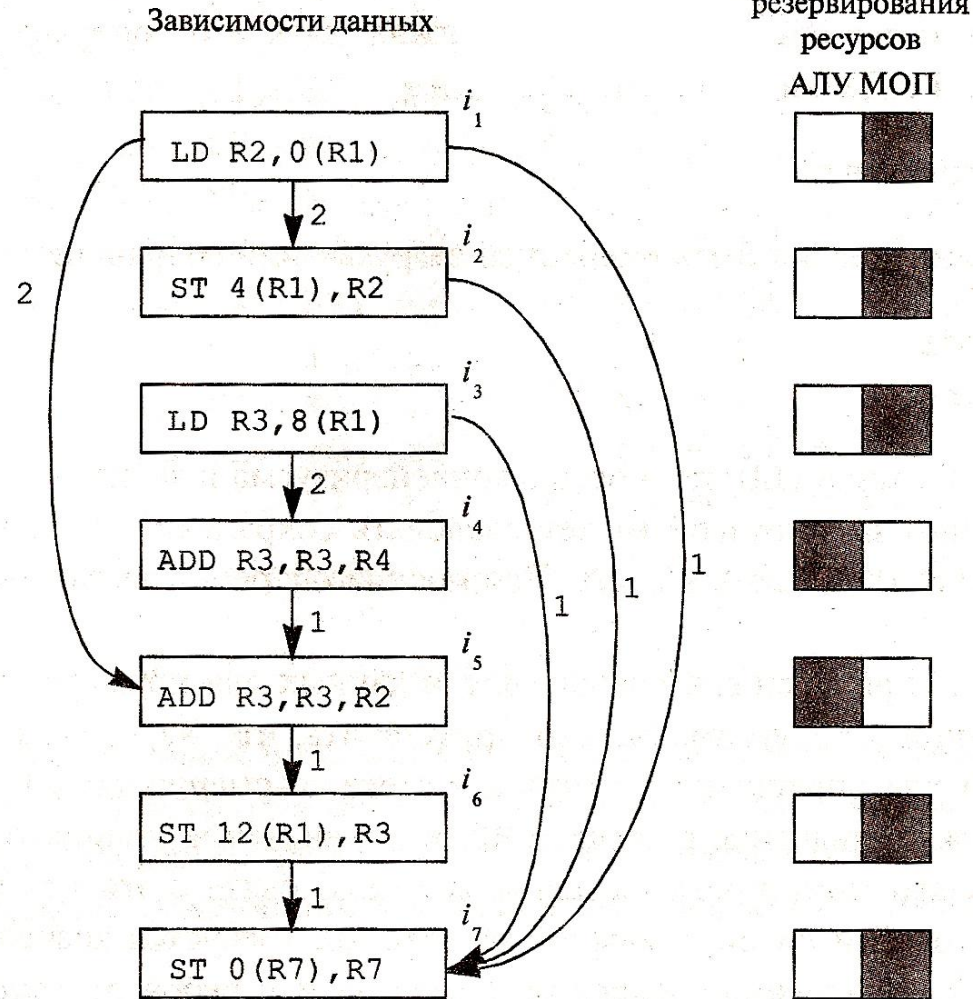


# 15.5 Планирование базовых блоков

## 15.5.2 Пример

Загрузка регистра занимает 2 такта; следовательно, операция, одним из операндов которой является регистр, загружаемый командой (операцией) LD, может начаться не раньше, чем через 2 такта после начала операции загрузки регистра LD.

Арифметическая операция (например, ADD) занимает 1 такт; следовательно, операция ST сохранения ее результата, может начаться через один такт; операции LD и ST, имеющие разные операнды (регистры и адреса памяти) независимы и могут выдаваться на каждом такте.



## **15.5 Планирование базовых блоков**

### **15.5.3 Алгоритм планирования списка**

- ◇ Алгоритм планирования списка работает без откатов: каждый узел планируется раз и только раз.
- ◇ Простейший подход к планированию базового блока включает посещение каждого узла графа в «приоритетном топологическом порядке»
- ◇ Граф зависимостей по данным будучи ориентированным ациклическим графом имеет минимум один, а на самом деле — несколько топологических порядков. Наиболее удобный из них называется «приоритетным топологическим порядком»

## **15.5 Планирование базовых блоков**

### **15.5.3 Алгоритм планирования списка**

- ◇ Приоритетные топологические порядки:
  - ◇ При отсутствии ограничений, связанных с ресурсами, кратчайший план получается при помощи критического пути – самого длинного пути в графе зависимости данных. Метрика, используемая в качестве функции приоритета, высота узла, представляющая собой длину самого длинного пути от данного узла в графе.
  - ◇ Если все операции независимы (между ними нет зависимостей по данным и/или по управлению), то длина плана ограничена доступными ресурсами. Критический ресурс – это ресурс с наибольшим отношением количества использований к количеству доступных единиц ресурса. Более высокий приоритет будут иметь операции, использующие более критические ресурсы.
  - ◇ Для разрешения неопределенностей можно использовать исходный порядок операций – операция, находящаяся в исходной программе раньше других, должна быть запланирована раньше..

## **15.5 Планирование базовых блоков**

### **15.5.3 Алгоритм планирования списка**

- ◇ Алгоритм обходит узлы графа зависимостей по данным в «приоритетном топологическом порядке» и для каждого узла вычисляет наиболее раннее время, когда он может быть вычислен, в соответствии с ограничениями зависимости данных ранее спланированных узлов.
- ◇ Затем выполняется проверка наличия требующихся для узла ресурсов в таблице резервирования ресурсов, в которой собраны все выделенные к настоящему времени ресурсы, и план корректируется: команда планируется к выполнению в самый ранний момент, в который имеется достаточно ресурсов.

## 15.5 Планирование базовых блоков

### 15.5.3 Алгоритм планирования списка

**Вход:** вектор ресурсов машины  $R = [r_1, r_2, \dots]$ , граф зависимостей по данным  $G = (N, E)$ ; каждой операции  $n \in N$  сопоставлена таблица резервирования ресурсов  $RT_n$ ; каждое ребро  $e = n_1 \rightarrow n_2$  из  $E$  помечено задержкой  $d_e$ :  $n_2$  можно выполнить не ранее чем через  $d_e$  тактов после  $n_1$ .

**Выход:** план  $S$ , который отображает операции из  $N$  на временны́е интервалы, когда удовлетворяются все ограничения по ресурсам для этих операций и когда может начинаться их выполнение.

**Метод:** выполнить программу (псевдокод):

## 15.5 Планирование базовых блоков

### 15.5.3 Алгоритм планирования списка

```
RT = пустая таблица резервирования ресурсов;
for (каждый $n \in N$ в приоритетном
топологическом порядке) {
 /* Вычисление самого раннего времени $S(n) = s$
 возможного начала команды n на основе заданного
 $S(p)$ – времени начала предыдущей команды p */
 $s = \max_{(e = p \rightarrow n) \in E} (S(p) + d_e);$
 /* Вычисление задержки начала выполнения команды n
 до тех пор, пока не будут доступны все ресурсы. */
 while ($\exists i: RT[s + i] + RT_n[i] > R[i]$)
 $s = s + 1;$
 $S(n) = s;$
 /* Корректировка таблицы ресурсов для учета
 ресурсов, используемых командой n */
 for (всех ресурсов i)
 $RT[s + i] = RT[s + i] + RT_n[i]$
```

# 15.5 Планирование базовых блоков

## 15.5.3 Алгоритм планирования списка

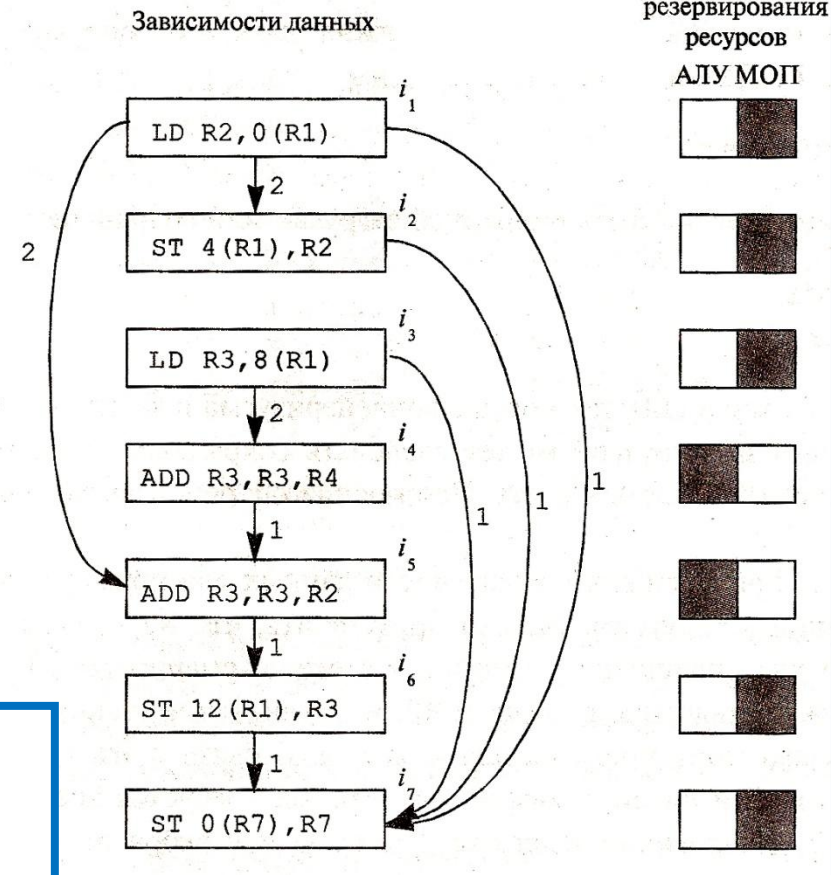
### Пример

- ◇ В графе зависимости данных (справа) критический путь имеет длину 6 тактов: путь от LD R3,8 (R1) до ST 0 (R1) ,R7. Суммарные задержки вдоль ребер равны 5, и к ним добавляется еще один такт, необходимый для выполнения команды ST 0 (R1) ,R7. Оптимальный план, выданный алгоритмом показан на нижнем рисунке

|               |                |
|---------------|----------------|
|               | LD R3,8 (R1)   |
|               | LD R2,0 (R1)   |
| ADD R3, R3,R4 |                |
| ADD R3, R3,R2 | ST 4 (R1) ,R2  |
|               | ST 12 (R1) ,R3 |
|               | ST 0 (R7) ,R7  |

АЛУ МОП

|  |  |
|--|--|
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |





## 15.6 Глобальное планирование кода

### 15.6.1 Вводные замечания

- ◇ Для современных процессоров планы, создаваемые путем работы только с базовыми блоками, как правило, несовершенны и приводят к простаиванию большого количества ресурсов. Для более эффективного использования машинных ресурсов можно рассмотреть стратегии планирования, использующие перемещение команд между базовыми блоками. Стратегии планирования кода, которые затрагивают больше одного базового блока, называются стратегиями *глобального планирования*. Стратегии глобального планирования должны гарантировать:
  - ◇ оптимизированная программа выполняет все команды исходной программы;
  - ◇ при выполнении некоторых команд исходной программы с упреждением не возникает никаких нежелательных побочных действий.

## 15.6 Глобальное планирование кода

### 15.6.2 Пример глобального планирования



#### Процессор



может выполнить любые две команды за один такт



каждая команда (кроме команды загрузки **LD**)

выполняется с задержкой в один такт



команда загрузки выполняется с задержкой в два такта

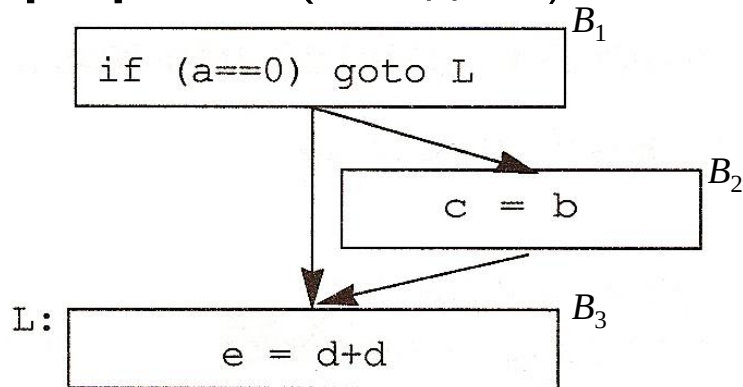
## 15.6 Глобальное планирование кода

### 15.6.2 Пример глобального планирования

#### ◇ Процессор

- ◇ может выполнить любые две команды за один такт
- ◇ каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
- ◇ команда загрузки выполняется с задержкой в два такта

#### ◇ Программа (исходная):



пусть все обращения к памяти  
корректны и удовлетворяются  
кэшем.

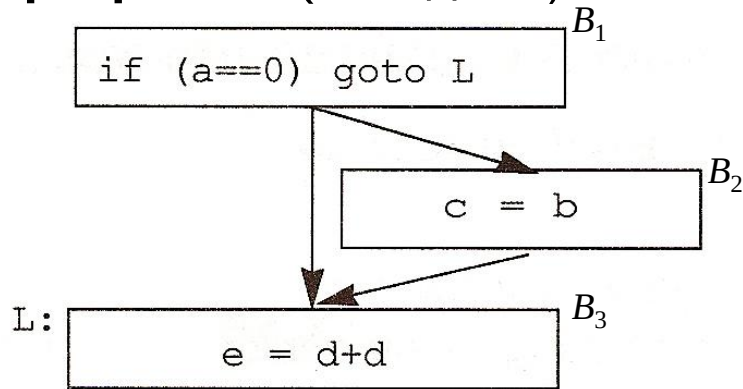
# 15.6 Глобальное планирование кода

## 15.6.2 Пример глобального планирования

### ◆ Процессор

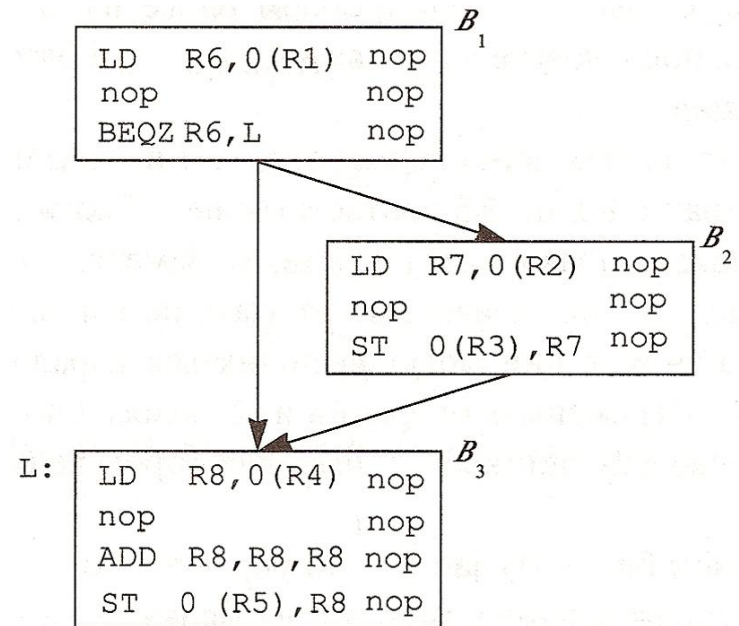
- ◆ может выполнить любые две команды за один такт
- ◆ каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
- ◆ команда загрузки выполняется с задержкой в два такта

### ◆ Программа (исходная):



пусть все обращения к памяти  
корректны и удовлетворяются  
кэшем.

Та же программа в машинном коде  
после локального планирования



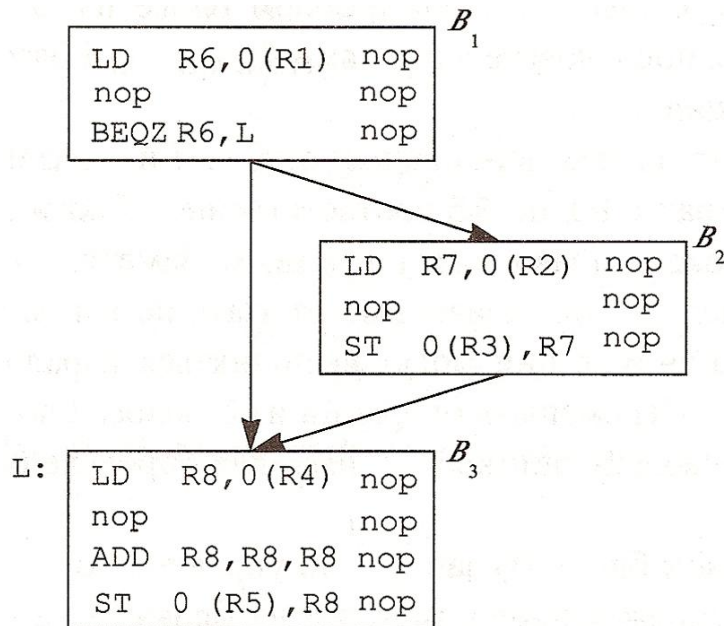
# 15.6 Глобальное планирование кода

## 15.6.2 Пример глобального планирования

### ◆ Процессор

- ◆ может выполнить любые две команды за один такт
- ◆ каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
- ◆ команда загрузки выполняется с задержкой в два такта

### Программа в машинном коде после локального планирования



Команды в каждом блоке из-за зависимостей по данным должны выполняться последовательно, поэтому при локальном планировании в каждую команду была вставлена операция **nop**.

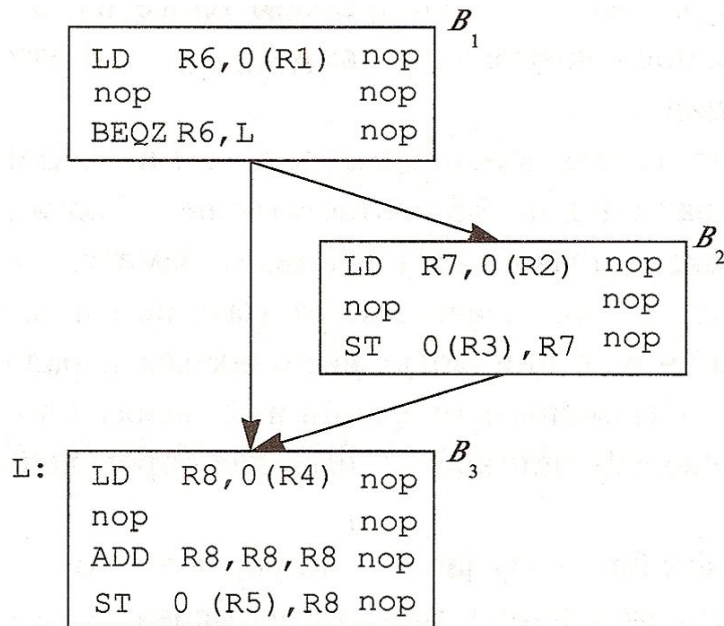
# 15.6 Глобальное планирование кода

## 15.6.2 Пример глобального планирования

### ◇ Процессор

- ◇ может выполнить любые две команды за один такт
- ◇ каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
- ◇ команда загрузки выполняется с задержкой в два такта

### Программа в машинном коде после локального планирования



Команды в каждом блоке из-за зависимостей по данным должны выполняться последовательно, поэтому при локальном планировании в каждую команду была вставлена операция **nop**.

Пусть переменные  $a$ ,  $b$ ,  $c$ ,  $d$  и  $e$  расположены в разных ячейках, а их адреса находятся на регистрах с  $R1$  по  $R5$  соответственно. Тогда у вычислений в разных базовых блоках нет зависимостей по данным, но есть зависимости по управлению.

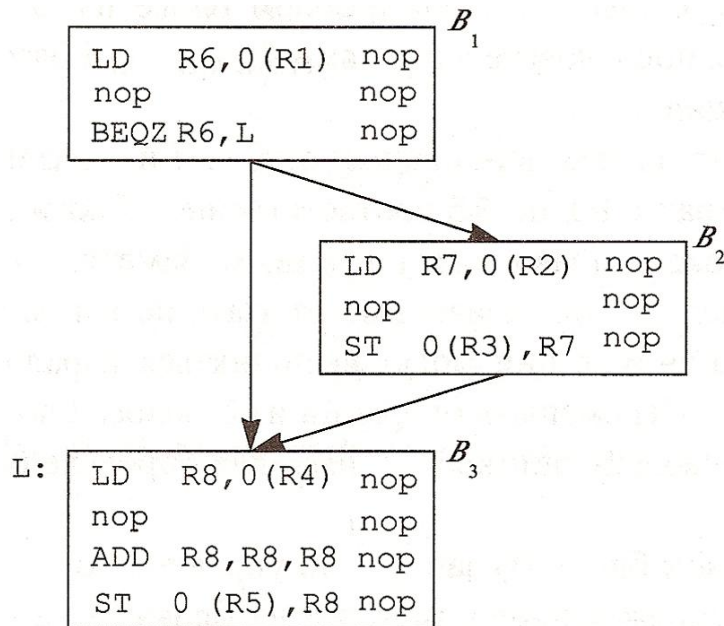
# 15.6 Глобальное планирование кода

## 15.6.2 Пример глобального планирования

### ◇ Процессор

- ◇ может выполнить любые две команды за один такт
- ◇ каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
- ◇ команда загрузки выполняется с задержкой в два такта

### Программа в машинном коде после локального планирования



Команды в каждом блоке из-за зависимостей по данным должны выполняться последовательно, поэтому при локальном планировании в каждую команду была вставлена операция **nop**.

Пусть переменные  $a$ ,  $b$ ,  $c$ ,  $d$  и  $e$  расположены в разных ячейках, а их адреса находятся на регистрах с  $R1$  по  $R5$  соответственно. Тогда у вычислений в разных базовых блоках нет зависимостей по данным, но есть зависимости по 44 управлению.

## 15.6 Глобальное планирование кода

### 15.6.2 Пример глобального планирования

#### ◇ Процессор

- ◇ может выполнить любые две команды за один такт
- ◇ каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
- ◇ команда загрузки выполняется с задержкой в два такта

#### ◇ Глобальное планирование:

- ◇ Все команды в блоке  $B_3$  выполняются независимо от ветвления. Значит, они могут выполняться параллельно с командами из блока  $B_1$ .
- ◇ Перемещать команды из блока  $B_1$  в блок  $B_3$  нельзя из-за зависимостей по управлению: они необходимы для выбора пути ветвления.
- ◇ Команды в блоке  $B_2$  зависят по управлению от проверки в блоке  $B_1$ . Можно выполнить загрузку из блока  $B_2$  **с упреждением**, перенеся ее в блок  $B_1$  (на этом экономится 2 такта).
- ◇ сохранения выполнять с упреждением нельзя, так как необходимость сохранений выясняется только после ветвления, однако команды сохранения можно отложить (перенести сохранение  $s$  из  $B_2$  в копию  $B_3$ ).

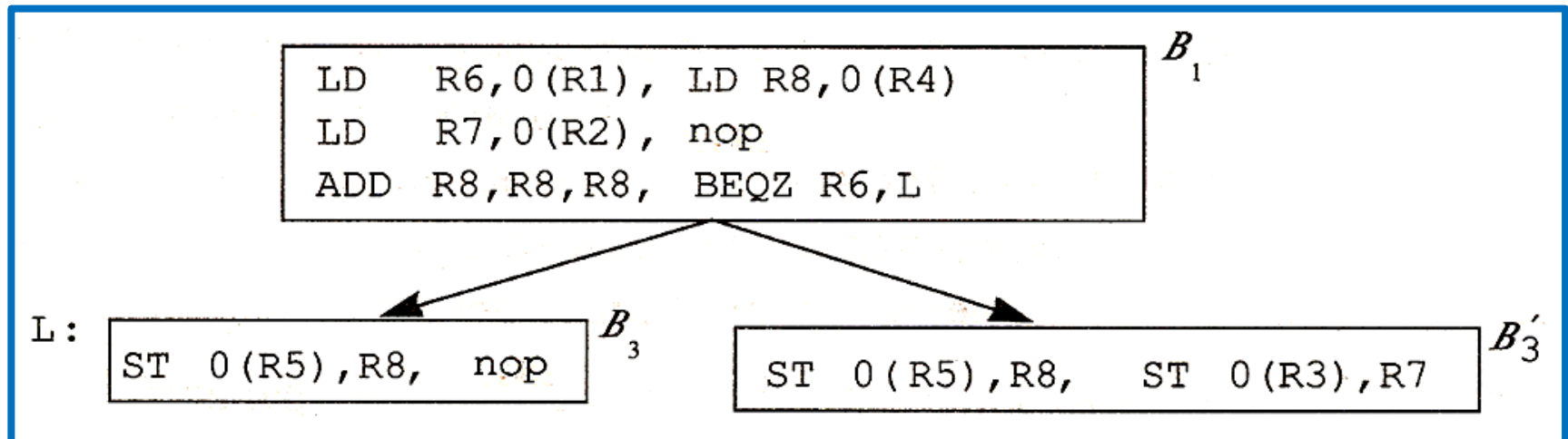


## 15.4 Глобальное планирование кода

### 15.4.2 Пример глобального планирования

◇ Глобальное планирование:

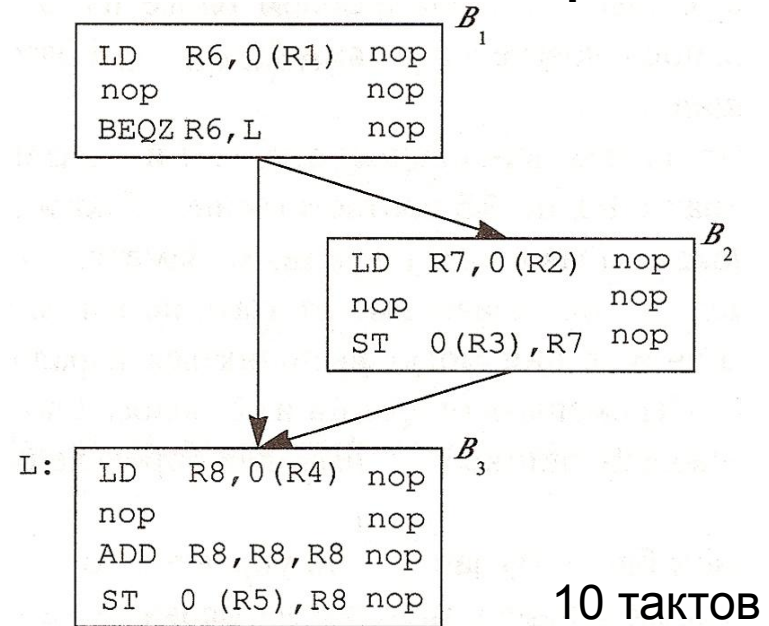
- ◇ Все команды в блоке  $B_3$  выполняются независимо от ветвления. Значит, они могут выполняться параллельно с командами из блока  $B_1$ .
- ◇ Перемещать команды из блока  $B_1$  в блок  $B_3$  нельзя из-за зависимостей по управлению: они необходимы для выбора пути ветвления.
- ◇ Команды в блоке  $B_2$  зависят по управлению от проверки в блоке  $B_1$ . Можно выполнить загрузку из блока  $B_2$  **с упреждением**, перенеся ее в блок  $B_1$  (на этом экономится 2 такта).
- ◇ сохранения выполнять с упреждением нельзя, так как необходимость сохранений выясняется только после ветвления, однако команды сохранения можно отложить (перенести сохранение  $c$  из  $B_2$  в копию  $B_3$ ).



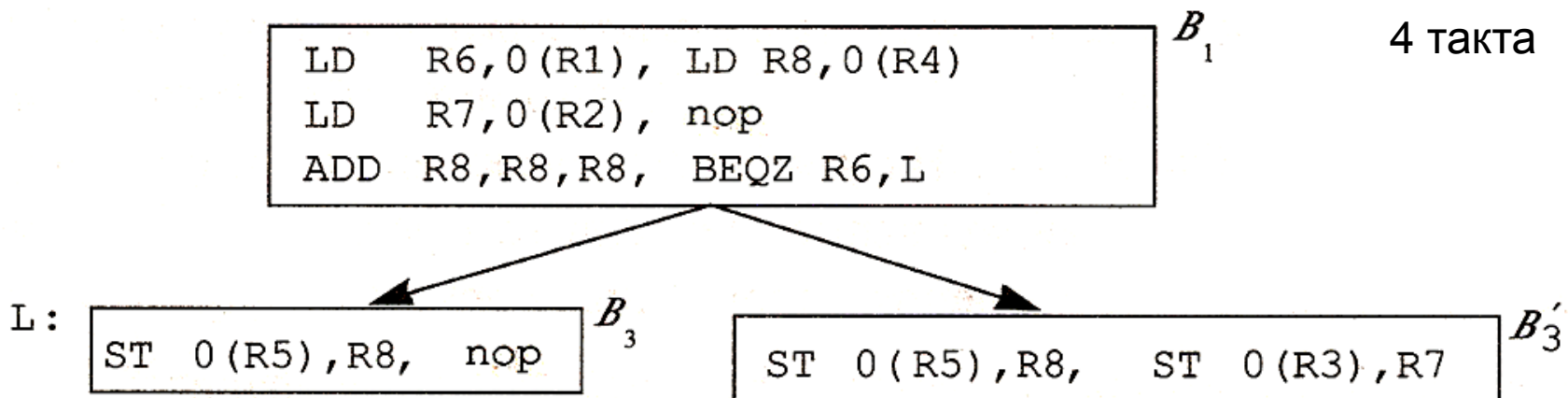
## 15.6 Глобальное планирование кода

### 15.6.2 Пример глобального планирования

#### После локального планирования



#### После глобального планирования



## 15.6 Глобальное планирование кода

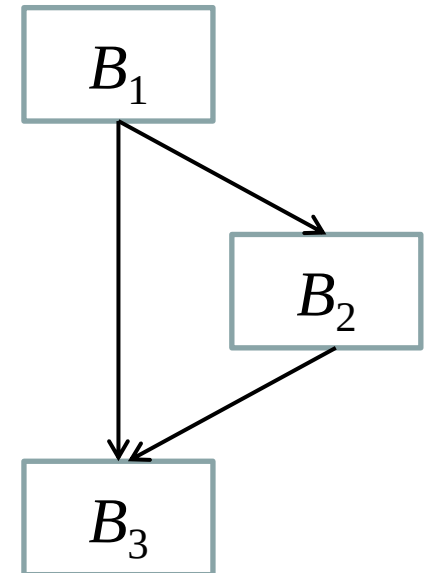
### 15.6.3 Эквивалентность по управлению

- ◇ **Определение.** Базовые блоки  $B$  и  $B'$  эквивалентны по управлению, если блок  $B$  выполняется тогда и только тогда, когда выполняется блок  $B'$ .
- ◇ **Утверждение.** Для того, чтобы базовые блоки  $B$  и  $B'$  были эквивалентны по управлению необходимо и достаточно, чтобы  $B = Dom(B') \ \& \ B' = PostDom(B)$
- ◇ **Пример.** В примере 3.4.2 блоки  $B_1$  и  $B_3$  эквивалентны по управлению.

## 15.6 Глобальное планирование кода

### 15.6.4 Перемещение кода

- ◇ Глобальное планирование кода между базовыми блоками.  
При этом, как видно из рассмотренного примера 15.4.2 возможны следующие отношения между блоками:
  - ◇ Блоки  $B_1$  и  $B_3$  эквивалентны по управлению:  
$$B_1 = Dom(B_3) \ \& \ B_3 = Postdom(B_1)$$
  - ◇ Блок  $B_1$  доминирует над блоком  $B_2$ ,  
а блок  $B_2$  не постдоминирует над блоком  $B_1$ :  
$$B_1 = Dom(B_2) \ \& \ B_2 \neq Postdom(B_1)$$
  - ◇ Блок  $B_2$  не доминирует над блоком  $B_3$ ,  
блок  $B_3$  постдоминирует над блоком  $B_2$ :  
$$B_2 \neq Dom(B_3) \ \& \ B_3 = Postdom(B_2)$$
  - ◇ Пара блоков вдоль пути выполнения не связана ни отношением доминирования, ни отношением постдоминирования.



## 15.6 Глобальное планирование кода

### 15.6.5 Перемещение кода вверх по пути управления

- ◇ Пусть команда из блока  $src$  перемещается вверх по пути управления в блок  $dst$ , не нарушая никаких зависимостей по данным.
  - ◇  $dst = Dom(src) \ \& \ src = Postdom(dst)$ :  
перемещенная команда выполняется только один раз
  - ◇  $dst \neq Dom(src) \ \& \ src = Postdom(dst)$ :  
существует один или несколько путей, которые достигают  $src$ , но не проходят через  $dst$ ; в этом случае в базовые блоки, которые образуют разрез, отделяющий  $Entry$  от  $src$  необходимо вставить *компенсирующий код*: копии перемещаемой команды (чтобы она выполнялась на всех путях, достигающих  $src$ ).
  - ◇  $dst = Dom(src) \ \& \ src \neq Postdom(dst)$ :  
существует один или несколько путей, которые проходят через  $dst$ , но не достигают  $src$ , и в результате на этих путях могут быть выполнены лишние команды; эти команды не должны приводить к нежелательным побочным эффектам, иначе перемещение кода **некорректно**

## 15.6 Глобальное планирование кода

### 15.6.5 Перемещение кода вверх по пути управления

- ◇ Перемещение кода повышает эффективность программы только тогда, когда оно использует только те ресурсы, которые иначе простаивали бы; такое перемещение называется *бесплатным*
- ◇ В каждом месте разреза, в которое вставляется команда, должны удовлетворяться следующие ограничения:
  - ◇ операнды вставленной команды должны иметь те же значения, что и в исходной команде.
  - ◇ результат выполнения команды не должен убивать значения, которое потребуется в дальнейшем.
  - ◇ результат команды нельзя убивать до достижения *src*.
- ◇ Компенсирующий код потенциально в состоянии сделать некоторые из путей более медленными, так что перемещение кода повышает производительность программы, только тогда, когда оптимизированные пути выполняются более часто, чем неоптимизированные

## 15.6 Глобальное планирование кода

### 15.6.6 Перемещение кода вниз по пути управления

- ◇ Пусть команда из блока  $src$  перемещается вниз по пути управления в блок  $dst$ , не нарушая никаких зависимостей по данным.
- ◇  $src = Dom(dst) \ \& \ dst = Postdom(src)$ :  
перемещенная команда выполняется только один раз
- ◇  $src \neq Dom(dst)$ :  
существует хотя бы один путь, который достигает  $dst$ , минуя  $src$ ; в этом случае будут выполняться дополнительные операции.
- ◇  $dst \neq Postdom(src)$ :  
существует хотя бы один путь, который проходит через  $src$ , но не достигает  $dst$ ; в этом случае в базовые блоки, которые образуют разрез, отделяющий  $src$  от  $dst$  необходимо вставить *компенсирующий код*: копии перемещаемой команды (чтобы она выполнялась на всех путях от  $src$  до  $dst$ ).

**Замечание.** Такой разрез размещается ниже блока  $src$ .

## 15.6 Глобальное планирование кода

### 15.6.7 Резюме по восходящему и нисходящему перемещениям кода

- ◇ Перемещение команд между эквивалентными по управлению базовыми блоками – простейшее и наиболее эффективное: не требуется ни дополнительных команд, ни компенсирующего кода.
- ◇ Если при перемещении кода вверх  $src \neq Postdom(dst)$  (либо при перемещении кода вниз  $src \neq Dom(dst)$ ), может потребоваться *выполнение дополнительных команд*. Поэтому такое перемещение кода выгодно в том случае, когда дополнительные операции могут быть выполнены бесплатно (с использованием только тех ресурсов, которые иначе простаивали бы), и когда выполняется путь, проходящий через  $src$ .
- ◇ Если при перемещении кода вверх  $dst \neq Dom(src)$  (либо при перемещении кода вниз  $dst \neq Postdom(src)$ ), требуется *компенсирующий код*. Пути с компенсирующим кодом могут оказаться замедленными, поэтому важно, чтобы оптимизированные пути выполнялись чаще, чем неоптимизированные.



## 15.4 Глобальное планирование кода

### 15.4.7 Резюме по восходящему и нисходящему перемещениям кода

- ◇ Случай, когда при перемещении кода вверх
$$src \neq Postdom(dst) \ \& \ dst \neq Dom(src)$$
(либо при перемещении кода вниз
$$src \neq Dom(dst) \ \& \ dst \neq Postdom(src)),$$
объединяет недостатки двух предыдущих случаев:  
требуется компенсирующий код и может потребоваться выполнение дополнительных операций.  
Поэтому в этом случае перемещение кода не производится

## 15.6 Глобальное планирование кода

### 15.6.7 Резюме по восходящему и нисходящему перемещениям кода

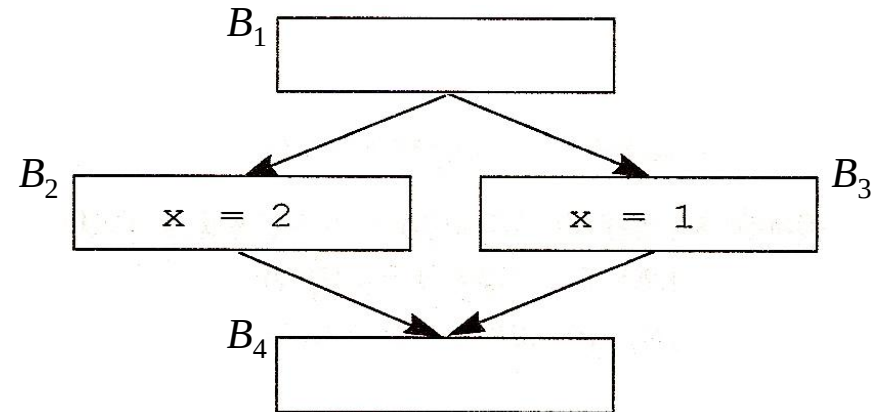
- ◇ Различные варианты перемещения кода можно свести в следующую таблицу

| $\uparrow$   | $src = Postdom(dst)$ | $dst = Dom(src)$     | Дублирование<br>кода | Компенсирующий код |
|--------------|----------------------|----------------------|----------------------|--------------------|
| $\downarrow$ | $src = Dom(dst)$     | $dst = Postdom(src)$ |                      |                    |
|              | +                    | +                    | −                    | −                  |
|              | −                    | +                    | +                    | −                  |
|              | +                    | −                    | −                    | +                  |
|              | −                    | −                    | +                    | +                  |

## 15.6 Глобальное планирование кода

### 15.6.8 Поучительный пример

◇ В ГПУ на рисунке в блок  $B_1$  можно перенести и присваивание  $x = 2$ , и присваивание  $x = 1$ , так как при переносе сохраняются все зависимости исходной программы.



◇ Однако после того, как одно из присваиваний будет перенесено в  $B_1$ , второе присваивание переносить будет нельзя: до перемещения переменная  $x$  на выходе из блока  $B_1$  не является живой, а после перемещения она становится живой.

**Если в некоторой точке программы переменная жива, то упреждающее определение этой переменной не может быть перемещено выше данной точки.**

◇ Пример показывает, что перемещение кода может изменить отношения зависимостей по данным между командами.

◇ Таким образом, после каждого перемещения кода следует выполнять обновление зависимостей по данным

## ***15.7 Алгоритмы глобального планирования***

### **15.7.1 Вводные замечания**

- ◇ Перемещение кода повышает эффективность некоторых путей за счет снижения эффективности остальных путей.
- ◇ Считается, что обычно 90% времени работы программы тратится на выполнение 10% ее кода.
- ◇ Цель как можно более ускорить часто выполнимые пути, за счет того, что более редкие пути замедлятся.

# 15.7 Алгоритмы глобального планирования

## 15.7.2 Оценка частоты выполнения путей

- ◇ Разумные соображения типа:
  - ◇ «команды во внутренних циклах выполняются гораздо чаще, чем код внешних циклов»
  - ◇ «при ветвлении чаще выбираются пути, ведущие назад, чем вперед»
  - ◇ «крайне редко отрабатываются ветвления, ведущие к выходу из программы или в подпрограммы обработки исключений»  
и т.п.
- ◇ Динамическое профилирование:
  - ◇ Программа *инструментируется*: в нее добавляется код для подсчета частоты выполнения ветвей
  - ◇ Выполняется тестовый запуск программы с «типичными входными данными».
  - ◇ Результаты, полученные при помощи этого метода, обычно оказываются достаточно точными. Результаты профилирования могут использоваться компилятором и для других оптимизаций.

## **15.7 Алгоритмы глобального планирования**

### **15.7.3 Алгоритм глобального планирования на основе областей**

- ◇ Рассмотрим планировщик, который поддерживает два вида перемещения кода:
  - ◇ перемещение команд вверх в блоки, эквивалентные с точки зрения управления;
  - ◇ упреждающее перемещение команд по одной из ветвей в предшествующий доминирующий блок.
- ◇ Алгоритм планирования использует области  
Напоминание:
  - ◇ область – это подмножество базовых блоков ГПУ, которые могут быть достигнуты через один входной базовый блок
  - ◇ существует три типа областей:
    - ◆ область-лист
    - ◆ область-тело
    - ◆ область-цикл
  - ◇ любая процедура может быть представлена как иерархия областей

## **15.7 Алгоритмы глобального планирования**

### **15.7.3 Алгоритм глобального планирования на основе областей**

- ◇ Порядок обхода графа потока:
  - ◇ Все обратные ребра удаляются – граф потока становится ациклическим
  - ◇ Все зависимости, идущие к заголовку области, удаляются из графа зависимостей – граф зависимостей становится ациклическим
  - ◇ Планирование кода ведется от внутренних областей к внешним: сначала планируется код в областях-листьях (базовых блоках), потом в областях-телах первого уровня, потом в областях-телах второго уровня и т.д.
  - ◇ При планировании области каждая вложенная подобласть рассматривается как черный ящик: команды не могут перемещаться в подобласть или из нее; однако команды могут перемещаться вокруг подобласти при условии удовлетворения ограничений зависимостей по данным и по управлению

## **15.7 Алгоритмы глобального планирования**

### **15.7.3 Алгоритм глобального планирования на основе областей**

- ◇ Порядок обхода графа потока (окончание):
  - ◇ Базовые блоки в каждой области посещаются в топологическом порядке. Такое упорядочение гарантирует, что базовый блок не будет планироваться до тех пор, пока не будут спланированы все команды, от которых он зависит.
  - ◇ Команды, которые должны быть спланированы в базовом блоке  $B$ , выводятся из всех блоков, эквивалентных по управлению блоку  $B$  (включая сам базовый блок  $B$ ), а также из непосредственных преемников блока  $B$ , над которыми он доминирует



## 15.7 Алгоритмы глобального планирования

### 15.7.3 Алгоритм глобального планирования на основе областей

- ◇ Планирование внутри каждого базового блока ведется с помощью алгоритма планирования списка, который поддерживает приоритетный список *CandInsts*, содержащий команды блоков-кандидатов, все предшественники которых уже спланированы
  - ◇ В списке *CandInsts* используется функция приоритета, подобная функции приоритета для планирования списка, но с одним изменением – она обеспечивает выполнение команд из блоков-преемников блока только с упреждением: командам блоков, эквивалентных по управлению блоку  $B$ , назначается более высокий приоритет, чем командам блоков-преемников.
  - ◇ План создается такт за тактом.
  - ◇ Для каждого такта в порядке приоритета проверяется каждая команда  $n$  из списка *CandInsts* и включается в план, если позволяют ресурсы (функция  $S(n)$ ).
  - ◇ Затем обновляется список *CandInsts*
  - ◇ Процесс повторяется до тех пор, пока не будут спланированы все команды из базового блока  $B$ .

## 15.7 Алгоритмы глобального планирования

### 15.7.3 Алгоритм глобального планирования на основе областей

- ◇ Формирование списка *CandInsts* следующим образом:
  - ◇ Функция *ControlEquiv*( $B$ ) строит множество блоков, эквивалентных блоку  $B$  по управлению
  - ◇ Функция *DominatedSucc*, примененная к множеству базовых блоков, строит множество базовых блоков, являющихся преемниками как минимум одного блока множества, для которых все блоки множества являются доминирующими
- ◇ Построение множества *CandBlocks*:  
оно состоит из множества блоков, эквивалентных рассматриваемому блоку  $B$  по управлению, к которому добавлены доминируемые последователи  
В список *CandInsts* включаются все уже спланированные команды из множества *CandBlocks*
- ◇ Формирование плана с помощью функции  $S(n, B, t)$  – включение команды  $n$  из  $B$  блока в план на такт  $t$ .

## ***15.7 Алгоритмы глобального планирования***

### **15.7.3 Алгоритм глобального планирования на основе областей**

- ◇ **Вход:** граф потока управления и описание машинных ресурсов.
- ◇ **Выход:** план  $S$ , отображающий каждую команду на базовый блок и интервал времени.
- ◇ **Метод:** выполнить следующую программу:

## 15.7 Алгоритмы глобального планирования

### 15.7.3 Алгоритм глобального планирования на основе областей

```
for (каждая область R в топологическом порядке, при котором
 внутренние области обрабатываются раньше внешних) {
 Вычисление зависимостей данных;
 for (каждый базовый блок B области R в приоритетном
 топологическом порядке) {
 $CandBlocks = ControlEquiv(B) \cup DominatedSucc (ControlEquiv (B))$;
 $CandInsts$ = готовые команды из $CandBlocks$;
 for ($t = 0, 1, \dots$, пока не будут спланированы все
 команды из B) {
 for (каждая команда n из $CandInsts$ в приоритетном порядке)
 if (n не имеет конфликтов ресурсов в момент t) {
 $S(n, B, t)$;
 Обновление имеющихся ресурсов;
 Обновление зависимостей данных;
 }
 Обновление $CandInsts$;
 }
 }
}
```

## ***15.7 Алгоритмы глобального планирования***

### **15.7.4 Развертка циклов**

- ◇ Планирование на основе областей ведется в рамках одной итерации цикла (удаление обратных ребер), так что граница итерации цикла является барьером для перемещения кода.  
Простой, но эффективный метод упрощения этой проблемы состоит в частичной развертке цикла перед планированием кода.
- ◇ Развертывание создает в теле цикла большое количество команд, позволяя алгоритму глобального планирования достичь большей степени параллелизма.

## 15.7 Алгоритмы глобального планирования

### 15.7.4 Развертка циклов

◇ Пример 1. Цикл `for` вида:

```
for (i = 0; i < N; i++) {
 S(i) ;
}
```

можно переписать следующим образом:

```
for (i = 0; i+4 < N; i+=4) {
 S(i) ;
 S(i+1) ;
 S(i+2) ;
 S(i+3) ;
}
for (; i < N;) {
 S (i) ;
}
```

## ***15.7 Алгоритмы глобального планирования***

### **15.7.4 Развертка циклов**

◇ Пример 2. Цикл **repeat**

```
repeat
 S;
until C;
```

можно переписать следующим образом:

```
repeat {
 S;
 if (C) break;
 S;
 if (C) break;
 S;
 if (C) break;
 S;
} until C;
```

## 15.7 Алгоритмы глобального планирования

### 15.7.5 Планирование с добавлением компенсирующего кода

- ◇ Алгоритм 15.5.3 поддерживает только два вида перемещения кода:
  - ◇  $B_1 = Dom(B_2) \ \& \ B_2 = Postdom(B_1)$
  - ◇  $B_1 = Dom(B_2) \ \& \ B_2 \neq Postdom(B_1)$
- ◇ Однако может оказаться полезным перемещение, которое требует добавления компенсирующего кода, т.е. случай  $B_1 \neq Dom(B_2) \ \& \ B_2 = Postdom(B_1)$

Реализовать такое перемещение кода можно следующим образом:

- ◇ просматриваются все пары базовых блоков, выполняющиеся один за другим, и для каждой пары проверяется, нельзя ли переместить какую-либо команду между блоками пары для снижения общего времени выполнения этих блоков;
- ◇ когда требуемая пара блоков найдена, проверяется, не надо ли дублировать перемещаемую команду на других путях (компенсирующий код);
- ◇ перемещение выполняется только в том случае, когда оно дает выигрыш времени выполнения.



## ***15.7 Алгоритмы глобального планирования***

### **15.7.5 Планирование с добавлением компенсирующего кода**

- ◇ Предложенное простое расширение алгоритма может оказаться эффективным для повышения производительности циклов:

В частности, оно может переместить команду из начала каждой итерации цикла, кроме первой, в конец предыдущей, а соответствующую команду из первой итерации вынести за пределы цикла. Такая оптимизация особенно полезна для небольших циклов, итерации которых состоят всего лишь из нескольких команд.

- ◇ Возможности этого метода ограничены, так как каждое решение о перемещении кода принимается локально и независимо от других.

## 15.7 Алгоритмы глобального планирования

### 15.7.6 Агрессивные алгоритмы перемещения кода

- ◇ Если целевой процессор статически имеет большие возможности по обеспечению параллелизма на уровне команд (например, *Intel Itanium*), может потребоваться более агрессивный алгоритм планирования кода.
- ◇ Агрессивный алгоритм можно получить путем усовершенствования алгоритма 15.5.3 в следующих направлениях:
  - ◇ Добавить новые базовые блоки в критические ребра графа потока управления  
**(Напоминание.** Критическим называется ребро, идущее от узла *Src* с более чем одним преемником к узлу *Dst* с более чем одним предшественником:  
 $|Succ(Src)| > 1 \ \& \ |Pred(Dst)| > 1$ )  
Если в конце планирования кода добавленные блоки окажутся пустыми, они будут устранены.  
Добавление базовых блоков в критические ребра существенно упрощает структуру графа потока управления и облегчает его анализ.

## ***15.7 Алгоритмы глобального планирования***

### **15.7.6 Агрессивные алгоритмы перемещения кода**

- ◇ Агрессивный алгоритм можно получить путем усовершенствования алгоритма 15.5.3 в следующих направлениях:
  - ◇ Одна из полезных эвристик состоит в перемещении команд из почти пустых базовых блоков, чтобы сделать их полностью пустыми и исключить из ГПУ.
  - ◇ Код, который будет выполняться в базовом блоке  $B$ , планируется раз и навсегда при посещении блока  $B$ . Такого простого подхода достаточно, так как алгоритм может перемещать операции только вверх в доминирующие блоки.

## **15.7 Алгоритмы глобального планирования**

### **15.7.7 Взаимодействие с динамическим планировщиком**

- ◇ Динамический планировщик – это, например, аппаратный планировщик *OoO*-процессора
- ◇ Динамический планировщик обладает тем преимуществом, что он может создавать новые планы в зависимости от условий времени выполнения, а не рассматривать заблаговременно все возможные планы.
- ◇ Если целевой процессор оснащен динамическим планировщиком (принадлежит классу *OoO*), основная функция статического планировщика состоит в обеспечении ранней выборки команд с большими задержками, чтобы динамический планировщик мог запланировать как можно более раннее выполнение этих команд. Это особенно эффективно в том случае, когда в целевом процессоре имеются команды предвыборки данных.
- ◇ В частности, предвыборка данных помогает предотвратить промахи кэша, представляющие собой класс непредсказуемых событий, которые могут привести к большим разбросам производительности программы.

## 15.7 Алгоритмы глобального планирования

### 15.7.7 Взаимодействие с динамическим планировщиком

- ◇ Если целевой процессор не оснащен динамическим планировщиком (не принадлежит классу *OoO*), статический планировщик должен быть консервативен и разделять все пары команд с зависимостями по данным минимальной задержкой (вставляя *nop*).  
Для достижения наилучшей производительности компилятор должен назначать большие задержки наиболее вероятным зависимостям и небольшие – маловероятным.
- ◇ Часто потеря производительности связана с неверным предсказанием ветвлений.

# **16. Программная конвейеризация**

# 16.1 Что такое программная конвейеризация

## 16.1.1 Постановка задачи

- ◇ При планировании на основе областей граница итерации цикла является барьером для перемещения кода, так как, вообще говоря, операции из одной итерации не могут перекрываться операциями из другой. Развертка цикла является способом «загнать» несколько итераций в одну и тем самым расширить область планирования.
- ◇ **Пример.** Итерации цикла:

```
for (i = 0; i < n; i++)
```

```
 F[i] = A[i] * C[i] + b;
```

выполняют запись в разные ячейки памяти, не совпадающие с ячейками, из которых читаются данные.

*Следовательно, между итерациями этого цикла не имеется никаких зависимостей по данным, и все его итерации могут быть выполнены в произвольном порядке, а значит их можно выполнить параллельно.*

# 16.1 Что такое программная конвейеризация

## 16.1.1 Постановка задачи

- ◇ Циклы, итерации которых независимы одна от другой называются *универсальными*.
- ◇ Пример универсального цикла приведен на предыдущем слайде.
- ◇ Универсальные циклы характерны для многих численных приложений (речь идет о научно-технических расчетах).
- ◇ **Цель** – разработать методы планирования кода, которые позволят в результате планирования процедур, содержащих универсальные циклы, получать достаточно компактный код, который будет как можно более полно использовать возможности компьютера по параллельному выполнению команд, или (для компьютеров классов *VLIW* и *EPIC*) операций (частей команд).



# 16.1 Что такое программная конвейеризация

## 16.1.2 Модель процессора

- ◇ На каждом такте МОДЕЛЬНЫЙ КОМПЬЮТЕР может выполнить:
  - ◇ одну арифметическую операцию (**A**),
  - ◇ одну операцию загрузки на регистр (**LD**),
  - ◇ одну операцию сохранения регистра (**ST**) и
  - ◇ одну операцию перехода (**B**).
- ◇ Арифметические операции выполняются в конвейерном режиме. Они могут быть инициированы на любом такте, но их результаты становятся доступны **два такта спустя**.
- ◇ Задержка всех прочих команд – **один такт**.



**Замечание.** Модельный компьютер введен лишь для того, чтобы можно было на простых и достаточно компактных примерах продемонстрировать программную конвейеризацию циклов.

# 16.1 Что такое программная конвейеризация

## 16.1.2 Модель процессора

- ◇ Цикл в модельном компьютере может организоваться с помощью специальной операции перехода

**BL R, L**

которая

- ◇ уменьшает на единицу целочисленное значение регистра **R** (автодекрементный режим)
- ◇ пока значение регистра **R** не равно 0, передает управление на метку **L**.
- ◇ Операции с памятью могут выполняться в **автоинкрементном режиме**: если в команде после ссылки на регистр помещены символы **++**, то после ее выполнения значение регистра **R** увеличивается таким образом, что он указывает на следующее значение соответствующего типа (следующий элемент массива).

# 16.1 Что такое программная конвейеризация

## 16.1.2 Модель процессора

- ◇ Цикл организуется с помощью специальной операции перехода

**BL R, L**

которая

^

На регистре **R** находится текущее значение переменной цикла **i**, которая изменяется от **i<sub>max</sub>** до **0**, т.е. шаг цикла равен **-1** (цикл выполняется «с конца»).

на метку **L**.

- ◇ Операции с памятью могут выполняться в **автоинкрементном режиме**: если в команде после ссылки на регистр помещены символы **++**, то после ее выполнения значение регистра **R** увеличивается таким образом, что он указывает на следующее значение соответствующего типа (следующий элемент массива).

## 16.1 Что такое программная конвейеризация

### 16.1.3 Последовательное выполнение итераций цикла

```
for (i = 0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Если итерации выполняются последовательно, то наилучший план для модельного компьютера имеет вид

```
// R1, R2, R3 = &A, &C, &F
// R4 = b
// R10 = n-1
```

```
L: LD R5, 0(R1++)
 LD R6, 0(R2++)
 MUL R7, R5, R6
 nop //R7 будет доступно на следующем такте
 ADD R8, R7, R4
 nop //R8 будет доступно на следующем такте
 ST 0(R3++), R8; BL R10, L
```

Задержка арифметических операций – 2 такта

**ST** – операция с памятью, **BL** – операция перехода

## 16.1 Что такое программная конвейеризация

### 16.1.3 Последовательное выполнение итераций цикла

```
for(i = 0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Если итерации выполняются последовательно, то наилучший план для модельного компьютера имеет вид

```
// R1, R2, R3 = &A, &C, &F
// R4 = b
// R10 = n-1
```

```
L: LD R5, 0(R1++)
 LD R6, 0(R2++)
 MUL R7, R5, R6
 nop //R7 будет доступно на следующем такте
 ADD R8, R7, R4
 nop //R8 будет доступно на следующем такте
 ST 0(R3++), R8; BL R10, L
```

Задержка арифметических операций – 2 такта

**ST** – операция с памятью, **BL** – операция перехода

**0 (R1++)** : 0 – смещение относительно базового адреса, являющегося значением регистра **R1**. Операция **++** увеличивает значение **R1** на значение **sizeof (T)**, где **T** – тип значения регистра **R1**.

## 16.1 Что такое программная конвейеризация

### 16.1.3 Последовательное выполнение итераций цикла

```
for (i = 0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Если итерации выполняются последовательно, то наилучший план для модельного компьютера имеет вид

**LD R5, 0 (R1++)** на самом деле означает

**LD nop; R5, 0 (R1++); nop; nop;**

первая **nop** – отсутствие арифметической операции,

вторая **nop** – отсутствие операции **ST**,

третья **nop** – отсутствие операции перехода.

Аналогично **ADD R8, R7, R4** на самом деле означает

**ADD R8, R7, R4; nop; nop; nop;**

А строка с **nop;** на самом деле означает

**nop; nop; nop; nop;**

## 16.1 Что такое программная конвейеризация

### 16.1.3 Последовательное выполнение итераций цикла

```
for (i = 0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Если итерации выполняются последовательно, то наилучший план для модельного компьютера имеет вид

```
// R1, R2, R3 = &A, &C, &F
// R4 = b
// R10 = n-1
```

```
L: LD R5, 0 (R1++)
```

```
LD
```

```
MUL
```

```
nop
```

```
ADD
```

```
nop
```

```
ST
```

Регистры **R1**, **R2** и **R3** сначала содержат адреса первых элементов массивов **A**, **C** и **F** соответственно, а **++** обеспечивает, что на второй итерации эти адреса станут адресами вторых элементов указанных массивов и т.д.

## 16.1 Что такое программная конвейеризация

### 16.1.3 Последовательное выполнение итераций цикла

```
for (i = 0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Если итерации выполняются последовательно, то наилучший план для модельного компьютера имеет вид

```
// R1, R2, R3 = &A, &C, &F
// R4 = b
// R10 = n-1
```

```
L: LD R5, 0 (R1++)
 LD R6, 0 (R2++)
 MUL R7, R5, R6
 nop //R7 будет доступно на следующем такте
 ADD R8, R7, R4
 nop //R8 будет доступно на следующем такте
 ST 0 (R3++), R8; BL R10, L
```

Задержка арифметических операций – 2 такта

- ◇ Отметим, что на каждом такте, кроме последнего, выполняется всего одна операция, а каждая итерация цикла выполняется 7 тактов.

Причина в том – что *в рамках одной итерации цикла нет параллелизма*: весь параллелизм между итерациями



## 16.1 Что такое программная конвейеризация

### 16.1.3 Последовательное выполнение итераций цикла

```
for (i = 0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Если итерации выполняются последовательно, то наилучший план для модельного компьютера имеет вид

```
// R1, R2, R3 = &A, &C, &F
// R4 = b
// R10 = n-1
```

```
L: LD R5, 0 (R1++)
 LD R6, 0 (R2++)
```

Задержка арифметических

**Вывод:** нужно стремиться не к параллельному выполнению каждой итерации цикла, а к параллельному выполнению его итераций

```
пор //R8 будет доступно на следующем такте
ST 0 (R3++), R8; BL R10, L
```

- ◇ Отметим, что на каждом такте, кроме последнего, выполняется всего одна операция, а каждая итерация цикла выполняется 7 тактов.

Причина в том – что *в рамках одной итерации цикла нет параллелизма*: весь параллелизм между итерациями

# 16.1 Что такое программная конвейеризация

## 16.1.4 Частичная развертка цикла

```
for(i =0; i < n; i++) F[i] = A[i] * C[i] + b;
```

Развернуто  
5 итераций  
цикла

◇ Параллелизм между итерациями цикла позволяет использовать развертку цикла, так как в этом случае несколько итераций попадают в один базовый блок.

◇ Для блока, полученного в результате развертки цикла можно применить алгоритм планирования списка, который в случае, когда развернуто 4 итерации, даст план, показанный на рисунке (в плане для краткости указаны только операции)

◇ Итерация развернутого цикла выполняется **13 тактов**, но это 4 итерации исходного цикла. Следовательно, в рассматриваемом случае на одну итерацию исходного цикла тратится **13/4 = 3,25 такта (а не 7 тактов)**

|    |    | 1   | 2   | 3   | 4   | 5      |
|----|----|-----|-----|-----|-----|--------|
| 1  | L: | LD  |     |     |     |        |
| 2  |    | LD  |     |     |     |        |
| 3  |    |     | LD  |     |     |        |
| 4  |    | MUL | LD  |     |     |        |
| 5  |    |     | MUL | LD  |     |        |
| 6  |    | ADD |     | LD  |     |        |
| 7  |    |     | ADD |     | LD  |        |
| 8  |    | ST  |     | MUL | LD  |        |
| 9  |    |     | ST  |     | MUL |        |
| 10 |    |     |     | ADD |     |        |
| 11 |    |     |     |     | ADD |        |
| 12 |    |     |     | ST  |     |        |
| 13 |    |     |     |     | ST  | BL (L) |

# 16.1 Что такое программная конвейеризация

## 16.1.4 Частичная развертка цикла

```
for(i =0; i < n; i++) F[i] = A[i] * C[i] + b;
```

Развернуто  
5 итераций  
цикла

- ◇ Параллелизм между итерациями цикла позволяет использовать

Все дело в том, что модельный компьютер не может начать выполнять две операции **LD** в одном такте,

но может в одном такте начать выполнять операции **LD**, **ST**, **MUL** и **BL (L)**.

При развертке цикла несколько итераций выполняются одновременно и появляется надежда совместить операцию **LD** из одной итерации с операцией **ST** из другой итерации (такт 8 на рисунке).

- ◇ Итерация развернутого цикла выполняется **13 тактов**, но это 4 итерации исходного цикла. Следовательно, в рассматриваемом случае на одну итерацию исходного цикла тратится **13/4 = 3,25 такта** (а не 7 тактов)

|    | 1     | 2   | 3   | 4   | 5      |
|----|-------|-----|-----|-----|--------|
| 1  | L: LD |     |     |     |        |
| 2  | LD    |     |     |     |        |
| 3  |       | LD  |     |     |        |
| 4  | MUL   | LD  |     |     |        |
| 5  |       | MUL | LD  |     |        |
| 6  | ADD   |     | LD  |     |        |
| 7  |       | ADD |     | LD  |        |
| 8  | ST    |     | MUL | LD  |        |
| 9  |       | ST  |     | MUL |        |
| 10 |       |     | ADD |     |        |
| 11 |       |     |     | ADD |        |
| 12 |       |     | ST  |     |        |
| 13 |       |     |     | ST  | BL (L) |

# 16.1 Что такое программная конвейеризация

## 16.1.4 Частичная развертка цикла

```
for(i =0; i < n; i++) F[i] = A[i] * C[i] + b;
```

- ◇ Цикл, развернутый  $k$  раз, требует как минимум  $2k + 5$  тактов,  
 $2k$  тактов уйдет на  $k$  операций **ADD** и  $k$  операций **MUL** и еще 5 тактов на 2 операции **LD**, 1 операцию **ST** и 2 операции **nop**

достигая, таким образом, выполнения одной итерации исходного цикла за  $(2k + 5)/k = 2 + 5/k$  тактов.

Следовательно, чем больше итераций будет развернуто, тем быстрее будет выполнен цикл.

- ◇ При  $k \rightarrow \infty$  полностью развернутый цикл может выполнять одну итерацию исходного цикла в среднем за два такта.
- ◇ **Однако** чем больше итераций будет развернуто, тем большего размера код будет получен:  
    развертка 4 итераций даст 13 команд  
    развертка 8 итераций даст 21 команду  
и так далее.

# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

итерации цикла

|                                                                                                                                                                                                                                                                 |    | $i = 1$ | $i = 2$ | $i = 3$ | $i = 4$ | $i = 5$ |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|---------|---------|---------|---------|---------|
| ◇ Программная конвейеризация – удобный способ достичь оптимального использования ресурсов одновременно с компактным кодом.<br>На рисунке показан <b>пятикратно развернутый код цикла</b><br><code>for(i =0; i &lt; n; i++)<br/>  F[i] = A[i] * C[i] + b;</code> | 1  | LD      |         |         |         |         |
|                                                                                                                                                                                                                                                                 | 2  | LD      |         |         |         |         |
|                                                                                                                                                                                                                                                                 | 3  | MUL     | LD      |         |         |         |
|                                                                                                                                                                                                                                                                 | 4  |         | LD      |         |         |         |
|                                                                                                                                                                                                                                                                 | 5  |         | MUL     | LD      |         |         |
|                                                                                                                                                                                                                                                                 | 6  | ADD     |         | LD      |         |         |
| ◇ ◇ в строке $j$ показаны все операции, выполняемые на такте $j$ ;<br>◇ в столбце $i$ показаны все команды, относящиеся к итерации $i$ .                                                                                                                        | 7  |         |         | MUL     | LD      |         |
|                                                                                                                                                                                                                                                                 | 8  | ST      | ADD     |         | LD      |         |
|                                                                                                                                                                                                                                                                 | 9  |         |         |         | MUL     | LD      |
| ◇ Свойства плана:<br>◇ каждая итерация $i$ имеет один и тот же план выполнения относительно ее начала<br>◇ каждая итерация $i$ начинается через два такта после начала предыдущей.                                                                              | 10 |         | ST      | ADD     |         | LD      |
|                                                                                                                                                                                                                                                                 | 11 |         |         |         |         | MUL     |
|                                                                                                                                                                                                                                                                 | 12 |         |         | ST      | ADD     |         |
|                                                                                                                                                                                                                                                                 | 13 |         |         |         |         |         |
|                                                                                                                                                                                                                                                                 | 14 |         |         |         | ST      | ADD     |
|                                                                                                                                                                                                                                                                 | 15 |         |         |         |         |         |
| ◇ План удовлетворяет всем ограничениям, накладываемым ресурсами и зависимостями данных.                                                                                                                                                                         | 16 |         |         |         |         | ST      |

# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

итерации цикла

|   |                                                                                                                          | $i = 1$ | $i = 2$ | $i = 3$ | $i = 4$ | $i = 5$ |
|---|--------------------------------------------------------------------------------------------------------------------------|---------|---------|---------|---------|---------|
| ◇ | Программная конвейеризация – удобный способ достичь оптимального использования ресурсов одновременно с компактным кодом. | 1       | LD      |         |         |         |
|   |                                                                                                                          | 2       | LD      |         |         |         |
|   | На рисунке показан <b>пятикратно развернутый код цикла</b>                                                               | 3       | MUL     | LD      |         |         |
|   | <code>for(i = 0; i &lt; n; i++)</code>                                                                                   | 4       |         | LD      |         |         |
|   | <code>  F[i] = A[i] * C[i] + b;</code>                                                                                   | 5       |         | MUL     | LD      |         |
|   |                                                                                                                          | 6       | ADD     |         | LD      |         |
| ◇ | ◇ в строке $j$ показаны все операции, выполняемые на такте $j$ ;                                                         | 7       |         |         | MUL     | LD      |
|   | ◇ в столбце $i$ показаны все команды, относящиеся                                                                        | 8       | ST      | ADD     |         | LD      |
| ◇ | Свойства плана                                                                                                           |         |         |         |         | LD      |
|   | ◇ каждая итерация                                                                                                        |         |         |         |         | LD      |
|   | же план выполнения относительно ее начала                                                                                | 12      |         |         | ST      | ADD     |
|   | ◇ каждая итерация $i$ начинается через два такта после начала предыдущей.                                                | 13      |         |         |         |         |
|   |                                                                                                                          | 14      |         |         | ST      | ADD     |
|   |                                                                                                                          | 15      |         |         |         |         |
| ◇ | План удовлетворяет всем ограничениям, накладываемым ресурсами и зависимостями данных.                                    | 16      |         |         |         | ST      |

# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

итерации цикла

|   |                                                                                                                                                                     | $i = 1$ | $i = 2$ | $i = 3$ | $i = 4$ | $i = 5$ |     |
|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|---------|---------|---------|---------|-----|
| ◇ | Программная конвейеризация – удобный способ достичь оптимального использования ресурсов одновременно с компактным кодом.                                            | 1       | LD      |         |         |         |     |
|   | На рисунке показан <b>пятикратно развернутый код</b>                                                                                                                | 2       | LD      |         |         |         |     |
|   | <code>for (i = 0; i &lt; 16; i++)</code>                                                                                                                            | 3       | MUL     | LD      |         |         |     |
|   | <code>  F[i] = A[i] + B[i];</code>                                                                                                                                  |         |         | LD      |         |         |     |
|   | использует результат <code>MUL</code> из 3-ей строки, так как                                                                                                       |         |         | MUL     | LD      |         |     |
| ◇ | ◇ в строке $j$ показаны все команды, выполняемые в итерации $j$                                                                                                     |         | ADD     |         | LD      |         |     |
|   | ◇ в столбце $i$ показаны все команды, относящиеся к итерации $i$                                                                                                    |         |         |         | MUL     | LD      |     |
|   | В каждой итерации цикла: <code>MUL</code> и <code>ADD</code> не могут быть в одной строке; также между <code>ADD</code> и <code>ST</code> должна быть пустая строка |         | ST      | ADD     |         | LD      |     |
| ◇ | Свойства плана выполнения:                                                                                                                                          |         |         |         |         |         |     |
| ◇ | каждая итерация $i$ начинается через два такта после начала предыдущей.                                                                                             | 12      |         |         | ST      | ADD     |     |
|   | Отметим также, что каждая строка плана содержит <b>nop</b> вместо команды перехода                                                                                  | 13      |         |         |         | ST      | ADD |
| ◇ | План удовлетворяет ограничениям, накладываемым ресурсами и зависимостями данных.                                                                                    | 16      |         |         |         | ST      |     |

# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

итерации цикла

|   |                                                                                                                                                      | <i>i</i> = 1 | <i>i</i> = 2 | <i>i</i> = 3 | <i>i</i> = 4 | <i>i</i> = 5 |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|--------------|--------------|--------------|--------------|
| ◇ | На тактах 7 и 8 и на тактах 9 и 10 выполняются <i>одинаковые операции</i> .                                                                          | 1            | LD           |              |              |              |
| ◇ | Во время тактов 7 и 8 выполняются операции из четырех итераций исходной программы – с первой по четвертую.                                           | 2            | LD           |              |              |              |
|   | Во время тактов 9 и 10 выполняются операции из четырех итераций исходной программы со второй по пятую.                                               | 3            | MUL          | LD           |              |              |
|   |                                                                                                                                                      | 4            |              | LD           |              |              |
|   |                                                                                                                                                      | 5            |              | MUL          | LD           |              |
|   |                                                                                                                                                      | 6            | ADD          |              | LD           |              |
|   |                                                                                                                                                      | 7            |              |              | MUL          | LD           |
|   |                                                                                                                                                      | 8            | ST           | ADD          |              | LD           |
| ◇ | План можно рассматривать как выполнение в цикле команд, содержащих по три операции: <code>&lt;nop MUL LD&gt;</code> и <code>&lt;ST ADD LD&gt;</code> | 9            |              |              | MUL          | LD           |
|   |                                                                                                                                                      | 10           |              | ST           | ADD          | LD           |
|   |                                                                                                                                                      | 11           |              |              |              | MUL          |
| ◇ | Работа цикла представляет собой завершение самой старой итерации и начало новой итерации до тех пор, пока не будут выполнены все итерации.           | 12           |              | ST           | ADD          |              |
|   |                                                                                                                                                      | 13           |              |              |              |              |
|   |                                                                                                                                                      | 14           |              |              | ST           | ADD          |
|   |                                                                                                                                                      | 15           |              |              |              |              |
|   | <code>for(i =0; i &lt; n; i++)</code><br><code>    F[i] = A[i] * C[i] + b;</code>                                                                    | 16           |              |              |              | ST           |



# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

- ◇ Такое динамическое поведение можно представить при помощи кода, показанного на рисунке (считая, что цикл содержит не меньше четырех итераций).
- ◇ Каждая строка на рисунке соответствует одной машинной команде.
- ◇ Строки 7 и 8 образуют цикл, который выполняется  $n - 3$  раза ( $n$  – количество итераций в исходном цикле; как уже было отмечено, предполагается, что  $n \geq 4$ )
- ◇ Описанный метод называется *программной конвейеризацией*.
- ◇ План, выполняющий каждую итерацию в данном примере, можно рассматривать как восьмиэтапный конвейер.

|    |     |     |     |     |        |  |  |
|----|-----|-----|-----|-----|--------|--|--|
| 1  | LD  |     |     |     |        |  |  |
| 2  | LD  |     |     |     |        |  |  |
| 3  | MUL | LD  |     |     |        |  |  |
| 4  |     | LD  |     |     |        |  |  |
| 5  |     | MUL | LD  |     |        |  |  |
| 6  | ADD |     | LD  |     |        |  |  |
| 7  | L:  |     | MUL | LD  |        |  |  |
| 8  | ST  | ADD |     | LD  | BL (L) |  |  |
| 9  |     |     |     | MUL |        |  |  |
| 10 |     | ST  | ADD |     |        |  |  |
| 11 |     |     |     |     |        |  |  |
| 12 |     |     | ST  | ADD |        |  |  |
| 13 |     |     |     |     |        |  |  |
| 14 |     |     |     | ST  |        |  |  |

# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

- ◇ Новая итерация может начинаться в конвейере каждые два такта.
- ◇ Сначала в конвейере находится только одна итерация. Когда первая итерация переходит на третий этап, начинается выполнение первого этапа второй итерации.
- ◇ На восьмом такте конвейер полностью заполнен: параллельно выполняются четыре итерации.
- ◇ Когда наиболее ранняя итерация заканчивается и покидает конвейер, начинается выполнение новой итерации. Это обеспечивает, которая не относится ни к одной итерации.
- ◇ Когда новых итераций больше нет, конвейер начинает опустошаться. Все оставшиеся итерации выполняются до конца.

|    |    |     |     |     |     |        |
|----|----|-----|-----|-----|-----|--------|
| 1  |    | LD  |     |     |     |        |
| 2  |    | LD  |     |     |     |        |
| 3  |    | MUL | LD  |     |     |        |
| 4  |    |     | LD  |     |     |        |
| 5  |    |     | MUL | LD  |     |        |
| 6  |    | ADD |     | LD  |     |        |
| 7  | L: |     |     | MUL | LD  |        |
| 8  |    | ST  | ADD |     | LD  | BL (L) |
| 9  |    |     |     |     | MUL |        |
| 10 |    |     | ST  | ADD |     |        |
| 11 |    |     |     |     |     |        |
| 12 |    |     |     | ST  | ADD |        |
| 13 |    |     |     |     |     |        |
| 14 |    |     |     |     | ST  |        |

# 16.1 Что такое программная конвейеризация

## 16.1.5 Программная конвейеризация цикла

- ◇ Новая итерация может начинаться в конвейере каждые два такта.

|    |     |     |     |     |        |
|----|-----|-----|-----|-----|--------|
| 1  | LD  |     |     |     |        |
| 2  | LD  |     |     |     |        |
| 3  | MUL | LD  |     |     |        |
| 4  |     | LD  |     |     |        |
| 5  |     | MUL | LD  |     |        |
| 6  | ADD |     | LD  |     |        |
| 7  | L:  |     | MUL | LD  |        |
| 8  | ST  | ADD |     | LD  | BL (L) |
| 9  |     |     |     |     |        |
| 10 |     | ST  | ADD |     |        |
| 11 |     |     |     |     |        |
| 12 |     |     | ST  | ADD |        |
| 13 |     |     |     |     |        |
| 14 |     |     |     | ST  |        |
- ◇ Сначала в конвейере находится только одна итерация. Когда первая итерация переходит на третий этап, начинается выполнение первого этапа второй итерации.
- ◇ На восьмом такте конвейер полностью заполнен: параллельно выполняются четыре итерации.
- ◇ Отметим, что только 8-ой такт не содержит ни одной операции **nop**.

|    |  |    |     |     |  |
|----|--|----|-----|-----|--|
| 10 |  | ST | ADD |     |  |
| 11 |  |    |     |     |  |
| 12 |  |    | ST  | ADD |  |
| 13 |  |    |     |     |  |
| 14 |  |    |     | ST  |  |
- ◇ Когда новых итераций больше нет, конвейер начинает опустошаться. Все оставшиеся итерации выполняются до конца.

## 16.1 Что такое программная конвейеризация

### 16.1.5 Программная конвейеризация цикла

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                           |                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◇ Последовательность команд, необходимых для заполнения конвейера (команды с 1 по 6), называется <b><i>прологом</i></b>.</li> <li>◇ Команды 7 и 8 представляют <b><i>устойчивое состояние</i></b>.</li> <li>◇ Последовательность команд, необходимых для опустошения конвейера (команды с 9 по 14), называется <b><i>эпилогом</i></b>.</li> <li>◇ В данном примере цикл не может работать быстрее, чем итерация за два такта (в каждой итерации два LD). Программно-конвейеризованный цикл выполняется за <math>2n + 6</math> тактов (<math>n</math> – число итераций в исходном цикле).</li> <li>◇ При <math>n \rightarrow \infty</math> скорость работы цикла стремится к одной итерации за два такта.</li> </ul> | <div style="background-color: #e0f0ff; padding: 10px; border: 1px solid #add8e6;"> </div> | <p>П<br/>р<br/>о<br/>л<br/>о<br/>г</p> <p>Э<br/>п<br/>и<br/>л<br/>о<br/>г</p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
- Программная конвейеризация может обеспечить оптимальный

**Программная конвейеризация может обеспечить оптимальный план при компактном коде.**

## 16.2 Программная конвейеризация цикла

### 16.2.1 Принцип конвейерного планирования

- ◇ По отношению к отдельно взятой итерации составленный план не является оптимальным: по сравнению с локально оптимизированным планом он содержит лишнюю задержку перед операцией **ADD**.
- ◇ **Цель этой стратегической задержки: обеспечить начало каждой новой итерации спустя два такта после начала предыдущей без каких-либо конфликтов ресурсов.**
- ◇ Если использовать локальный план, то во избежание конфликтов каждая новая итерация должна будет начинаться через четыре такта после начала предыдущей, и скорость работы программы станет вдвое меньше.
- ◇ Пример иллюстрирует *принцип конвейерного планирования*: **для оптимизации производительности следует очень аккуратно выбирать план одной итерации.**  
Локально оптимизированный план, минимизирующий время выполнения одной итерации, может привести к неоптимальной производительности при конвейеризации

## 16.2 Программная конвейеризация цикла

### 16.2.2 Добавление регистров

- ◇ Осталось полностью составить программу для рассматриваемого примера – не только операции, но и их операнды (регистры).
- ◇ Результат операции умножения в первой итерации получается на **3** такте, а используется – на **6** такте.  
Результат операции умножения во второй итерации получается на **5** такте, а используется – на **8** такте.  
Поэтому результаты этих двух операций умножения должны храниться на разных регистрах.
- ◇ Временные интервалы существования результата перекрываются только для соседних итераций, поэтому для хранения указанных значений достаточно использовать два разных регистра:
  - ◇ один – для нечетных итераций,
  - ◇ другой – для четных.
- ◇ В результате добавления регистров код нечетных итераций отличается от кода четных итераций (регистры **R7** и **R9**), таким образом *размер устойчивого состояния цикла удваивается.*

## 16.2 Программная конвейеризация цикла

### 16.2.2 Добавление регистров

◇ Цикл `for (i = 0; i < N2; i++) F[i] = A[i]* C[i] + b;`

|    |    |     |          |          |     |                               |        |           |           |    |    |       |
|----|----|-----|----------|----------|-----|-------------------------------|--------|-----------|-----------|----|----|-------|
| 1  | LD | R5, | 0 (R1++) |          |     | // R1 = &A, R2 = &C, R3 = &F, |        |           |           |    |    |       |
| 2  | LD | R6, | 0 (R2++) |          |     | // R4 = b, R10 = N2 - 1       |        |           |           |    |    |       |
| 3  | LD | R5, | 0 (R1++) | MUL      | R7, | R5, R6                        |        |           |           |    |    |       |
| 4  | LD | R6, | 0 (R2++) |          |     |                               |        |           |           |    |    |       |
| 5  | LD | R5, | 0 (R1++) | MUL      | R9, | R5, R6                        |        |           |           |    |    |       |
| 6  | LD | R6, | 0 (R2++) | ADD      | R8, | R7, R4                        |        |           |           |    |    |       |
| 7  | L: | LD  | R5,      | 0 (R1++) | MUL | R7,                           | R5, R6 |           |           |    |    |       |
| 8  |    | LD  | R6,      | 0 (R2++) | ADD | R8,                           | R9, R4 | ST        | 0 (R3++), | R8 |    |       |
| 9  |    | LD  | R5,      | 0 (R1++) | MUL | R9,                           | R5, R6 |           |           |    |    |       |
| 10 |    | LD  | R6,      | 0 (R2++) | ADD | R8,                           | R7, R4 | ST        | 0 (R3++), | R8 | BL | R10,L |
| 11 |    |     |          | MUL      | R9, | R5, R6                        |        |           |           |    |    |       |
| 12 |    |     |          | ADD      | R8, | R9, R4                        | ST     | 0 (R3++), | R8        |    |    |       |
| 13 |    |     |          |          |     |                               |        |           |           |    |    |       |
| 14 |    |     |          | ADD      | R8, | R7, R4                        | ST     | 0 (R3++), | R8        |    |    |       |
| 15 |    |     |          |          |     |                               |        |           |           |    |    |       |
| 16 |    |     |          |          |     |                               | ST     | 0 (R3++), | R8        |    |    |       |

## 16.2 Программная конвейеризация цикла

### 16.2.2 Добавление регистров

◇ Цикл `for (i = 0; i < N2; i++) F[i] = A[i]* C[i] + b;`

```
1 LD R5, 0(R1++) // R1 = &A, R2 = &C, R3 = &F,
2 LD R6, 0(R2++) // R4 = b, R10 = N2 - 1
3 LD R5, 0(R1++) MUL R7, R5, R6
4 LD R6, 0(R2++)
```

Результат операции `MUL` на нечетной итерации помещается на регистр `R7`, а на четной итерации – на регистр `R9`, так как если бы оба результата помещались на один и тот же регистр, операция `ADD` не успевала бы его использовать.

Таким образом, **зависимость по данным удваивает размер основного цикла.**

```
8 LD R6, 0(R2++) ADD R8, R9, R4 ST 0(R3++), R8
9 LD R5, 0(R1++) MUL R9, R5, R6
10 LD R6, 0(R2++) ADD R8, R7, R4 ST 0(R3++), R8 BL R10,L
11 MUL R9, R5, R6
12 ADD R8, R9, R4 ST 0(R3++), R8
13
14 ADD R8, R7, R4 ST 0(R3++), R8
15
16 ST 0(R3++), R8
```



## 16.2 Программная конвейеризация цикла

### 16.2.2 Добавление регистров

- ◇ Для работы с циклами, содержащими менее пяти итераций, и циклами с четным числом итераций ( $N2 = N - 3$  четное) можно использовать код, который на уровне исходного языка эквивалентен следующему:

```
if (N >= 5) //N2 – четное число или 0
 N2 = 3 + 2 * floor((N - 3) / 2);
else N2 = 0;
for (i = 0; i < N2; i++) //четное число итераций
 F[i] = A[i]* C[i] + d;
for (i = N2; i < N; i++) // N - N2 ≤ 4
 F[i] = A[i]* C[i] + d;
```

- ◇ Первый цикл имеет четное количество итераций, и для него можно составить конвейеризованный план.
- ◇ Второй цикл не требует оптимизации, так как в нем выполняется не более четырех итераций. Этот цикл добавляется к эпилогу.

## 16.2 Программная конвейеризация цикла

### 16.2.3 Перекрестные циклы

- ◇ Программная конвейеризация применима и к перекрестным циклам.  
Цикл называется *перекрестным*, если между его итерациями имеются зависимости по данным.

- ◇ **Пример.** Код

```
for (i = 0; i < n; i++) {
 sum = sum + A[i];
 B[i] = A[i] * d;
}
```

(массивы **A**, **B** и переменные **d** и **sum** имеют тип **float**, что исключает изменение порядка суммирования), содержит зависимости по данным между соседними итерациями по переменной **sum**: для получения нового значения **sum** используется старое значение **sum**.

## 16.2 Программная конвейеризация цикла

### 16.2.3 Перекрестные циклы

◇ Пример. Код

```
for (i = 0; i < n; i++) {
 sum = sum + A[i];
 B[i] = A[i] * d;
}
```

◇ В рассматриваемой модели процессора **ADD** выполняется за два такта, так что цикл не может выполняться быстрее, чем одна итерация за два такта.

◇ Добавочные функциональные устройства не помогут ускорить перекрестный цикл: **его производительность ограничена цепочкой зависимостей между итерациями.**

## 16.2 Программная конвейеризация цикла

### 16.2.3 Перекрестные циклы

◇ Локально оптимизированный план каждой итерации:

```
// R1 = &A; R2 = &B; R3 = sum
// R4 = d; R10 = n-1
L: LD R5, 0 (R1++)
 MUL R6, R5, R4
 ADD R3, R3, R5
 ST R6, 0 (R2++) BL R10, L
```

## 16.2 Программная конвейеризация цикла

### 16.2.3 Перекрестные циклы

◇ Локально оптимизированный план каждой итерации:

```
// R1 = &A; R2 = &B; R3 = sum
// R4 = d; R10 = n-1

L: LD R5, 0 (R1++)
 MUL R6, R5, R4
 ADD R3, R3, R5
 ST R6, 0 (R2++) BL R10, L
```

На R5 загружается значение очередного элемента **A[i]**

Это значение умножается на **d** (регистр R4) и получается значение **B[i]** (на регистре R6).

К старому значению **sum** (регистр R3) прибавляется значение **A[i]** (регистр R5) и новое значение **sum** запоминается на регистре R3.

Значение **B[i]** (регистр R6) запоминается в очередном элементе массива **B[]**, адрес которого получается на регистре R2.

## 16.2 Программная конвейеризация цикла

### 16.2.3 Перекрестные циклы

◇ Программно-конвейеризованная версия имеет вид:

```
// R1 = &A; R2 = &B
```

```
// R3 = sum; R4 = d; R10 = n-2
```

```
LD R5, 0(R1++)
```

```
MUL R6, R5, R4
```

```
L: ADD R3, R3, R4 LD R5, 0(R1++)
```

```
ST R6, 0(R2++) MUL R6, R5, R4 BL R10, L
```

```
ADD R3, R3, R4
```

```
ST R6, 0(R2++)
```

## 16.2 Программная конвейеризация цикла

### 16.2.3 Перекрестные циклы

◇ Программно-конвейеризованная версия имеет вид:

```
// R1 = &A; R2 = &B
```

```
// R3 = sum; R4 = d; R10 = n-2
```

```
LD R5, 0(R1++)
```

```
MUL R6, R5, R4
```

```
L: ADD R3, R3, R4 LD R5, 0(R1++)
```

```
ST R6, 0(R2++) MUL R6, R5, R4 BL R10, L
```

```
ADD R3, R3, R4
```

```
ST R6, 0(R2++)
```

**ST** нельзя поместить вместо **nop** в предыдущей строке, так как задержка операции **MUL** равна 2.

## 16.3 Цели и ограничения программной конвейеризации

### 16.3.1 Цели программной конвейеризации

- ◇ Цели программной конвейеризации
  - ◇ увеличение производительности долго выполняющихся циклов.
  - ◇ генерация кода относительно малого размера: устойчивое состояние конвейера программно-конвейеризованного цикла должно быть достаточно малым.
- ◇ *Производительность цикла* определяется как величина, обратная интервалу  $T$  между запусками итераций. Следовательно, **цель программной конвейеризации – минимизация  $T$ .**
- ◇ Чтобы устойчивое состояние конвейера было достаточно малым, достаточно, чтобы:
  - ◇ относительные планы каждой итерации цикла были одинаковыми
  - ◇ интервалы между любыми двумя последовательными итерациями цикла были одинаковыми



## 16.3 Цели и ограничения программной конвейеризации

### 16.3.2 Ограничения программной конвейеризации

- ◇ План программной конвейеризации для графа зависимостей по данным  $G = (N, E)$  определяется
  - ◇ интервалом между запусками итераций  $T$ ;
  - ◇ относительным планом  $S$ , который для каждой операции указывает время ее выполнения относительно начала итерации, которой принадлежит эта операция.

Операция  $n$  в  $i$ -ой итерации, считая от нуля, выполняется в момент  $i \times T + S(n)$ .

- ◇ Представим машинные ресурсы в виде  $R = [r_1, r_2, \dots]$ , где  $r_i$  – количество единиц  $i$ -го ресурса.

Если итерация цикла требует  $n_i$  единиц ресурса  $i$ , то средний интервал между запусками итераций конвейеризованного цикла составляет не меньше, чем  $\max_i (n_i / r_i)$  тактов.

- ◇ Для того, чтобы интервалы запуска между любыми соседними итерациями были одинаковы, необходимо, чтобы интервал между запусками содержал не меньше  $\max_i \lceil n_i / r_i \rceil$  тактов.

## 16.3 Цели и ограничения программной конвейеризации

### 16.3.2 Ограничения программной конвейеризации

- ◇ План программной конвейеризации для графа зависимостей по данным  $G = (N, E)$  определяется
  - ◇ интервалом между запусками итераций  $T$ ;
  - ◇ относительным планом  $S$ , который для каждой операции указывает время ее выполнения относительно начала итерации, которой принадлежит эта операция.

Операция  $n$  в  $i$ -ой итерации начинается в момент  $i \times T - \text{начало итерации}$ ,  $S(n)$  – начало операции  $n$  в каждой итерации.

- ◇ Представим машинные ресурсы в виде  $R = [r_1, r_2, \dots]$ ,

Если на итерации требуется выполнить 7 операций сложения, а у процессора всего 2 функциональных устройства, выполняющих сложение, то для выполнения итерации потребуется не менее, чем  $7/2 = 3,5$  тактов (т.е. 4 тактов)

- ◇ Для того, чтобы интервалы запуска между любыми соседними итерациями были одинаковы, необходимо, чтобы интервал между запусками содержал не меньше  $\max_i \lceil n_i / r_i \rceil$  тактов.

# 16.3 Цели и ограничения программной конвейеризации

## 16.3.3 Пример

- ◇ Рассмотрим программно-конвейеризованный цикл, показанный на рисунке (для краткости в плане показаны операции без операндов).
- ◇ За один такт целевая машина может выполнить одну загрузку, одну арифметическую операцию, одно сохранение и одну операцию перехода.
- ◇ Цикл содержит две загрузки (**LD**), две арифметические операции (**MUL** и **ADD**) и одно сохранение (**ST**)
- ◇ Следовательно, минимальный интервал  $T$  между запусками итераций, вычисленный на основании ограничений, связанных с ресурсами, равен 2 тактам:  $T = 2$

|    |    |     |     |     |           |
|----|----|-----|-----|-----|-----------|
| 1  |    | LD  |     |     |           |
| 2  |    | LD  |     |     |           |
| 3  |    | MUL | LD  |     |           |
| 4  |    |     | LD  |     |           |
| 5  |    |     | MUL | LD  |           |
| 6  |    | ADD |     | LD  |           |
| 7  | L: |     |     | MUL | LD        |
| 8  |    | ST  | ADD |     | LD BL (L) |
| 9  |    |     |     |     | MUL       |
| 10 |    |     | ST  | ADD |           |
| 11 |    |     |     |     |           |
| 12 |    |     |     | ST  | ADD       |
| 13 |    |     |     |     |           |
| 14 |    |     |     |     | ST        |

# 16.3 Цели и ограничения программной конвейеризации

## 16.3.3 Пример

- ◇ Рассмотрим программно-конвейеризованный цикл, показанный на рисунке (для краткости в плане показаны операции без операндов).
- ◇ За один такт целевая машина может выполнить одну загрузку, одну арифметическую операцию, одно сохранение и одну операцию перехода.
- ◇ Цикл содержит две загрузки (**LD**), две арифметические операции (**MUL** и **ADD**) и одно сохранение (**ST**)
- ◇ Следовательно, минимальный интервал  $T$  между запусками итераций, вычисленный на основании ограничений, связанных с ресурсами, равен 2 тактам:  $T = 2$

|    |    |     |     |     |     |
|----|----|-----|-----|-----|-----|
| 1  |    | LD  |     |     |     |
| 2  |    | LD  |     |     |     |
| 3  |    | MUL | LD  |     |     |
| 4  |    |     | LD  |     |     |
| 5  |    |     | MUL | LD  |     |
| 6  |    | ADD |     | LD  |     |
| 7  | L: |     |     | MUL | LD  |
| 8  |    | ST  | ADD |     | LD  |
| 9  |    |     |     |     | MUL |
| 10 |    |     | ST  | ADD |     |
| 11 |    |     |     |     |     |
| 12 |    |     |     | ST  | ADD |
| 13 |    |     |     |     |     |
| 14 |    |     |     |     | ST  |

**MUL** и **ADD** нельзя запустить в одном и том же такте, так как для запуска каждой из них требуется одно и то же ФУ

# 16.3 Цели и ограничения программной конвейеризации

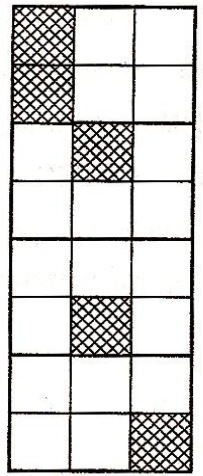
## 16.3.3 Пример

На каждой итерации требуется выполнить 2 операции LD, 2 арифметические операции (MUL и ADD) и одну операцию ST.

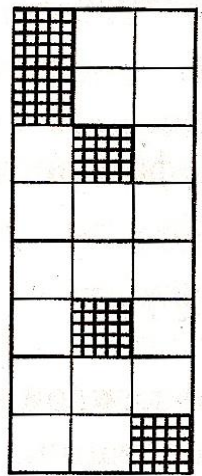
LD  
LD  
MUL  
  
ADD  
  
ST

Время  
↓

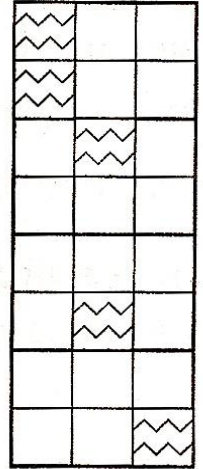
Итерация 1  
LD ALU ST



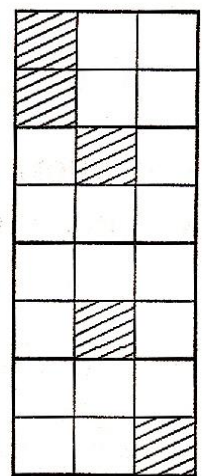
Итерация 2  
LD ALU ST



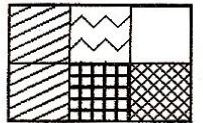
Итерация 3  
LD ALU ST



Итерация 4  
LD ALU ST



Устойчивое состояние  
LD ALU ST

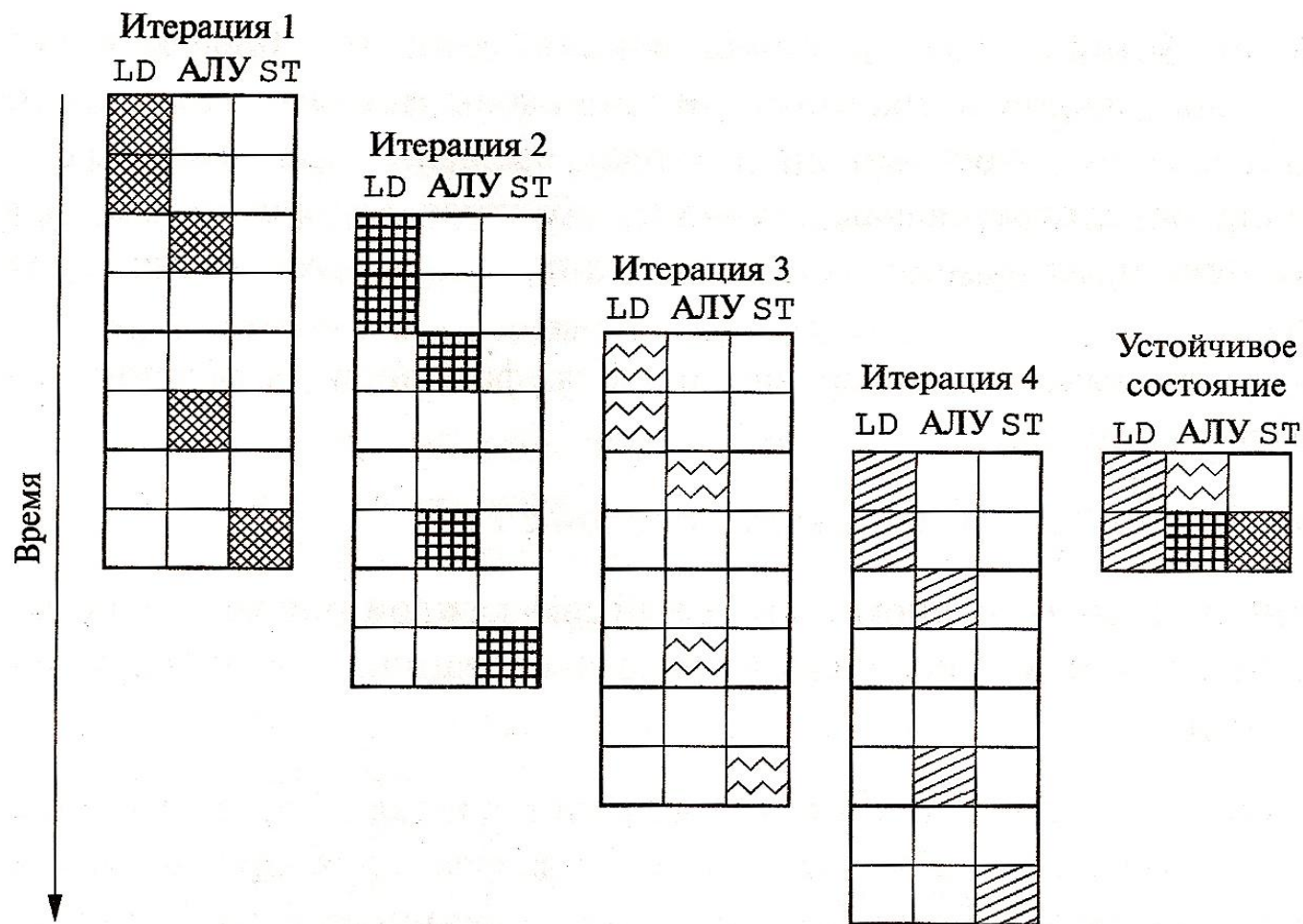


MUL LD  
ST ADD LD

На рисунке показаны требования четырех последовательных итераций к ресурсам в разные моменты времени. Чем больше итераций запущено, тем выше требования к ресурсам.  
Максимальные требования к ресурсам – в устойчивом состоянии.

# 16.3 Цели и ограничения программной конвейеризации

## 16.3.3 Пример



На рисунке показаны требования четырех последовательных итераций к ресурсам в разные моменты времени. Чем больше итераций запущено, тем выше требования к ресурсам.

Максимальные требования к ресурсам – в устойчивом состоянии.

## 16.3 Цели и ограничения программной конвейеризации

### 16.3.3 Пример

◇ Пусть

$RT[j]$  – таблица резервирования ресурсов итерации  $j$ ,

$RT_S$  – таблица резервирования ресурсов устойчивого состояния (она называется *таблицей модульного резервирования ресурсов*).

◇  $RT_S$  объединяет запросы четырех последовательных итераций, начавшихся в пределах  $T$  тактов.

◇ Запросы в 0-й строке  $RT_S$  соответствуют сумме ресурсов, запрошенных в таблицах  $RT[0]$ ,  $RT[2]$ ,  $RT[4]$  и  $RT[6]$ .

Запросы в 1-й строке  $RT_S$  соответствуют сумме ресурсов, запрошенных в таблицах  $RT[1]$ ,  $RT[3]$ ,  $RT[5]$  и  $RT[7]$ .

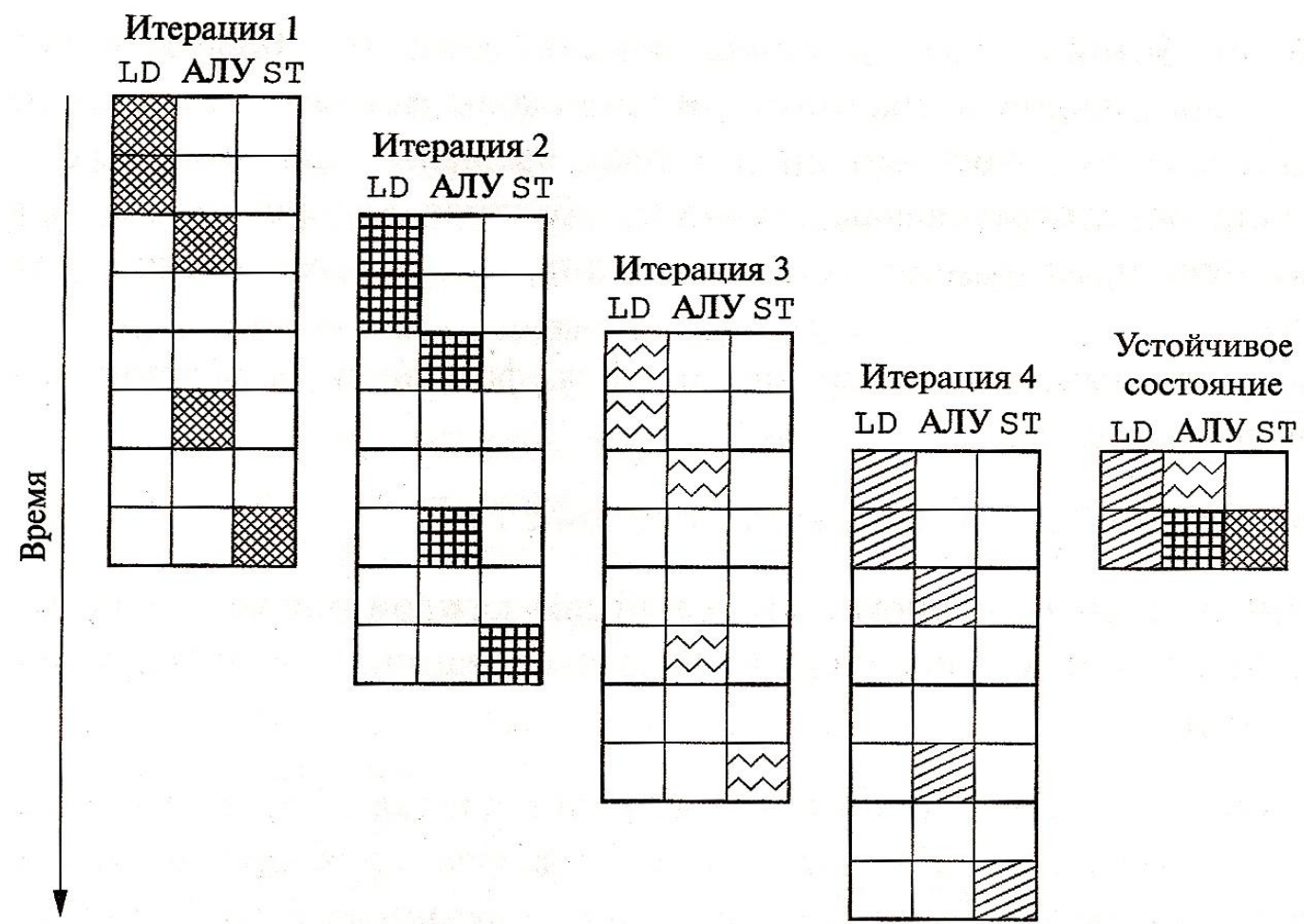
Таким образом, в рассматриваемом примере ресурсы, запрошенные в  $i$ -й строке  $RT_S$  равны:

$$RT_S[i] = \sum_{\{t | (t \bmod 2 = i)\}} RT[t]$$



# 16.3 Цели и ограничения программной конвейеризации

## 16.3.3 Пример



Чтобы проверить наличие в плане программной конвейеризации конфликтов, связанных с ресурсами, можно просто проверить запросы в таблице модульного резервирования ресурсов. Само собой разумеется, если запросы в устойчивом состоянии могут быть удовлетворены, то тем более они могут быть удовлетворены в прологе и эпилоге.



## 16.3 Цели и ограничения программной конвейеризации

### 16.3.3 Пример

- ◇ В общем случае для данного интервала между запусками  $T$  и таблицы резервирования ресурсов итерацией  $RT$  конвейеризованный план на машине с вектором ресурсов  $R$  не имеет конфликтов, связанных с ресурсами, тогда и только тогда, когда  $RT_S[j] \leq R$  для всех  $j = 0, 1, \dots, T-1$ .

На каждом такте  $j$  количество требуемых ресурсов  $RT_S[j]$  не должно превышать количества имеющихся ресурсов  $R$ .

## 16.3 Цели и ограничения программной конвейеризации

### 16.3.4 Ограничения, связанные с зависимостями по данным

- ◇ Зависимости по данным при программной конвейеризации **могут образовывать циклы**: операция  $n$  может зависеть от результата операции  $n$  из предыдущей итерации.
- ◇ Ребра графа зависимостей недостаточно помечать значениями задержек; требуется различать и различные экземпляры одной и той же операции в разных итерациях.
- ◇ Ребро графа зависимостей  $n_1 \rightarrow n_2$  помечается меткой  $\langle \delta, d \rangle$ , где  $\delta$  – разность между номерами итераций,  $d$  – время задержки в тактах:

$n_1 \xrightarrow{\langle \delta, d \rangle} n_2$  означает, что операция  $n_2$  в итерации  $i$  должна быть задержана как минимум на  $d$  тактов после выполнения операции  $n_1$  в итерации  $i - \delta$

## 16.3 Цели и ограничения программной конвейеризации

### 16.3.4 Ограничения, связанные с зависимостями по данным

◇ План  $S$ , полученный путем программной конвейеризации, можно рассматривать как функцию, определенную на множестве узлов графа зависимости по данным, областью значений которой являются целые числа – номера тактов.

◇ Имеет место соотношение:

$$S(n_2) - S(n_1) \geq d - (\delta \times T)$$

или

$$(\delta \times T) + S(n_2) - S(n_1) \geq d$$

где  $\langle \delta, d \rangle$  – метка ребра  $n_1 \rightarrow n_2$ ,  $T$  – интервал между запусками итераций,  $S(n)$  – номер такта, на котором запускается операция  $n$ .

◇ Разность итераций  $\delta$  – неотрицательна.  
Более того, для данного цикла из ребер зависимости данных как минимум одно ребро имеет положительную разность итераций.

## **16.3 Цели и ограничения программной конвейеризации**

### **16.3.4 Ограничения, связанные с зависимостями по данным**

- ◇ Итак, установлено, что интервал между запусками итераций каждого программно-конвейеризованного цикла ограничен использованием ресурсов на каждой итерации. А именно, этот интервал должен быть не меньше, чем отношение количества необходимых единиц каждого ресурса и количества доступных единиц ресурса в целевой машине.
- ◇ Кроме того, если имеются циклические зависимости по данным, то интервал между запусками оказывается еще более ограниченным суммой задержек в цикле, деленной на сумму разностей итераций.  
Наибольшая из этих величин определяет нижнюю границу между запусками итераций.

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.1 Алгоритм программной конвейеризации**

- ◇ Цель программной конвейеризации состоит в том, чтобы найти план с наименьшим возможным интервалом между запусками итераций. Это NP-полная задача, которая может быть переформулирована как задача целочисленного линейного программирования.
- ◇ Было показано, что если величина минимального интервала известна, то алгоритм разработки расписания выдачи операций может избежать конфликта по ресурсам с помощью таблиц модульного резервирования ресурсов при размещении каждой операции.  
Но величина минимального интервала становится известной только после разработки расписания. **Как же разорвать этот круг?**

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.1 Алгоритм программной конвейеризации**

- ◇ Было показано, что интервал между запусками операций должен быть больше, чем граница, вычисленная с учетом требований цикла к ресурсам и чем граница, учитывающая циклические зависимости по данным.  
Если удастся составить расписание, учитывающее эти ограничения, то это – оптимальное расписание.  
Если же такое расписание составить не удастся, можно повторить попытку его составления с большим значением интервала между запусками операций, и так до тех пор, пока расписание не будет составлено.
- ◇ Заметим, что если используется не исчерпывающий поиск, а эвристика, то оптимальное расписание может так и не быть составлено.
- ◇ Можно ли в последнем случае составить расписание, близкое к оптимальному, зависит от архитектуры целевого компьютера и свойств графа зависимостей по данным.

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.1 Алгоритм программной конвейеризации**

- ◇ Расписание, близкое к оптимальному, нетрудно составить в случае ациклического графа зависимостей по данным, при условии, что каждая машинная команда требует для выполнения только одной единицы каждого ресурса.
- ◇ Расписание, близкое к оптимальному, можно составить и в том случае, когда целевой компьютер предоставляет больше аппаратных ресурсов, чем может быть использовано, когда граф зависимостей по данным имеет циклы.  
В этом случае можно начать с нижней границы в качестве начального значения интервала между запусками итераций, а затем увеличивать этот интервал на один такт для каждой следующей попытки построения расписания.  
Другая возможность вычисления близкого к оптимальному интервала между запусками операций состоит в бинарном поиске, когда в качестве верхней границы используется длина расписания для одной итерации, полученную с помощью алгоритма планирования списка.

## 16.4 Алгоритмы программной конвейеризации

### 16.4.2 Алгоритм программной конвейеризации для случая ациклического графа зависимостей по данным

- ◇ **Вход:** вектор ресурсов целевого компьютера  $R = [r_1, r_2, \dots]$ , где  $r_i$  — количество доступных единиц ресурса  $i$ -го вида; граф зависимостей по данным  $G = (N, E)$ : каждая операция  $n \in N$  помечается таблицей резервирования ресурсов  $RT_n$ , а каждое ребро  $e = n_1 \rightarrow n_2$  из  $E$  помечается  $\langle \delta_e, d_e \rangle$ .
- ◇ **Выход:** программно-конвейеризованное расписание  $S$  и интервал между запусками итераций  $T$ .
- ◇ **Метод:** выполнить программу



## 16.4 Алгоритмы программной конвейеризации

### 16.4.2 Алгоритм программной конвейеризации для случая ациклического графа зависимостей по данным

```
main () {
```

$$T_0 = \max_j \left\lceil \frac{\sum_{n,i} RT_n(i, j)}{r_j} \right\rceil;$$

```
for (T = T0, T0 + 1, ..., пока не будут составлены
расписания для всех узлов из N) {
```

```
 RT = пустая таблица резервирования ресурсов с T
 строками;
```

```
 for (каждый узел n ∈ N в приоритетном топологическом
 порядке) {
```

$$s_0 = \max_{e=p \rightarrow n \in E} (S(p) + d_e);$$

```
 for (s = s0, s0 + 1, ..., s0 + T - 1) {
```

```
 if (NodeScheduled (RT, T, n, s)) break;
```

```
 if (n не может быть спланировано в RT)
 break;
```

```
 } }
```

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.2 Алгоритм программной конвейеризации для случая ациклического графа зависимостей по данным**

```
NodeScheduled (RT, T, n, s) {
 RT' = RT;
 for (каждая строка r из RTn)
 RT'[(s + i) mod T] = RT'[(s + i) mod T] + RTn[i];
 if (для всех i RT'(i) < R) {
 RT = RT';
 S(n) = s;
 return true;
 };
 else return false;
}
```

## 16.4 Алгоритмы программной конвейеризации

### 16.4.2 Алгоритм программной конвейеризации для случая ациклического графа зависимостей по данным

- ◇ Сначала алгоритм находит границу интервала между запусками итераций  $T_0$  на основе требований к ресурсам операций в графе. Затем он пытается найти программно-конвейеризованное расписание, начиная с  $T_0$  в качестве целевого интервала между запусками итераций. Если для текущего значения  $T$  построить расписание не удастся, алгоритм повторяется с большим значением интервала  $T$ .
- ◇ При каждой попытке алгоритм применяет подход из алгоритма планирования списка. Этот подход использует модульное резервирование ресурсов для отслеживания запросов ресурсов в устойчивом состоянии. Операции планируются в топологическом порядке, так что **зависимости по данным всегда можно удовлетворить соответствующими задержками операций**. Для планирования операции мы сначала находим нижнюю границу  $S_0$ , соответствующую ограничениям зависимостей по данным.

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.2 Алгоритм программной конвейеризации для случая ациклического графа зависимостей по данным**

- ◇ Затем вызывается функция `NodeScheduled`, проверяющая возможные конфликты ресурсов в устойчивом состоянии.
- ◇ Если конфликт существует, алгоритм пытается спланировать операцию на следующий такт.
- ◇ Если выясняется, что операция вызывает конфликт для  $\Gamma$  последовательных тактов, то в силу модульной природы обнаружения конфликтов, связанных с ресурсами, последующие попытки будут гарантированно неуспешными, так что следует переходить к другому значению интервала между запусками итераций.
- ◇ Эвристика, состоящая в планировании операций на как можно более ранние моменты времени, стремится минимизировать длину плана итерации. Однако планирование операции на как можно более ранний срок может привести к росту времени жизни некоторых переменных. Например, загрузка данных при этом может оказаться выполненной задолго до того, как потребуются загружаемые данные.

## ***16.4 Алгоритмы программной конвейеризации***

### **16.4.2 Алгоритм программной конвейеризации для случая ациклического графа зависимостей по данным**

- ◇ Одна из простейших эвристик состоит в планировании графа зависимости данных в обратном направлении, поскольку обычно загрузок существенно больше, чем сохранений.

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.3 Планирование графов с циклическими зависимостями**

- ◇ Наличие обратных ребер в графе зависимостей по данным существенно усложняет программную конвейеризацию. При планировании операций в ациклическом графе в топологическом порядке зависимости по данным у планируемых операций могут указать только нижнюю границу размещения каждой операции. В результате всегда можно удовлетворить ограничения, связанные с зависимостями по данным, путем задержки операций.
- ◇ Но концепция топологического порядка неприменима к графам с циклами (обратными ребрами). В самом деле, для данной пары операций в цикле размещение одной из них указывает как нижнюю, так и верхнюю границы для размещения второй.

## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

◇ Пусть

$n_1$  и  $n_2$  – две операции, связанные циклической зависимостью по данным,

$S$  – расписание (список тактов выдачи операций), полученное с помощью программной конвейеризации,

$T$  – интервал между запусками итераций цикла для данного расписания.

Ребро зависимости  $n_1 \rightarrow n_2$  с меткой  $(\delta_1, d_1)$  накладывает на

$S(n_1)$  и  $S(n_2)$  следующее ограничение:

$$(\delta_1 \times T) + S(n_2) - S(n_1) > d_1$$

Аналогично ребро зависимости  $n_2 \rightarrow n_1$  с меткой  $(\delta_2, d_2)$  накладывает ограничение

$$(\delta_2 \times T) + S(n_1) - S(n_2) > d_2$$

Таким образом,

$$S(n_1) + d_1 - (\delta_1 \times T) \leq S(n_2) \leq S(n_1) - d_2 + (\delta_2 \times T)$$

## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

◇ *Сильно связанным компонентом (ССК) графа* называется множество узлов, в котором каждый узел компонента достижим из любого другого узла компонента.

Составление расписания для одного узла в сильно связанном компоненте ограничивает время каждого другого узла компонента как снизу, так и сверху.

Транзитивно, если существует путь  $p$ , ведущий из  $n_1$  в  $n_2$ , то

$$S(n_2) - S(n_1) \geq \sum_{e \in p} (d_e - (\delta_e \times T)) \quad (1)$$



## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

- ◇ Отметим следующее.
- ◇ **Сумма всех  $S$  вдоль цикла должна быть положительной.** Если бы она была равна 0 или отрицательна, это означало бы, что либо операция в цикле должна предшествовать самой себе, либо все операции цикла должны выполняться одновременно, в одном и том же такте.
- ◇ **Расписание выдачи операций внутри итерации одинаково для всех итераций;** это требование, по сути, и означает «программную конвейеризацию». В результате сумма задержек (вторых компонентов меток ребер в графе зависимостей по данным) в цикле представляет собой нижнюю границу интервала  $T$  между запусками итераций.

$$T \geq \sum_{e \in p} d_e$$

## 16.4 Алгоритмы программной конвейеризации

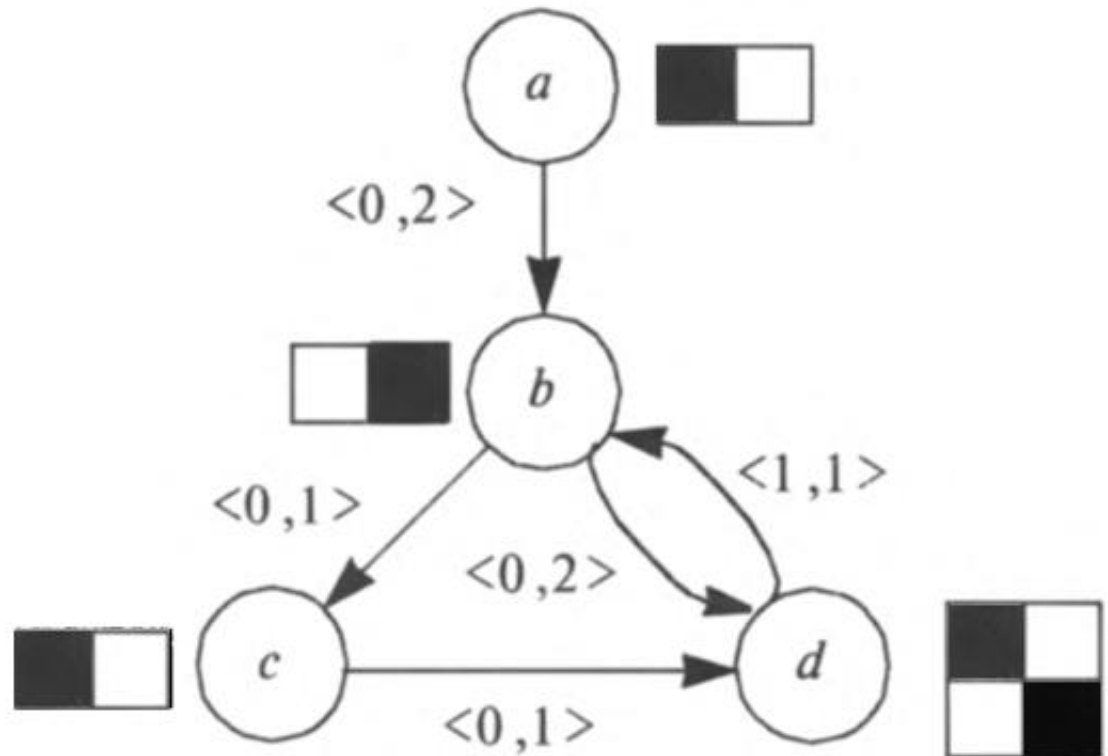
### 16.4.3 Планирование графов с циклическими зависимостями

- ◇ Комбинируя эти два наблюдения, можно увидеть, что для любого допустимого интервала между запусками итераций  $T$  значение правой части неравенства (1) должно быть отрицательно или равно 0, если  $p$  представляет собой цикл.  
В результате **наиболее строгие ограничения на размещение узлов получаются из путей, не содержащих циклы.**
- ◇ Таким образом, **для каждого допустимого  $T$  вычисление транзитивного влияния зависимостей по данным на каждую пару узлов эквивалентно поиску длины наибольшего простого пути от первого узла ко второму.**  
Более того, поскольку циклы не могут увеличивать длину пути, для поиска наидлиннейших путей без требования «простоты пути» можно воспользоваться простым **алгоритмом динамического программирования** и быть уверенным, что полученные длины будут также длинами наидлиннейших простых путей.

## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

- ◇ **Пример.** На рисунке показан граф зависимости данных с четырьмя узлами:  $a$ ,  $b$ ,  $c$  и  $d$ . Возле каждого узла изображена его таблица резервирования ресурсов, а возле каждого ребра – разность итераций  $\delta$  и задержка  $d$ .



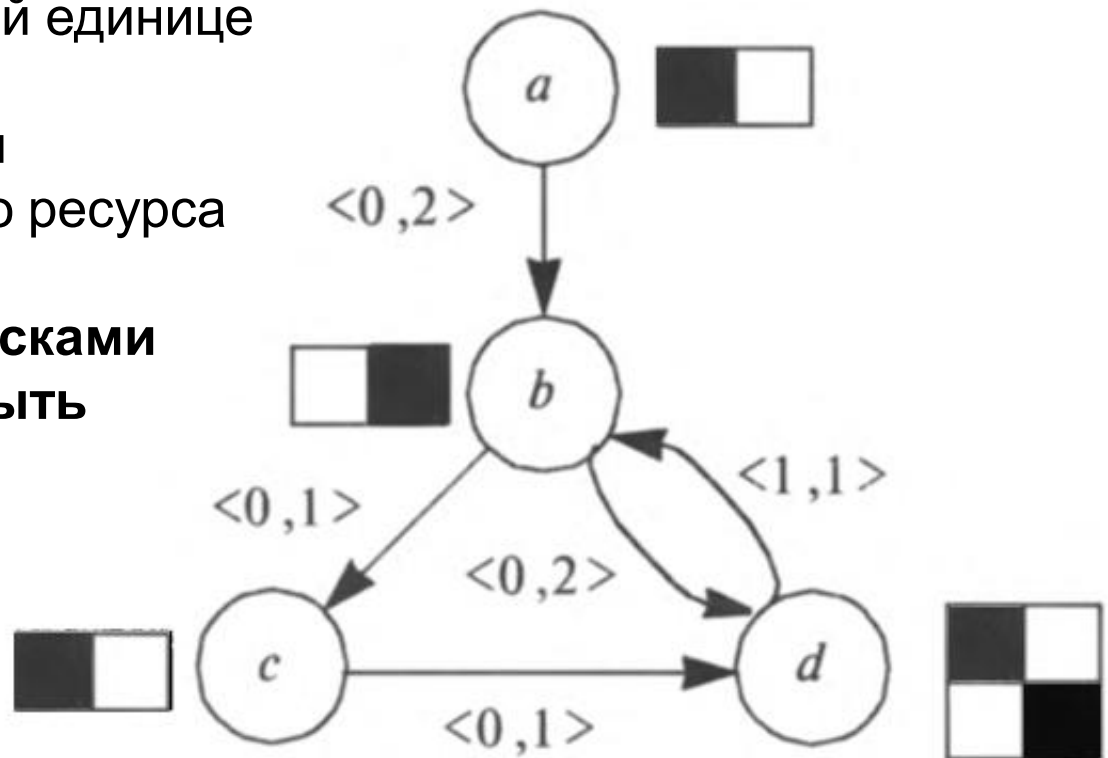
## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

◇ **Пример.** На рисунке показан граф зависимости данных с четырьмя узлами:  $a$ ,  $b$ ,  $c$  и  $d$ . Возле каждого узла изображена его таблица резервирования ресурсов, а возле каждого ребра – разность итераций  $\delta$  и задержка  $d$ .

◇ Будем считать, что целевая машина имеет по одной единице каждого ресурса. Поскольку имеется три использования первого ресурса и два – второго, интервал между запусками итераций не может быть меньше трех тактов:

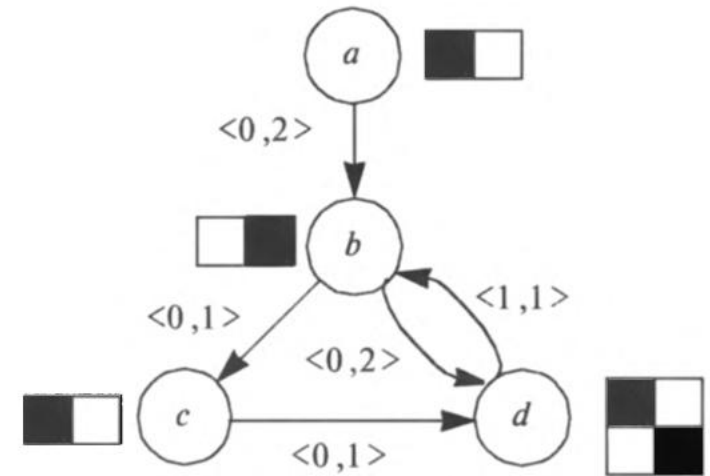
$$T \geq 3$$



## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

- ◇ В данном графе имеется два сильно связанных компонента:
  - ◇ первый, тривиальный, состоит из единственного узла  $a$ ,
  - ◇ второй – из узлов  $b$ ,  $c$  и  $d$ .



- ◇ Самый длинный цикл,  $(b, c, d, b)$ , имеет общую задержку, равную трем тактам в пределах одной итерации.  
Таким образом, **нижняя граница интервала между запусками итераций, основанная на ограничениях циклической зависимости по данным, также равна трем тактам.**

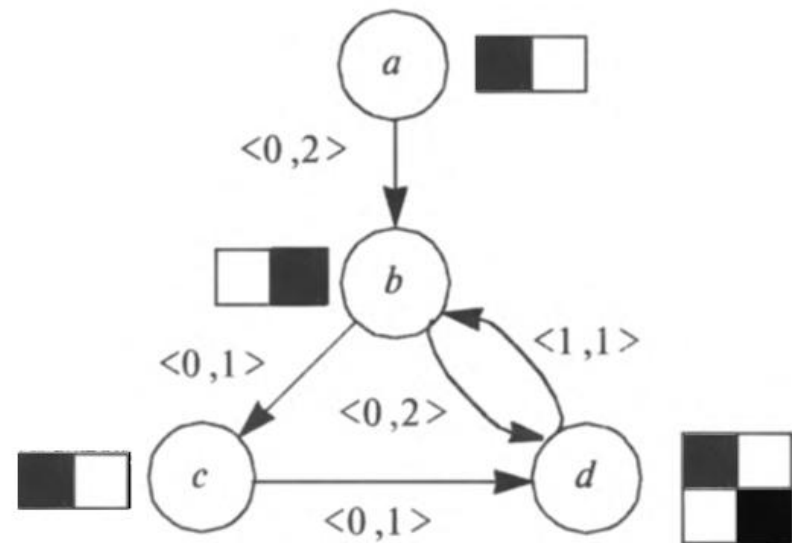
## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

Рассмотрим длины всех путей между вершинами графа.

Сначала рассмотрим простые пути.

Нетрудно показать, что длины самых длинных простых путей зависят от  $T$ .



В таблице указаны метки  $\langle \delta, d \rangle$  ребер графа зависимостей, причем значения  $d$  указаны явно, а значения  $\delta$  как  $-\delta \times T$ :

2 в клетке  $(a, b)$  означает

$2 - 0 \times T$ , т.е.  $\langle 0, 2 \rangle$

$1 - T$  в клетке  $(d, b)$  означает

$1 - 1 \times T$ , т.е.  $\langle 1, 1 \rangle$

|     | $a$ | $b$     | $c$ | $d$ |
|-----|-----|---------|-----|-----|
| $a$ |     | 2       |     |     |
| $b$ |     |         | 1   | 2   |
| $c$ |     |         |     | 1   |
| $d$ |     | $1 - T$ |     |     |

## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

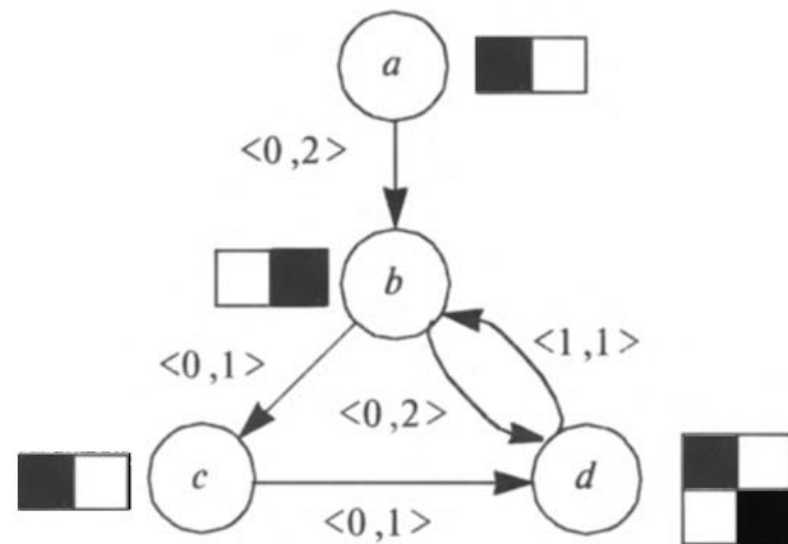
|     | $a$ | $b$     | $c$     | $d$ |
|-----|-----|---------|---------|-----|
| $a$ |     | 2       | 3       | 4   |
| $b$ |     |         | 1       | 2   |
| $c$ |     | $2 - T$ |         | 1   |
| $d$ |     | $1 - T$ | $2 - T$ |     |

$T=3$

|     | $a$ | $b$ | $c$ | $d$ |
|-----|-----|-----|-----|-----|
| $a$ |     | 2   | 3   | 4   |
| $b$ |     |     | 1   | 2   |
| $c$ |     | -1  |     | 1   |
| $d$ |     | -2  | -1  |     |

$T=4$

|     | $a$ | $b$ | $c$ | $d$ |
|-----|-----|-----|-----|-----|
| $a$ |     | 2   | 3   | 4   |
| $b$ |     |     | 1   | 2   |
| $c$ |     | -2  |     | 1   |
| $d$ |     | -3  | -2  |     |



В таблицах показаны длины самых длинных простых путей между вершинами графа, если эти пути существуют:  
в верхней для произвольного  $T$ , в нижних для значений  $T = 3$  и  $T = 4$ .

# 16.4 Алгоритмы программной конвейеризации

## 16.4.3 Планирование графов с циклическими зависимостями

|          | <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> |
|----------|----------|----------|----------|----------|
| <i>a</i> |          | 2        |          |          |
| <i>b</i> |          |          |          |          |
| <i>c</i> |          | $2 - T$  |          | 1        |
| <i>d</i> |          | $1 - T$  | $2 - T$  |          |

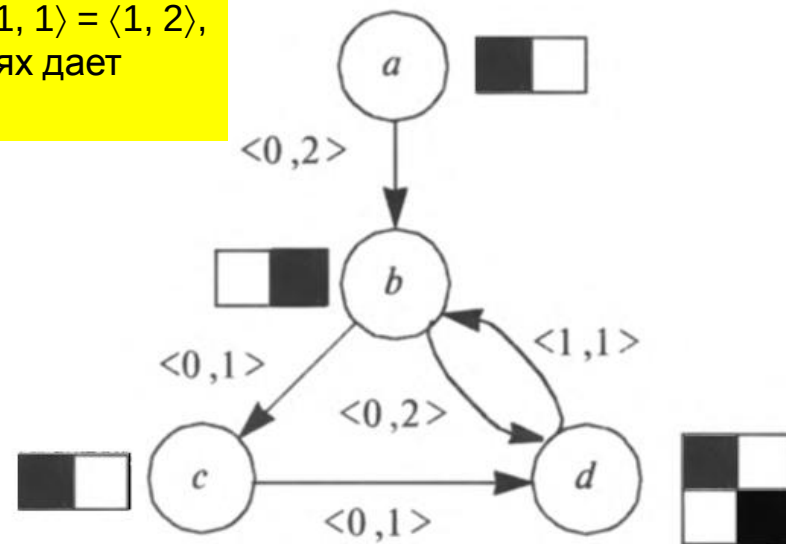
Например,  $(c, b) = (c, d) + (d, b)$ ;  
 суммарная пометка  $\langle 0, 1 \rangle + \langle 1, 1 \rangle = \langle 1, 2 \rangle$ ,  
 что в принятых обозначениях дает  
 $2 - 1 \cdot T = 2 - T$

$T=3$

|          | <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> |
|----------|----------|----------|----------|----------|
| <i>a</i> |          | 2        | 3        | 4        |
| <i>b</i> |          |          | 1        | 2        |
| <i>c</i> |          | -1       |          | 1        |
| <i>d</i> |          | -2       | -1       |          |

$T=4$

|          | <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> |
|----------|----------|----------|----------|----------|
| <i>a</i> |          | 2        | 3        | 4        |
| <i>b</i> |          |          | 1        | 2        |
| <i>c</i> |          | -2       |          | 1        |
| <i>d</i> |          | -3       | -2       |          |



В таблицах показаны длины самых длинных простых путей между вершинами графа, если эти пути существуют:  
 в верхней для произвольного  $T$ , в  
 нижних для значений  $T = 3$  и  $T = 4$ .



## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

- ◇ Например, рассмотрим значение  $2 - T$  наидлиннейшего (простого) пути от  $c$  к  $b$ . Это путь  $(c, d, b)$ . Общая задержка вдоль этого пути два такта, а сумма значений  $\delta$  равна 1, откуда следует, что номер итерации должен увеличиться на 1.

$T$  – продолжительность промежутка между двумя итерациями, значит узел  $b$  должен быть назначен на такт, находящийся как минимум через  $2 - T$  такта после такта, на который назначен узел  $c$ . Следовательно, так как  $T \geq 3$ ,  $b$  может быть выполнен не раньше, чем за  $T - 2$  такта до  $c$ .

- ◇ Заметим, что **рассмотрение непростых путей от  $c$  к  $b$  не дает более строгих ограничений.**

К пути  $(c, d, b)$  можно добавить любое количество итераций цикла  $(d, b)$ . Добавив  $k$  циклов, получим длину пути  $2 - T + k(3 - T)$ , так как общая задержка вдоль пути равна 3, а сумма значений  $\delta$  равна 1. И так как  $T \geq 3$ , то  $2 - T + k(3 - T) \leq 2 - T$ , т.е. расстояние между тактами  $c$  и  $b$  снова оценивается величиной  $2 - T$ .

## 16.4 Алгоритмы программной конвейеризации

### 16.4.3 Планирование графов с циклическими зависимостями

- ◇ Оценка самого длинного пути позволяет легко вычислить корректное место для узла, исходя из зависимостей по данным. Например, рассматривая  $(b, c)$  и  $(c, b)$  можно заметить, что

$$S(c) - S(b) \geq 1$$

$$S(b) - S(c) \geq 2 - T$$

Откуда следует, что

$$S(b) + 1 \leq S(c) \leq S(b) - 2 + T$$

Если  $T = 3$ , то последняя пара неравенств имеет вид:

$$S(b) + 1 \leq S(c) \leq S(b) + 1,$$

что означает, что узел  $c$  должен быть назначен на следующий такт после узла  $b$ .

Если  $T = 4$ , получаем:

$$S(b) + 1 \leq S(c) \leq S(b) + 2,$$

т.е. узел  $c$  должен быть назначен на один или два такта после  $b$ .

Итак, при  $T = 3$  есть всего один выбор, с ростом  $T$  количество возможных вариантов возрастает.

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.4 Структура алгоритма программной конвейеризации**

- ◇ Алгоритм программной конвейеризации для графов с циклами (АКЦ) имеет высокоуровневую структуру, аналогичную структуре алгоритма программной конвейеризации для ациклических графов (АКА).
- ◇ Минимальный интервал между запусками итераций в случае АКЦ ограничен не только требованиями к ресурсам, но и циклами в графе зависимостей по данным.
- ◇ Составление расписания для графа с циклами выполняется путем поочередного составления расписаний для его сильно связанных компонентов (ССК).
- ◇ Если рассматривать каждый ССК как отдельный модуль, то ребра между этими модулями обязательно образуют ациклический граф. Следовательно, АКЦ может планировать ССК в топологическом порядке.  
Как и раньше, если алгоритм не в состоянии спланировать все ССК, интервал между запусками итераций увеличивается и попытка планирования повторяется.  
Так что в случае ациклического графа зависимостей по данным 74 алгоритмы АКЦ и АКА совпадают.

## 16.4 Алгоритмы программной конвейеризации

### 16.4.4 Структура алгоритма программной конвейеризации

- ◇ АКЦ вычисляет дополнительные множества ребер:
  - ◇ множество  $E'$  всех ребер, у которых разность итераций  $\delta = 0$
  - ◇ множества  $E^*$  ребер, составляющих наидлиннейший путь через все вершины; множество  $E^*$  вычисляется для каждого значения интервала между запусками итераций  $T$ .
- ◇ Для каждой пары узлов  $(n, p)$  существует ребро  $e \in E^*$ , с которым связано значение  $d_e$ , равное длине наидлиннейшего простого пути от  $p$  к  $n$ , что обеспечивает существование как минимум одного пути от  $p$  к  $n$ .
- ◇ Множество  $E^*$  можно вычислить однократно (как в примере) с использованием символа  $T$ , а затем для каждой итерации подставлять вместо  $T$  конкретные значения.

## **16.4 Алгоритмы программной конвейеризации**

### **16.4.4 Структура алгоритма программной конвейеризации**

- ◇ АКЦ использует возвраты. Если он не в состоянии спланировать ССК, он пытается спланировать его на такт позже. Эти попытки продолжаются до достижения  $T$  тактов.
- ◇ Возврат важен в связи с тем, что, как было показано в примере, размещение первого узла ССК может полностью определять расписание для других узлов.  
Если составленное расписание не согласуется с расписание, уже разработанным для других ССК к этому моменту, попытка составления расписания признается неудачной.

## 16.4 Алгоритмы программной конвейеризации

### 16.4.5 Алгоритм программной конвейеризации

◇ **Вход:**

- ◇ вектор ресурсов целевой машины  $R = [r_1, r_2, \dots]$ , где  $r_i$  – количество доступных единиц ресурса  $i$ -го вида
- ◇ граф зависимостей по данным  $G = (N, E)$ .

Каждая операция  $n \in N$  помечается своей таблицей резервирования ресурсов  $RT_n$ ; каждое ребро  $e = n_1 \rightarrow n_2$  из  $E$  имеет метку  $\langle \delta_e, d_e \rangle$ , указывающую, что операция  $n_2$  должна выполняться не ранее чем через  $d_e$  тактов после операции  $n_1$  из  $\delta_e$ -й предшествующей итерации.

◇ **Выход:**

- ◇ программно конвейеризованный план  $S$
- ◇ интервал между запусками итераций  $T$ .

◇ **Метод:** выполнить программу, приведенную на следующих слайдах.

# 16.4 Алгоритмы программной конвейеризации

## 16.4.5 Алгоритм программной конвейеризации

```
main () {
```

```
 E' = {e | e ∈ E, δe = 0};
```

$T_0$  – начальное значение интервала между запусками итераций

$$T_0 = \max \left( \max_j \left\lceil \frac{\sum_{n,j} RT_n(i,j)}{r_j} \right\rceil, \max_{C \text{ - cycle\_in\_} G} \left\lceil \frac{\sum_{e \in C} d_e}{\sum_{e \in C} \delta_e} \right\rceil \right)$$

```
 for (T = T0, T0 + 1, ... пока все ССК в G не будут спланированы) {
```

```
 RT = пустая таблица резервирования с T строками;
```

```
 E* = AllPairsLongestPath (G, T);
```

```
 for (каждый ССК C из G в приоритетном топологическом порядке) {
```

```
 for (все n из C)
```

```
 S0 = maxe=p→n из E*, узел p спланирован (S(P) + de);
```

```
 first = некоторое n, такое, что s0(n) является минимумом;
```

```
 s0 = s0(first);
```

```
 for (s=s0; s<s0+T; s=s+1)
```

```
 if (SccScheduled(RT, T, C, first, s)) break;
```

```
 if (C не может быть спланирован в RT) break;
```

```
 }
```

```
 }
```

```
 }
```

```
 SccScheduled(RT, T, C, first, s) {
```

```
 RT' = RT;
```

```
 if (not NodeScheduled (RV, T, first, s)) return false;
```

```
 for (каждый остающийся n из C в приоритетном топологическом порядке ребер из E') {
```

```
 Sl = maxe=n'→n из E*, n'∈C узел n' спланирован (S(n') + de - (δe × T));
```

```
 Su = mine=n'→n из E*, n'∈C узел n' спланирован (S(n') - de + (δe × T));
```

```
 for (s = sl; s ^ min (su, s; + T - 1); s = s + 1)
```

```
 if (NodeScheduled (RT', T, n, s)) break;
```

```
 if (n не может быть спланирован в RT') return false;
```

```
 }
```

```
 RT = RT';
```

```
 return true;
```

```
 }
```

## 16.4 Алгоритмы программной конвейеризации

### 16.4.5 Алгоритм программной конвейеризации

```
main () {
```

```
 E' = {e | e ∈ E, δe = 0};
```

$$T_0 = \max \left( \max_j \left\lceil \frac{\sum_{n,i} RT_n(i,j)}{r_j} \right\rceil, \max_{C\text{-cycle\_in\_}G} \left\lceil \frac{\sum_{e \in C} d_e}{\sum_{e \in C} \delta_e} \right\rceil \right)$$

```
 for (T = T0, T0 + 1, ... пока все ССК в G не будут спланированы) {
```

```
 RT = пустая таблица резервирования с T строками;
```

```
 E* = AllPairsLongestPath (G, T);
```

```
 for (каждый ССК C из G в приоритетном топологическом порядке) {
```

```
 for (все n из C)
```

```
 S0 = maxe=p→n из E*, узел p спланирован (S(P) + de);
```

```
 first = некоторое n, такое, что s0(n) является минимумом;
```

```
 s0 = s0(first);
```

```
 for (s=s0; s<s0+T; s=s+1)
```

```
 if (SccScheduled(RT, T, C, first, s)) break;
```

```
 if (C не может быть спланирован в RT) break;
```

```
 }
```

```
 }
```

```
 }
```



## 16.4 Алгоритмы программной конвейеризации

### 16.4.5 Алгоритм программной конвейеризации

```
SccScheduled(RT, T, C, first, s) {
 RT' = RT;
 if(not NodeScheduled (RV, T, first, s)) return false;
 for (каждый остающийся n из C в приоритетном
 топологическом порядке ребер из E') {
 $S_l = \max_{e=n' \rightarrow n \text{ из } E^*, n' \in C \text{ узел } n' \text{ спланирован}} (S(n') + d_e - (\delta_e \times T))$;
 $S_u = \min_{e=n' \rightarrow n \text{ из } E^*, n' \in C \text{ узел } n' \text{ спланирован}} (S(n') - d_e + (\delta_e \times T))$;
 for (s = si; s ^ min (su, s; + T - 1); s = s + 1)
 if (NodeScheduled (RT', T, n, s)) break;
 if (n не может быть спланирован в RT') return false;
 }
 RT = RT' ;
 return true;
}
```

## 16.4 Алгоритмы программной конвейеризации

### 16.4.5 Алгоритм программной конвейеризации

- ◇ Для планирования ССК алгоритм определяет самый ранний момент, когда каждый узел компонента может быть спланирован с удовлетворением всех транзитивных зависимостей данных из  $E^*$ .

Затем в качестве первого планируемого ССК *first* выбирается ССК с наиболее ранним временем начала.

Далее алгоритм использует функцию **SccScheduled**, которая пытается составить расписание для ССК с наиболее ранним временем начала выполнения.

Алгоритм делает не более  $T$  попыток с последовательно возрастающим временем начала. Если попытка оказывается неудачной, алгоритм пытается использовать другое значение интервала между запусками итераций.

## 16.4 Алгоритмы программной конвейеризации

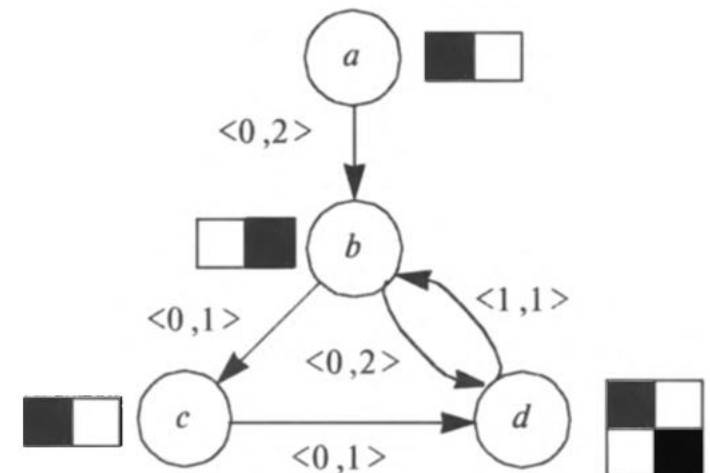
### 16.4.5 Алгоритм программной конвейеризации

- ◇ Алгоритм **SccScheduled** напоминает ранее рассмотренный алгоритм программной конвейеризации ациклических графов зависимостей по данным, но имеет три существенных отличия.
  1. Цель **SccScheduled** состоит в планировании ССК в данный интервал времени  $s$ . Если узел *first* из ССК не может быть спланирован в интервале  $s$ , функция **SccScheduled** возвращает значение *false*. В таком случае функция **main** может, если это потребуется, вновь вызвать **SccScheduled** с более поздним интервалом времени.
  2. Узлы ССК планируются в топологическом порядке, основанном на ребрах из  $E_f$ . Поскольку разность итераций для всех ребер из  $E'$  равна 0, эти ребра не пересекают никаких границ итераций и не могут образовывать циклы. (Ребра, пересекающие границы итераций, называются *циклонесущими*). Верхнюю границу размещения операций указывают только циклонесущие ребра. Данный порядок планирования вместе со стратегией по возможности наиболее раннего планирования каждой операции максимизируют диапазоны, в пределах которых могут планироваться последовательные операции.
  3. Для ССК зависимости указывают как нижнюю, так и верхнюю границы диапазона, в котором может планироваться узел. Функция **SccScheduled** вычисляет этот диапазон и использует его для 82 ограничения попыток планирования.

## 16.4 Алгоритмы программной конвейеризации

### 16.4.6 Пример применения алгоритма программной конвейеризации

- ◇ Применим алгоритм к графу зависимостей по данным на рисунке внизу.
- ◇ Сначала алгоритм вычислит, что в данном примере нижняя граница интервала между запусками итераций равна трем тактам.
- ◇ Заметим, что **достичь этой нижней границы невозможно**.  
В самом деле, когда интервал между запусками итераций  $T = 3$ , транзитивные зависимости требуют выполнения условия  $S(d) - S(b) = 2$ . Планирование узлов  $b$  и  $d$  на расстоянии двух тактов друг от друга приводит к конфликту в таблице модульного резервирования ресурсов длины 3.



## 16.4 Алгоритмы программной конвейеризации

### 16.4.6 Пример применения алгоритма программной конвейеризации

- ◇ На рисунке справа показана работа алгоритма над рассматриваемым графом. Сначала алгоритм пытается построить расписание для  $T = 3$ . Попытка начинается с планирования узлов  $a$  и  $b$  в наиболее ранние допустимые моменты времени. Однако после размещения узла  $b$  в такте 2 узел  $c$  может быть помещен только в такт 3, а это приводит к конфликту с использованием ресурсов узлом  $a$ . И узел  $a$ , и узел  $c$  требуют себе первый ресурс на тактах номер которых кратен 3.

| Попытка | Интервал между запусками итераций | Узел | Диапазон      | План | Модульное резервирование ресурсов                                                     |
|---------|-----------------------------------|------|---------------|------|---------------------------------------------------------------------------------------|
| 1       | $T = 3$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 2    |                                                                                       |
|         |                                   | $c$  | $(3, 3)$      | — —  |                                                                                       |
| 2       | $T = 3$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 3    |                                                                                       |
|         |                                   | $c$  | $(4, 4)$      | 4    |                                                                                       |
|         |                                   | $d$  | $(5, 5)$      | — —  |                                                                                       |
| 3       | $T = 3$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 4    |                                                                                       |
|         |                                   | $c$  | $(5, 5)$      | 5    |                                                                                       |
|         |                                   | $d$  | $(6, 6)$      | — —  |                                                                                       |
| 4       | $T = 4$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 2    |                                                                                       |
|         |                                   | $c$  | $(3, 4)$      | 3    |                                                                                       |
|         |                                   | $d$  | $(4, 5)$      | — —  |                                                                                       |
| 5       | $T = 4$                           | $a$  | $(0, \infty)$ | 0    |   |
|         |                                   | $b$  | $(2, \infty)$ | 3    |                                                                                       |
|         |                                   | $c$  | $(4, 5)$      | 5    |                                                                                       |
|         |                                   | $d$  | $(5, 5)$      | — —  |                                                                                       |
| 6       | $T = 4$                           | $a$  | $(0, \infty)$ | 0    |  |
|         |                                   | $b$  | $(2, \infty)$ | 4    |                                                                                       |
|         |                                   | $c$  | $(5, 6)$      | 5    |                                                                                       |
|         |                                   | $d$  | $(6, 7)$      | 6    |                                                                                       |

## 16.4 Алгоритмы программной конвейеризации

### 16.4.6 Пример применения алгоритма программной конвейеризации

- ◇ Алгоритм выполняет возврат и пытается составить расписание для ССК  $\{b, c, d\}$  на такт позже.
- На этот раз узел  $b$  помещается на такт 3, а узел  $c$  – на такт 4.
- Однако узел  $d$  нельзя поместить на такт 5, так как и узел  $b$ , и узел  $d$  требуют себе второй ресурс на тактах, номер которых кратен 3. Заметим, что тот факт, что оба рассматриваемых конфликта проявляются на тактах, номер которых кратен 3, является совпадением. В других примерах конфликт может проявиться и на других тактах.

| Попытка | Интервал между запусками итераций | Узел                     | Диапазон                                               | План               | Модульное резервирование ресурсов |
|---------|-----------------------------------|--------------------------|--------------------------------------------------------|--------------------|-----------------------------------|
| 1       | $T = 3$                           | $a$<br>$b$<br>$c$        | $(0, \infty)$<br>$(2, \infty)$<br>$(3, 3)$             | 0<br>2<br>— —      |                                   |
| 2       | $T = 3$                           | $a$<br>$b$<br>$c$<br>$d$ | $(0, \infty)$<br>$(2, \infty)$<br>$(4, 4)$<br>$(5, 5)$ | 0<br>3<br>4<br>— — |                                   |
| 3       | $T = 3$                           | $a$<br>$b$<br>$c$<br>$d$ | $(0, \infty)$<br>$(2, \infty)$<br>$(5, 5)$<br>$(6, 6)$ | 0<br>4<br>5<br>— — |                                   |
| 4       | $T = 4$                           | $a$<br>$b$<br>$c$<br>$d$ | $(0, \infty)$<br>$(2, \infty)$<br>$(3, 4)$<br>$(4, 5)$ | 0<br>2<br>3<br>— — |                                   |
| 5       | $T = 4$                           | $a$<br>$b$<br>$c$<br>$d$ | $(0, \infty)$<br>$(2, \infty)$<br>$(4, 5)$<br>$(5, 5)$ | 0<br>3<br>5<br>— — |                                   |
| 6       | $T = 4$                           | $a$<br>$b$<br>$c$<br>$d$ | $(0, \infty)$<br>$(2, \infty)$<br>$(5, 6)$<br>$(6, 7)$ | 0<br>4<br>5<br>6   |                                   |

## 16.4 Алгоритмы программной конвейеризации

### 16.4.6 Пример применения алгоритма программной конвейеризации

- ◇ Алгоритм продолжает работу, откладывая начало расписания для ССК  $\{b, c, d\}$  еще на один такт. Но и эта попытка оказывается неудачной.
- ◇ Тогда алгоритм прекращает попытки составления расписания для  $T = 3$  и начинает попытки составления расписания для  $T = 4$ . В итоге алгоритм находит оптимальное расписание, выполняя шестую попытку.

| Попытка | Интервал между запусками итераций | Узел | Диапазон      | План | Модульное резервирование ресурсов                                                     |
|---------|-----------------------------------|------|---------------|------|---------------------------------------------------------------------------------------|
| 1       | $T = 3$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 2    |                                                                                       |
|         |                                   | $c$  | $(3, 3)$      | — —  |                                                                                       |
| 2       | $T = 3$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 3    |                                                                                       |
|         |                                   | $c$  | $(4, 4)$      | 4    |                                                                                       |
|         |                                   | $d$  | $(5, 5)$      | — —  |                                                                                       |
| 3       | $T = 3$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 4    |                                                                                       |
|         |                                   | $c$  | $(5, 5)$      | 5    |                                                                                       |
|         |                                   | $d$  | $(6, 6)$      | — —  |                                                                                       |
| 4       | $T = 4$                           | $a$  | $(0, \infty)$ | 0    |    |
|         |                                   | $b$  | $(2, \infty)$ | 2    |                                                                                       |
|         |                                   | $c$  | $(3, 4)$      | 3    |                                                                                       |
|         |                                   | $d$  | $(4, 5)$      | — —  |                                                                                       |
| 5       | $T = 4$                           | $a$  | $(0, \infty)$ | 0    |   |
|         |                                   | $b$  | $(2, \infty)$ | 3    |                                                                                       |
|         |                                   | $c$  | $(4, 5)$      | 5    |                                                                                       |
|         |                                   | $d$  | $(5, 5)$      | — —  |                                                                                       |
| 6       | $T = 4$                           | $a$  | $(0, \infty)$ | 0    |  |
|         |                                   | $b$  | $(2, \infty)$ | 4    |                                                                                       |
|         |                                   | $c$  | $(5, 6)$      | 5    |                                                                                       |
|         |                                   | $d$  | $(6, 7)$      | 6    |                                                                                       |

# **17. Параллельное выполнение циклов**



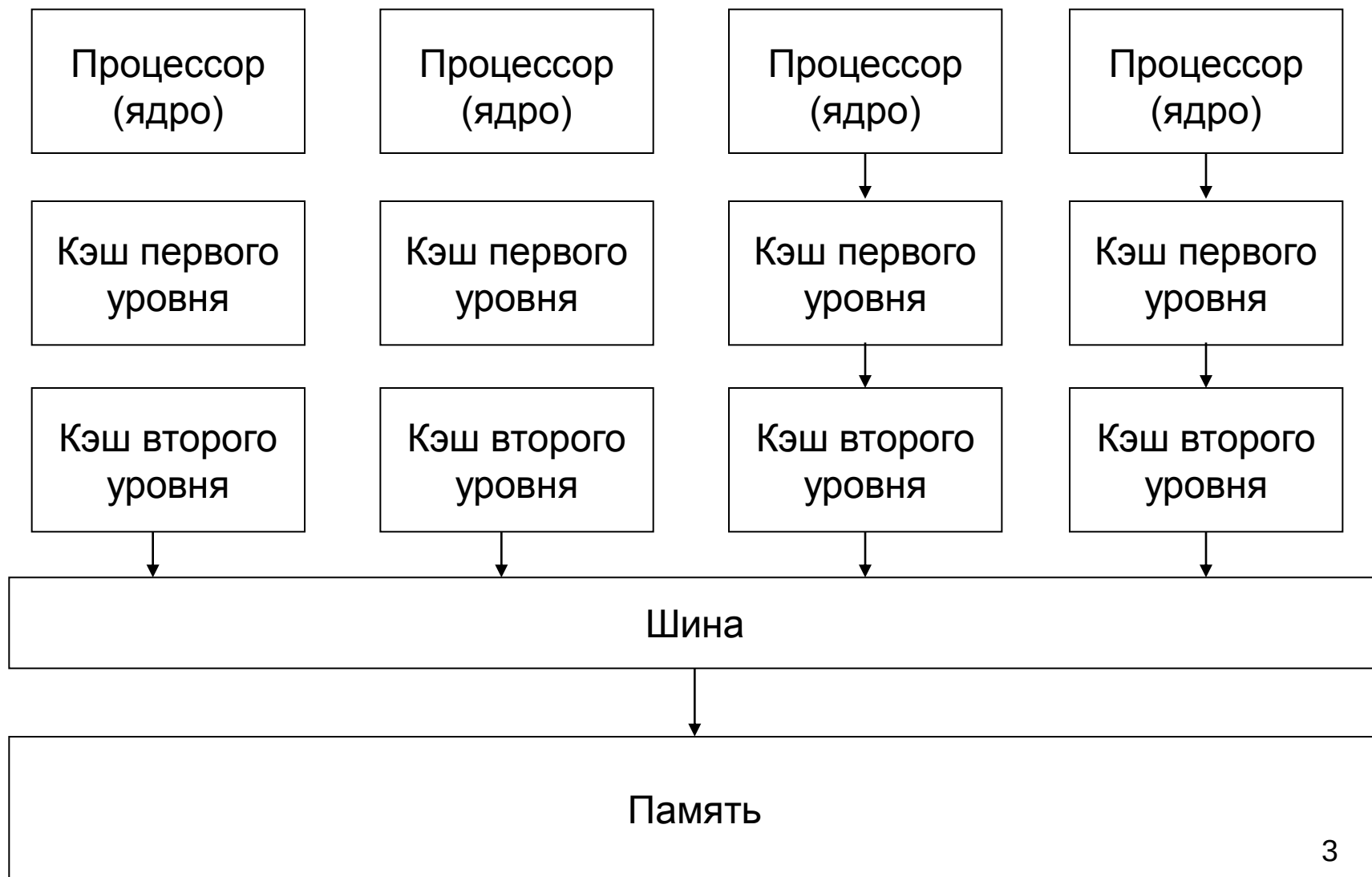
## 17.1 Возможности аппаратуры (обзор)

### 17.1.1. SMP-процессоры

- ◇ Симметричный мультипроцессор с общей памятью (*SMP*).  
Современные персональные компьютеры нередко имеют два процессора, многие серверы имеют четыре, восемь, а иногда десятки процессоров.
- ◇ Стало возможным размещать несколько высокопроизводительных процессоров (ядер) в одном чипе, что привело к еще более широкому распространению мультипроцессоров с общей памятью.
- ◇ Процессоры в *SMP* имеют *общее адресное пространство*.  
Коммуникация (взаимодействие между процессорами по данным) состоит в том, что процессор просто записывает данные в некоторую ячейку памяти, которые после этого считываются другим процессором.
- ◇ *SMP* названы симметричными, потому что все процессоры *SMP* имеют доступ ко всей памяти системы с одинаковым временем доступа.

## 17.1 Возможности аппаратуры (обзор)

### 17.1.1. SMP-процессоры



## 17.1 Возможности аппаратуры (обзор)

### 17.1.1. SMP-процессоры

- ◇ Процессоры (ядра) могут иметь собственные кэши первого, второго, (а иногда и третьего уровня). Кэши самого высокого уровня соединены общей шиной с основной памятью.
- ◇ Между основной памятью и кэшем данные пересылаются блоками, известными под названием *строки кэша*
- ◇ Симметричные мультипроцессоры используют *протокол когерентной кэш-памяти*, чтобы скрыть от программиста наличие кэшей.
  - ◇ При использовании такого протокола несколько процессоров могут одновременно хранить копии одной и той же строки кэша, при условии, что они только читают данные.
  - ◇ Когда процессору требуется записать данные в строку кэша, копии этой строки из всех остальных кэшей удаляются.
  - ◇ Когда процессор требует данные, не размещенные в его кэше, запрос передается общей шине, и данные загружаются либо из памяти, либо из кэша другого процессора.

## 17.1 Возможности аппаратуры (обзор)

### 17.1.1. SMP-процессоры

- ◇ В компьютерах с *SMP*-процессорами межпроцессорное взаимодействие является сравнительно дешевым, – оно не более, чем в два раза медленнее, чем доступ к памяти:  
при обмене данными между процессорами  $P_1$  и  $P_2$  строка кэша процессора  $P_1$ , содержащая эти данные, должна быть записана в память, а потом загружена в кэш процессора  $P_2$ .
- ◇ К сожалению, обмен данными между основной памятью и кэшем – дорогая операция (замедление может достигать сотен раз).
- ◇ Следовательно, чтобы *SMP*-система обеспечивала хорошую производительность, необходимо, чтобы большая часть обрабатываемых данных находилась в кэшах.
- ◇ Многоядерные вычислительные системы имеют похожую архитектуру, за исключением того, что ядра одного процессора имеют общий кэш.

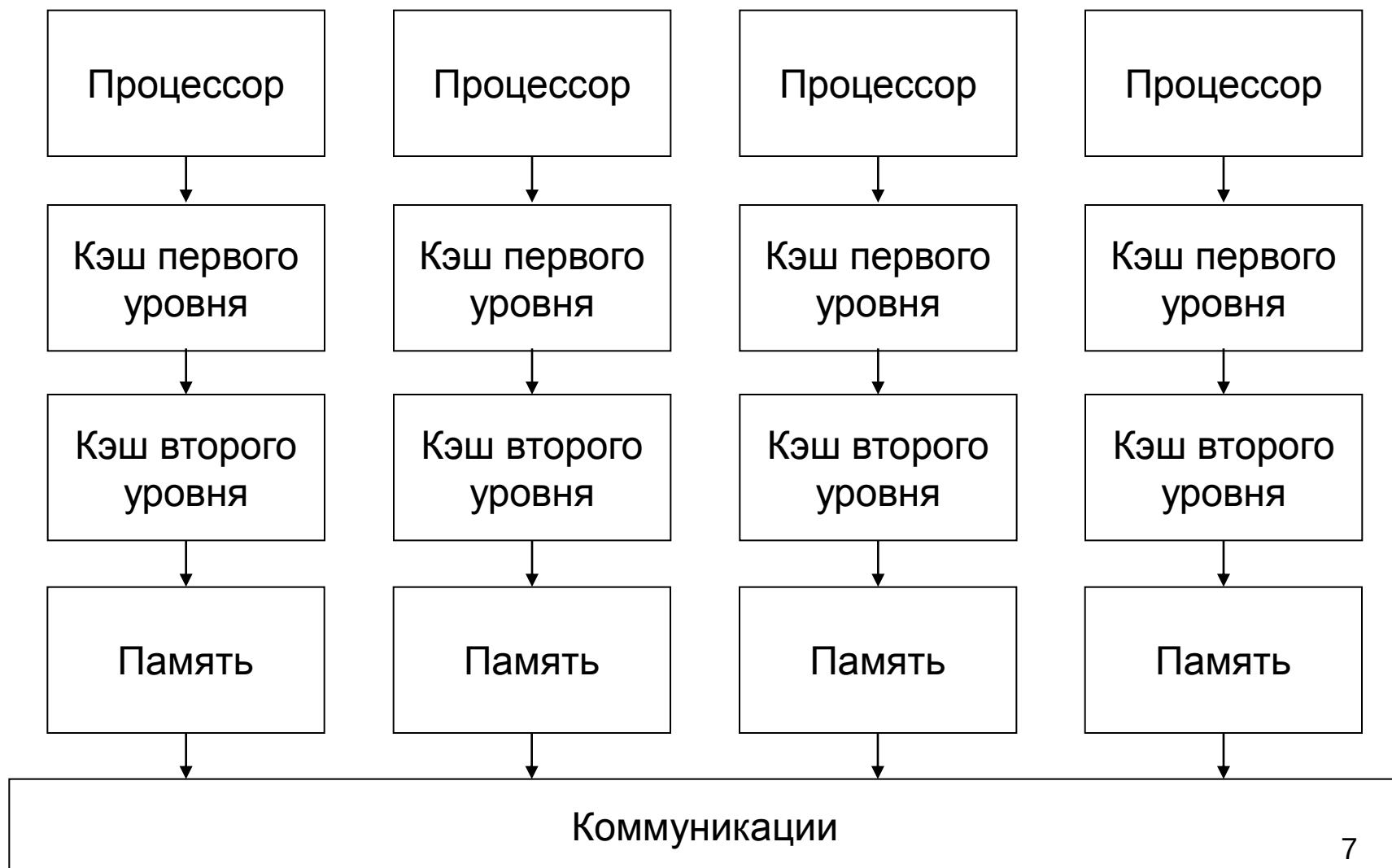
## **17.1 Возможности аппаратуры (обзор)**

### **17.1.2. Многопроцессорные системы с распределенной памятью**

◇ Существует два вида параллельных компьютерных систем с распределенной памятью: системы типа NUMA (nonuniform memory access) и системы, поддерживающие передачу сообщений. Архитектуры NUMA обеспечивают программам работу в общем адресном пространстве, давая возможность процессорам общаться путем чтения и записи в общую память. В системах с передачей данных, однако, процессоры имеют отдельные адресные пространства и общаются, посылая сообщения один другому. Отметим, что хотя программирование для машин с общей памятью проще, и в этом случае программы должны иметь хорошую локальность, чтобы обеспечить эффективность.

## 17.1 Возможности аппаратуры (обзор)

### 17.1.2. Многопроцессорные системы с распределенной памятью



## 17.2 Параллельное выполнение циклов

### 17.2.1. Пример параллельного цикла

◇ Цикл

```
for(i = 0; i < n; i++) {
 z[i] = x[i] - y[i];
 z[i] = z[i] * z[i];
}
```

вычисляет квадраты разностей между элементами векторов  $X$  и  $Y$  и запоминает их значения в векторе  $Z$ .

◇ Цикл допускает параллельное выполнение: его итерации обращаются к разным множествам данных и поэтому могут быть выполнены в произвольном порядке, — результат вычислений не изменится.

## 17.2 Параллельное выполнение циклов

### 17.2.1. Пример параллельного цикла

- ◇ Чтобы выполнить цикл на компьютере с  $M$  процессорами, присвоим каждому процессору уникальный  $ID$  (номер)  $p = 0, 1, \dots, M - 1$ . Разделим исходный цикл на  $M$  одинаковых порций:

```
b = ceil(n/M) ;
for (i = b*p; i < min(n,b*(p+1)) ; i++) {
 z[i] = x[i] - y[i] ;
 z[i] = z[i] * z[i] ;
}
```

- ◇ Итерации цикла распределяются между процессорами равномерно; на  $p$ -ый процессор передается для выполнения  $p$ -ая порция итераций.
- ◇ Таким образом, на всех процессорах, кроме последнего, выполняются одинаковые порции цикла.



## 17.2 Параллельное выполнение циклов

### 17.2.2. Закон Амдаля

- ◇ Параллельный код, приведенный в примере 17.2.1, представляет собой *SPMD*-программу (*Single Program Multiple Data*):  
Все процессоры выполняют одинаковый код, но для каждого процессора этот код параметризован идентификатором, уникальным для этого процессора, так что различные процессоры могут выполнять различные действия.
- ◇ **Закон Амдаля.** Если  $f$  – распараллеленная часть кода, выполняемого на  $p$ -процессорном компьютере без накладных расходов на коммуникации или распараллеливание, то максимальное ускорение программы составляет:

$$Speedup_{Parallel} = \frac{1}{(1 - f) + (f / p)}$$

- ◇ Из формулы, в частности следует:

если  $f = 50\%$ , то  $S_{max} = 2$

если  $f = 90\%$ , то  $S_{max} = 3$  на 4 процессорах

## 17.2 Параллельное выполнение циклов

### 17.2.2. Закон Амдаля

◇ Из формулы Амдаля

$$Speedup_{Parallel} = \frac{1}{(1 - f) + (f / p)}$$

следует:

если  $f = 50\%$ , то  $S_{max} = 2$

если  $f = 90\%$ , то  $S_{max} = 3$  на 4 процессорах

◇ Более того при  $p \rightarrow \infty$   $Speedup_{Parallel} \rightarrow \frac{1}{1 - f}$

откуда, в частности, следует, что при  $f = 90\%$   $Speedup_{Parallel} \rightarrow 10$

## 17.2 Параллельное выполнение циклов

### 17.2.3. Понятие локальности данных

◇ На процессорах с векторными операциями цикл

```
for(i = 0; i < n; i++) {
 z[i] = x[i] - y[i];
 z[i] = z[i] * z[i];
}
```

(код 1)

вычисляющий квадраты разностей между элементами векторов  $X$  и  $Y$  и запоминающий их значения в векторе  $Z$ , разбивается на два цикла

```
for (i = 0; i < n; i++) {
 z[i] = x[i] - y[i];
}
```

и (код 2)

```
for (i = 0; i < n; i++) {
 z[i] = z[i] * z[i];
}
```

Каждый из циклов выполняется с использованием векторных операций.

**Вопрос: Какая из двух программ будет работать быстрее?**

## 17.2 Параллельное выполнение циклов

### 17.2.3. Понятие локальности данных

- ◇ Ответ: (код 1) , так как у него лучше локальность данных.
- ◇ В (код 1) каждая разность возводится в квадрат сразу, как только она получается: можно вычислить разность на регистре, на регистре же возвести ее в квадрат, и один раз записать разность в ячейку памяти  $z[i]$ .
- ◇ (код 2) выбирает  $z[i]$  один раз, а записывает два раза. Более того, если размер массива больше кеша,  $z[i]$  требует повторной перезагрузки из памяти для второго использования в этом примере. Следовательно, (код 2) будет выполняться существенно медленнее, чем (код 1) .
- ◇ При распараллеливании (код 1) значение параметра  $M$  необходимо согласовать с размером кэша.

## 17.2 Параллельное выполнение циклов

### 17.2.3. Понятие локальности данных

- ◇ **Временна́я локальность:** обращение к ячейке памяти означает, что через небольшой промежуток времени к этой ячейке снова будет обращение.
- ◇ **Пространственная локальность:** обращение к ячейке памяти означает, что через небольшой промежуток времени будет обращение к ячейкам, находящимся вблизи этой ячейки.
- ◇ В (код 1) – временная локальность по обращению к  $z[i]$  и пространственная локальность между итерациями цикла, при условии, что  $n$  меньше длины строки кэша.
- ◇ **Как узнать, находится ли данная строка кэша в кэше?** Кэш обладает *k-вариантной множественной ассоциативностью*: каждая строка кэша может находиться только в  $k$  местах кэша, точнее данные с адресом  $n$  могут отображаться только в позиции  $n \bmod (s/k)$ , где  $s$  – размер кэша. У большинства кэшей  $k$  равно от 1 до 8.

## 17.2 Параллельное выполнение циклов

### 17.2.3. Понятие локальности данных

◇ Обнуление двумерного массива по столбцам:

```
for (j = 0; j < n; j++)
 for (i = 0; i < n; i++)
 z[i,j] = 0;
```

(код 1)

◇ Построчное обнуление:

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++)
 z[i,j] = 0;
```

(код 2)

◇ Построчное параллельное обнуление:

```
b = ceil(n/M) ;
for (i = b*p; i < min(n,b*(p+1)) ; i++)
 for (j = 0; j < n; j++)
 z[i,j] = 0;
```

(код 3)

## 17.2 Параллельное выполнение циклов

### 17.2.3. Понятие локальности данных

- ◇ На предыдущем слайде:
  - ◇ (код 2) обладает пространственной локальностью, так как во внутреннем цикле последовательно обрабатываются соседние элементы массива  $Z$
  - ◇ (код 1) не обладает пространственной локальностью так как элементы массива  $Z$ , последовательно обрабатываемые во внутреннем цикле, расположены на расстоянии  $(n + 1)$  один от другого
  - ◇ Вывод: предпочтительнее обнулять массив построчно.
- ◇ Если обнуление ведется по столбцам, то несмотря на то, что каждая строка кэша повторно используется в последовательных итерациях внешнего цикла, строки кэша будут выгружаться пред тем, как будут использованы еще раз, если размер столбца больше, чем размер кэша.
- ◇ Когда циклы допускают параллельное выполнение, их итерации могут выполняться в произвольном порядке;  
***их можно переставлять, чтобы улучшить локальность данных.***

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.1. Основные понятия

- ◇ *Пространство итераций* – это множество динамически выполняемых в процессе вычисления экземпляров итераций, то есть, множество комбинаций значений, принимаемых индексами цикла. Пространство итераций ограничивается нижними и верхними границами интервалов изменения индексов цикла и представляет собой  $d$ -мерное многообразие.
- ◇ *Пространство данных* – это множество элементов массива, к которым осуществляется доступ; это многомерное пространство, размерность которого определяется размерностями массивов, обрабатываемых в цикле.
- ◇ *Пространство процессоров* – это множество процессоров в системе. Обычно этим процессорам присваиваются целочисленные номера, чтобы можно было различать их; иногда это пространство бывает удобно представлять как многомерное, считая процессоры элементами массива.



## 17.3 Формальная постановка задачи распараллеливания циклов

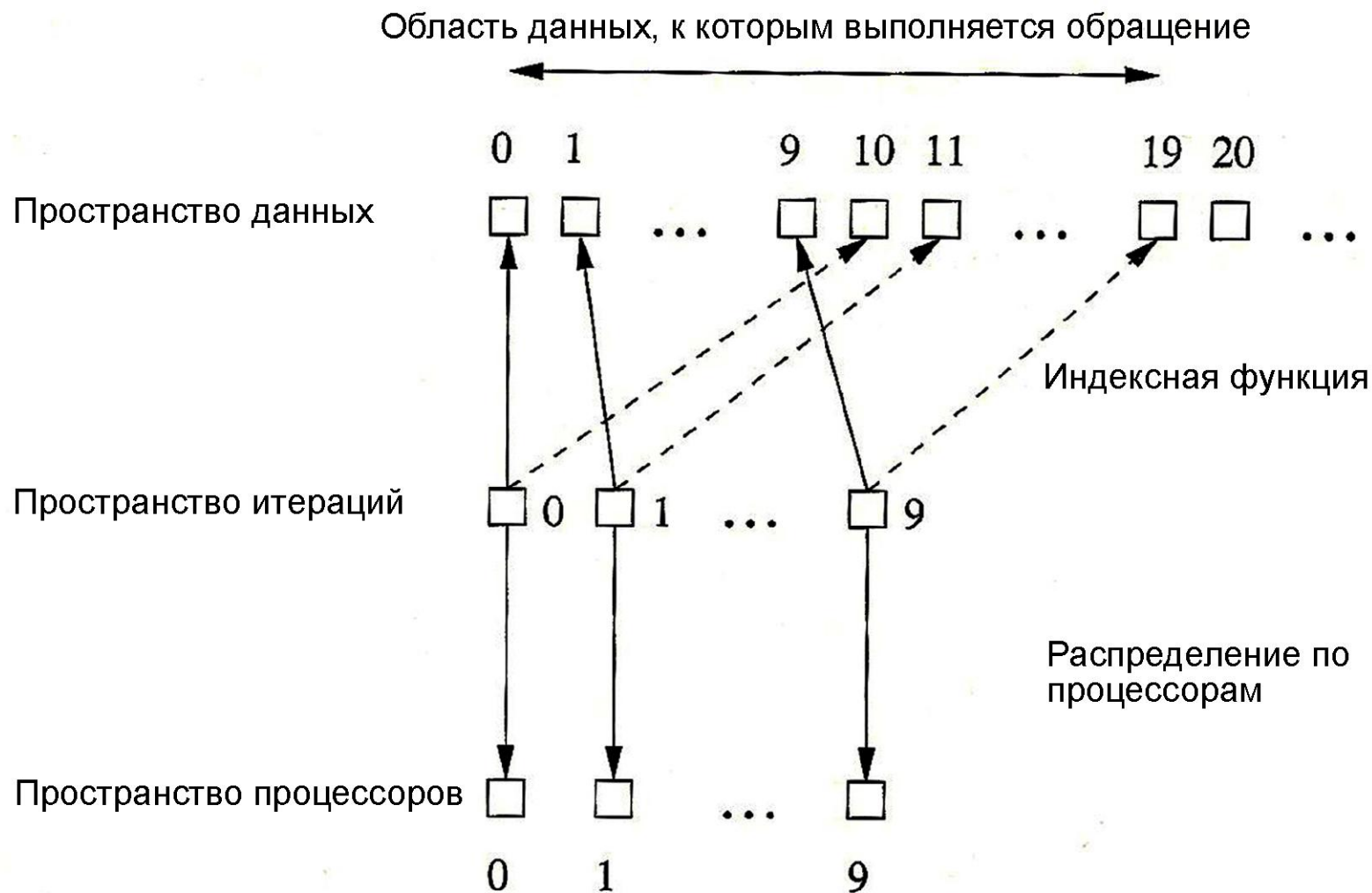
### 17.3.1. Основные понятия

- ◇ *Индексная функция.* Каждое обращение к массиву в коде определяет отображение итерации из пространства итераций на элемент массива в пространстве данных. Индексная функция позволяет определить размерность пространства данных.
- ◇ *Функция разбиения цикла* определяет распределение итераций цикла по процессорам.
- ◇ *Область данных, к которым выполняется обращение.* Чтобы найти наилучшее разбиение цикла, желательно знать область данных, к которым выполняется обращение. Эту область можно получить, объединяя информацию о пространстве итераций с индексной функцией массива.
- ◇ На следующем слайде показаны введенные пространства и отношения между ними для следующей программы:

```
float Z[100];
for (i = 0; i < 10; i++)
 Z[i+10] = Z[i];
```

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.1. Основные понятия



Область данных, к которым выполняется обращение  $Z[i + 10]$  равна  $\{x \mid 10 \leq x < 20\}$ ,  
а область данных, к которым выполняется обращение  $Z[i]$  равна  $\{x \mid 0 \leq x < 10\}$ .

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.1. Основные понятия

- ◇ *Гнездо циклов* глубины  $d$  – это  $d$  циклов, вложенных один в другой. В гнезде циклов определено  $d$  индексных переменных, и пространство итераций гнезда циклов имеет размерность  $d$ .
- ◇ В настоящее время полное решение задачи распараллеливания циклов получено только для циклов, у которых все индексные выражения и границы изменения индексов представляют собой аффинные выражения относительно индексов.
- ◇ Выражение называется *аффинным* по отношению к переменным  $v_1, v_2, \dots, v_n$ , если оно может быть выражено как  $c_0 + c_1 v_1 + \dots + c_n v_n$ , где  $c_0, c_1, \dots, c_n$  – константы. Аффинное выражение, у которого  $c_0 = 0$ , называется линейным.
- ◇ Для рассматриваемого класса циклов все отображения пространств можно выразить через матрицы, а все области данных представляют собой  $d$ -мерные выпуклые многогранники. Это позволяет применять методы линейной алгебры и линейного программирования.

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц

◇ Рассмотрим алгоритм умножения матриц:

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++){
 Z[i,j] = 0.0;
 for (k = 0; k < n; k++)
 Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
 }
```

◇ Алгоритм записан на языке C, но для удобства чтения принято соглашение вместо  $Z[i][j]$  писать  $Z[i,j]$ .

◇ Вычисления каждого элемента  $Z$  независимы, и могут выполняться параллельно.

◇ Этот алгоритм будем называть базовым алгоритмом умножения матриц.

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц

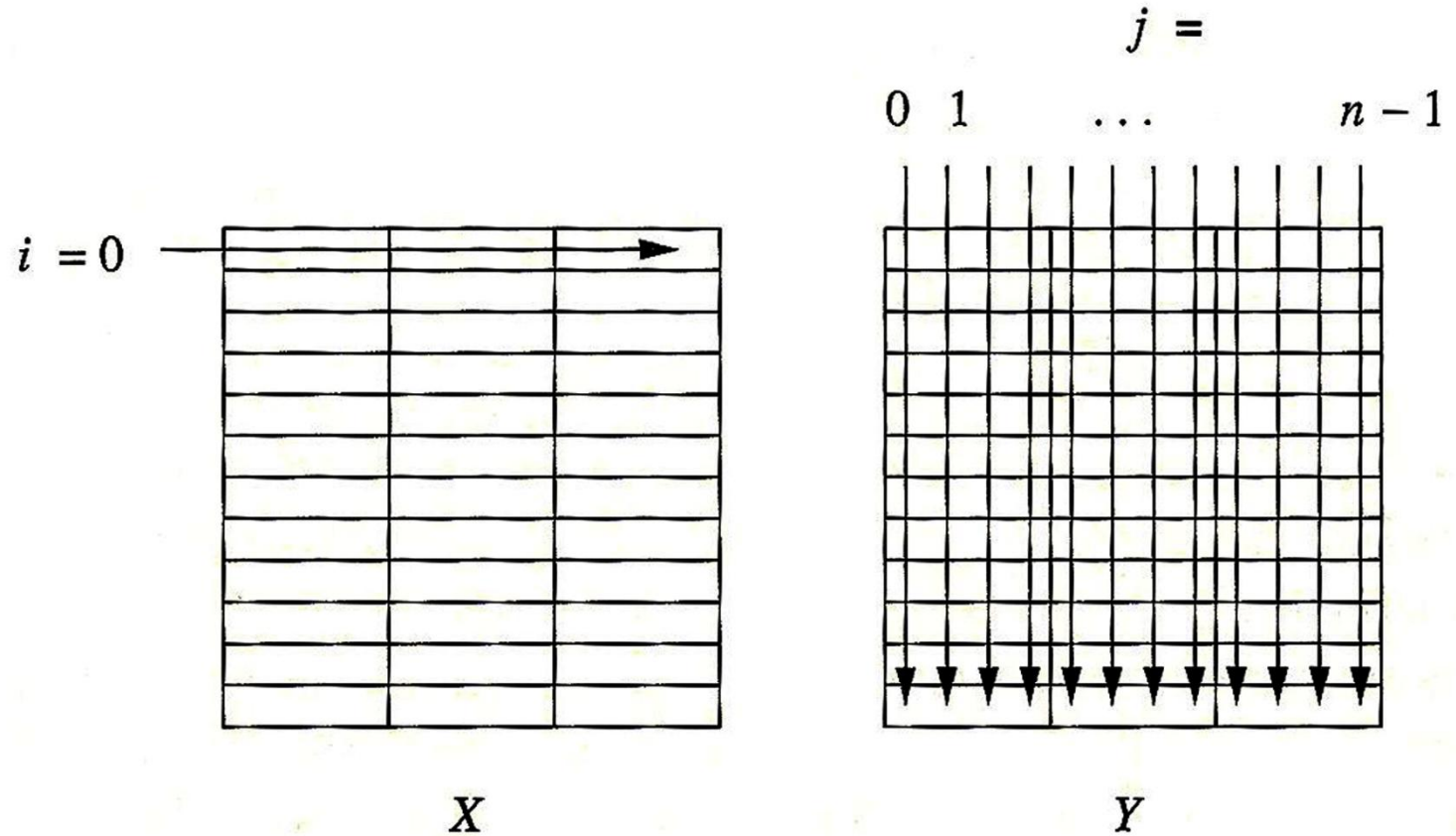
#### ◇ Последовательное выполнение умножения матриц

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++) {
 Z[i,j] = 0.0;
 for (k = 0; k < n; k++)
 Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
 }
```

- ◇ Предположим, что матрицы расположены в памяти по строкам, и что  $c$  – число элементов массив в строке кэша
- ◇ Самый внутренний цикл читает и записывает одни и те же элементы массива  $Z$ , и использует строку массива  $X$  и столбец массива  $Y$ .
- ◇  $Z[i, j]$  легко разместить на регистре, так что он не требует обращений к памяти.
- ◇ Обращения к массиву  $X$  дают небольшую нагрузку на кэш: одна строка  $X$  распределяется на  $n/c$  строк кэша (см. рисунок на следующем слайде).
- ◇ При использовании одной строки массива  $X$ , алгоритм обращается ко всем элементам массива  $Y$ , столбец за столбцом. При этом на одной итерации внешнего цикла произойдет от  $n^2/c$  до  $n^2$  промахов кэша. 22

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц



## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц

#### ◇ Распараллеливание по строкам

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++){
 Z[i,j] = 0.0;
 for (k = 0; k < n; k++)
 Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
 }
```

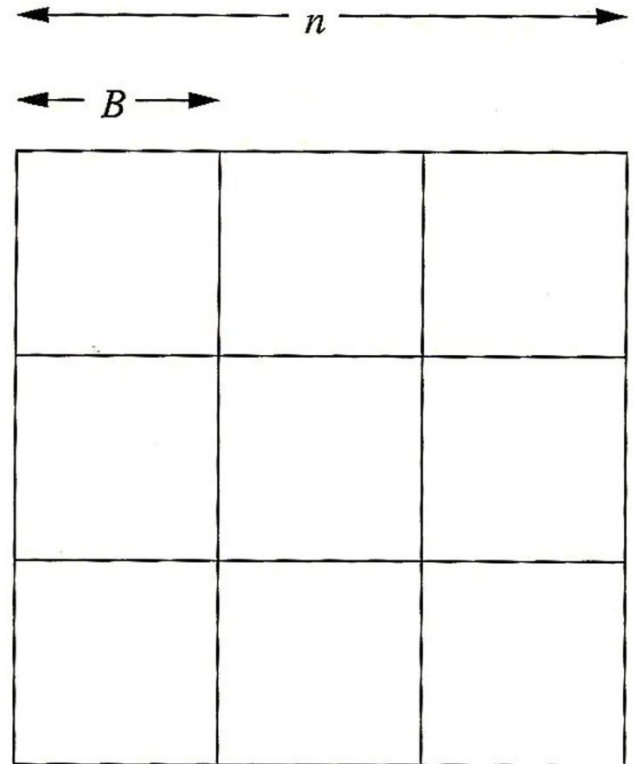
- ◇ Параллельное выполнение внешнего цикла: каждый из  $p$  процессоров отвечает за вычисление  $n/p$  соседних строк массива  $Z$  (для удобства предполагается, что  $n$  делится на  $p$ ).
- ◇ В то время, как время вычислений уменьшается пропорционально  $p$ , стоимость коммуникаций возрастает пропорционально  $p$ : каждый из  $p$  процессоров должен прочесть  $n^2/p$  элементов  $X$  и все  $n^2$  элементов  $Y$ .
- ◇ Когда  $p$  приближается к  $n$ , время вычислений становится  $O(n^2)$  при стоимости коммуникаций  $O(n^3)$ : шина, по которой данные передаются между памятью и кэшами процессоров становится узким местом.
- ◇ Таким образом, в рассматриваемом случае использование в вычислении большого числа процессоров может на самом деле замедлить вычисление, а не ускорить его.

## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц

#### ◇ Разбиение на блоки

- ◇ *Разбиение на блоки* – способ изменения порядка итераций цикла, который может **улучшить локальность программы**.
- ◇ Матрица делится на подматрицы, или *блоки*, как показано на рисунке, и операции упорядочиваются таким образом, чтобы алгоритм вычислял произведение матриц поблочно.
- ◇ Обычно блоки представляют собой квадраты со стороной  $B$ .
- ◇ При корректном выборе  $B$  можно существенно уменьшить число промахов кэша по сравнению с базовым алгоритмом, когда матрицы  $X$ ,  $Y$  и  $Z$  не помещаются в кэше



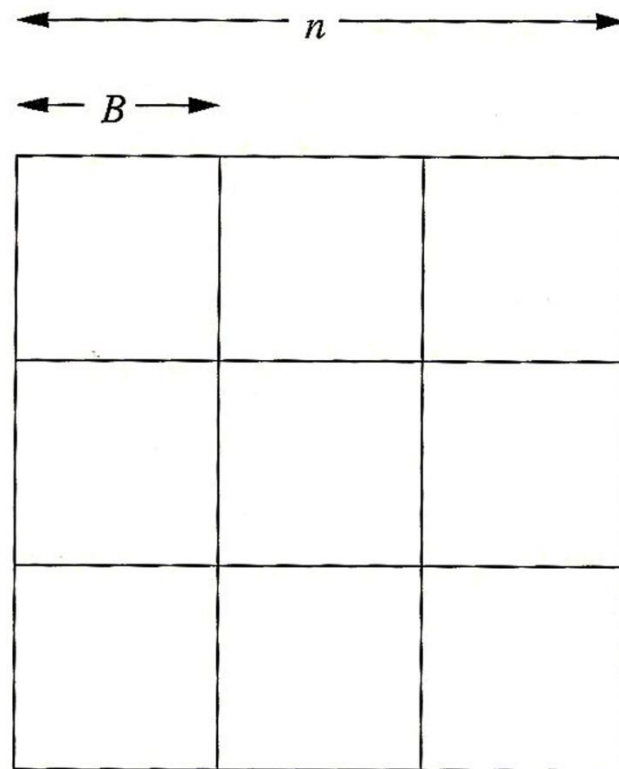


## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц

#### ◇ Разбиение на блоки

- ◇ Выберем  $B$  таким образом, чтобы было возможно поместить один блок каждой матрицы в кэш. Можно показать, что при разбиении на блоки максимальное число промахов кэша равно  $2n^3/Bc$ , т.е. использование разбиения на блоки при умножении матриц приносит большие выгоды:  $2n^3/Bc$  существенно меньше оценки  $n^2/c + n^3$  для распараллеливания по строкам



## 17.3 Формальная постановка задачи распараллеливания циклов

### 17.3.2. Пример. Параллельная программа умножения матриц

◇ **Блокирование: Разбиение на блоки**

◇ Алгоритм умножения матриц с разделением на блоки имеет вид:

```
1) for(ii = 0; ii < n; ii = ii + B)
2) for(jj = 0; jj < n; jj = jj + B)
3) for(kk = 0; kk < n; kk = kk + B)
4) for(i = ii; i < ii + B; i++)
5) for(j = jj; j < jj + B; j++)
6) Z[i, j] = 0;
7) for(i = ii; i < ii + B; i++)
8) for(j = jj; j < jj + B; j++)
9) for(k = kk; k < kk + B; k++)
10) Z[i, j] = Z[i, j] + X[i, k] * Y[k, j];
```

## 17.4 Пространство итераций

### 17.4.1. Построение пространств итераций для гнезд циклов

- ◇ Ограничения на рассматриваемые гнезда циклов:
  - ◇ Каждый цикл в гнезде имеет один индекс, значение которого увеличивается на 1 на каждой итерации. (На примере будет показано, что это предположение не приводит к потере общности).
  - ◇ Границы каждого цикла должны записываться с помощью аффинных выражений от индексов более внешних циклов.

◇ **Пример.** Рассмотрим цикл

```
for (i = 2; i <= 100; i = i+3)
 z[i] = 0;
```

в котором индекс  $i$  увеличивается на 3 на каждом витке цикла.

С помощью замены индекса  $i$  по формуле  $i = 3j + 2$  можно привести этот цикл к виду:

```
for (j = 0; j <= 32; j++)
 z[3*j+2] = 0;
```

Нижнюю границу для  $j$  находим, решая  $3j + 2 = 2$  относительно  $j$ , а для нахождения верхней границы упрощаем  $3j + 2 \leq 100$ .

Получим  $j \leq 32.67$ , откуда следует  $j \leq 32$ , так как  $j$  должно быть целым.

## 17.4 Пространство итераций

### 17.4.1. Построение пространств итераций для гнезд циклов



Гнездо циклов глубины  $d$  моделируется с помощью  $d$ -мерного линейного пространства, измерения которого упорядочены таким образом, что  $k$ -ое измерение представляет  $k$ -ый вложенный цикл, начиная с самого внешнего.

Точка  $(x_1, x_2, \dots, x_d)$  в этом пространстве представляет значения всех индексов цикла:

индекс самого внешнего цикла имеет значение  $x_1$ ,

индекс второго цикла – значение  $x_2$ , и т.д.

индекс самого внутреннего цикла имеет значение  $x_d$ .

## 17.4 Пространство итераций

### 17.4.1. Построение пространств итераций для гнезд циклов

- ◇ Каждая нижняя и верхняя граница цикла определяет неравенство, разделяющее итерационное пространство на два подпространства:
  - то, которое содержит итерации цикла (*положительное подпространство*),
  - и то, которое не содержит итераций цикла (*отрицательное подпространство*).
- ◇ Конъюнкция всех линейных неравенств представляет пересечение всех положительных подпространств, определяя выпуклый многогранник, который и называется *итерационным пространством* гнезда циклов.

## 17.4 Пространство итераций

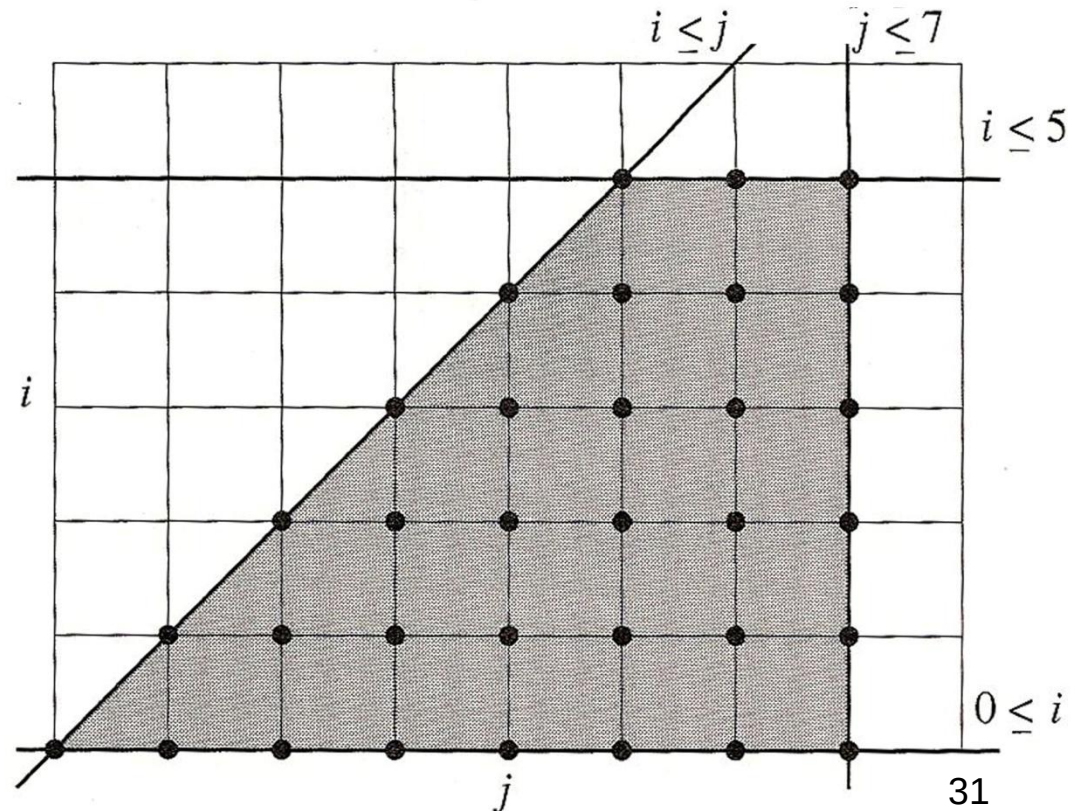
### 17.4.1. Построение пространств итераций для гнезд циклов

◇ **Пример.** Рассмотрим двумерное гнездо циклов:

```
for (i = 0; i <= 5; i++)
 for (j = i; j <= 7; j++)
 z[j,i] = 0;
```

Это гнездо циклов глубины 2 можно смоделировать с помощью двумерного многогранника, показанного на рисунке.

Оси координат представляют значения индексов цикла  $i$  и  $j$ . Индекс  $i$  может принимать любое целое значение между 0 и 5; индекс  $j$  принимать любое целое значение, удовлетворяющее условию  $i < j \leq 7$ .



## 17.4 Пространство итераций

### 17.4.1. Построение пространств итераций для гнезд циклов

- ◇ Последовательное выполнение гнезда циклов выполняет итерации в его итерационном пространстве  
в **возрастающем лексикографическом порядке**.
- ◇ **Определение.** Вектор  $i = [i_0, i_1, \dots, i_n]$  лексикографически меньше, чем другой вектор  $i' = [i'_0, i'_1, \dots, i'_n]$ , записывается  $i \pi i'$ , тогда, и только тогда, когда существует  $m < \min(n, n')$  такое что  $[i_0, i_1, \dots, i_m] = [i'_0, i'_1, \dots, i'_m]$  и  $i_{m+1} < i'_{m+1}$ .  
Отметим, что случай  $m = 0$  возможен, и наиболее типичен.
- ◇ **Пример.** Если  $i$  – индекс внешнего цикла, итерации в гнезде циклов из предыдущего примера выполняются в следующем порядке:

|         |         |         |         |         |         |         |        |
|---------|---------|---------|---------|---------|---------|---------|--------|
| [0, 0], | [0, 1], | [0, 2], | [0, 3], | [0, 4], | [0, 5], | [0, 6], | [0, 7] |
|         | [1, 1], | [1, 2], | [1, 3], | [1, 4], | [1, 5], | [1, 6], | [1, 7] |
|         |         | [2, 2], | [2, 3], | [2, 4], | [2, 5], | [2, 6], | [2, 7] |
|         |         |         | [3, 3], | [3, 4], | [3, 5], | [3, 6], | [3, 7] |
|         |         |         |         | [4, 4], | [4, 5], | [4, 6], | [4, 7] |
|         |         |         |         |         | [5, 5], | [5, 6], | [5, 7] |

## 17.4 Пространство итераций

### 17.4.1. Построение пространств итераций для гнезд циклов

- ◇ Итерации цикла глубины  $d$  можно выразить с помощью формулы

$$\{\mathbf{i} \in Z^d \mid \mathbf{B}\mathbf{i} + \mathbf{b} \geq \mathbf{0}\}$$

где  $Z$  – множество целых чисел,  
 $\mathbf{B}$  – целочисленная матрица размеров  $d \times d$ ,  
 $\mathbf{b}$  – целый вектор длины  $d$ , и  
 $\mathbf{0}$  – вектор из  $d$  нулей.

- ◇ В примере 1.4.1 формула, выражающая итерации цикла, имеет вид:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Выполнив операции в последнем соотношении, получим:

$$i \geq 0, i \leq 5, j \geq i, j \leq 7$$



## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Линейные неравенства для нижних и верхних границ циклов определяют множество итераций в выпуклом многограннике. Это представление не содержит каких-либо предположений о порядке выполнения итераций в итерационном пространстве.
- ◇ Исходная программа устанавливает один последовательный порядок итераций, который является лексикографическим порядком относительно индексных переменных цикла от самой внешней до самой внутренней.  
Однако итерации в этом пространстве могут выполняться в любом порядке, при котором соблюдаются их зависимости по данным (т.е., порядок, в котором производятся чтения и записи некоторого элемента массива, выполняемые разными операторами присваивания в гнезде циклов не меняется).
- ◇ Пусть допустимое и требуемый порядок итераций уже задан. Как сгенерировать код, который реализует этот порядок.

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения



**Пример.** Между итерациями в программе из рассматриваемого примера нет зависимостей. Поэтому можно выполнять эти итерации в произвольном порядке, последовательно или параллельно.

Так как в программе итерация  $[i, j]$  осуществляет обращение к элементу  $Z[j, i]$ , исходная программа посещает массив в следующем порядке:

|          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|
| $Z[0,0]$ | $Z[1,0]$ | $Z[2,0]$ | $Z[3,0]$ | $Z[4,0]$ | $Z[5,0]$ | $Z[6,0]$ | $Z[7,0]$ |
|          | $Z[1,1]$ | $Z[2,1]$ | $Z[3,1]$ | $Z[4,1]$ | $Z[5,1]$ | $Z[6,1]$ | $Z[7,1]$ |
|          |          | $Z[2,2]$ | $Z[3,2]$ | $Z[4,2]$ | $Z[5,2]$ | $Z[6,2]$ | $Z[7,2]$ |
|          |          |          | $Z[3,3]$ | $Z[4,3]$ | $Z[5,3]$ | $Z[6,3]$ | $Z[7,3]$ |
|          |          |          |          | $Z[4,4]$ | $Z[5,4]$ | $Z[6,4]$ | $Z[7,4]$ |
|          |          |          |          |          | $Z[5,5]$ | $Z[6,5]$ | $Z[7,5]$ |

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Для улучшения пространственной локальности, предпочтительнее посещать соседние слова массива последовательно, т.е. в следующем порядке:

|          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|
| $z[0,0]$ |          |          |          |          |          |
| $z[1,0]$ | $z[1,1]$ |          |          |          |          |
| $z[2,0]$ | $z[2,1]$ | $z[2,2]$ |          |          |          |
| $z[3,0]$ | $z[3,1]$ | $z[3,2]$ | $z[3,3]$ |          |          |
| $z[4,0]$ | $z[4,1]$ | $z[4,2]$ | $z[4,3]$ | $z[4,4]$ |          |
| $z[5,0]$ | $z[5,1]$ | $z[5,2]$ | $z[5,3]$ | $z[5,4]$ | $z[5,5]$ |
| $z[6,0]$ | $z[6,1]$ | $z[6,2]$ | $z[6,3]$ | $z[6,4]$ | $z[6,5]$ |
| $z[7,0]$ | $z[7,1]$ | $z[7,2]$ | $z[7,3]$ | $z[7,4]$ | $z[7,5]$ |

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Этот способ доступа достигается, когда итерации выполняются в следующем порядке

|       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|
| [0,0] |       |       |       |       |       |
| [0,1] | [1,1] |       |       |       |       |
| [0,2] | [1,2] | [2,2] |       |       |       |
| [0,3] | [1,3] | [2,3] | [3,3] |       |       |
| [0,4] | [1,4] | [2,4] | [3,4] | [4,4] |       |
| [0,5] | [1,5] | [2,5] | [3,5] | [4,5] | [5,5] |
| [0,6] | [1,6] | [2,6] | [3,6] | [4,6] | [5,6] |
| [0,7] | [1,7] | [2,7] | [3,7] | [4,7] | [5,7] |

- ◇ То есть, вместо просмотра итерационного пространства в горизонтальном направлении, реализуется просмотр итерационного пространства в вертикальном направлении, так что  $j$  становится индексом внешнего цикла:

```
for (j = 0; j <= 7; j++)
 for (i = 0; i <= min(5,j); i++)
 z[j.i] = 0;
```

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Пусть дан выпуклый многогранник и порядок индексных переменных, как породить границы циклов, которые просматривают все пространство в лексикографическом порядке переменных? В примере, рассмотренном выше, ограничение  $i \leq j$  определяет нижнюю границу индекса  $j$  во внутреннем цикле исходной программы, но верхнюю границу индекса  $i$ , снова во внутреннем цикле преобразованной программы.  
Границы самого внешнего цикла, выраженные в виде линейных комбинаций символьных констант и констант, определяют диапазон всех возможных значений, которые он может принять. Границы для переменных внутреннего цикла выражаются как линейные комбинации индексных переменных внешнего цикла, символьных констант и констант. Они определяют диапазон значений, которые может принять переменная для каждой комбинации значений переменных внешнего цикла.

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения



#### Проекция

Геометрически, границы изменения индекса внешнего цикла гнезда циклов глубины два можно определить, выполнив проекцию выпуклого многогранника, представляющего итерационное пространство, на ось соответствующую внешнему циклу в этом пространстве. Интуитивным представлением проекции многогранника на пространство меньшей размерности является тень проектируемого объекта на это пространство. Проекцией двумерного итерационного пространства с рис. 17.11 на ось  $i$  является the вертикаль от 0 до 5; а проекцией на ось  $j$  – горизонталь от 0 до 7. Когда 3-мерный объект проектируется в направлении оси  $z$  на 2-мерную плоскость  $(x, y)$ , производится исключение переменной  $z$  с потерей информации о высоте отдельной точки и попросту записывается 2-мерный след объекта на плоскости  $(x, y)$ .

Генерация границ цикла лишь одно из многочисленных применений проекции. Проекцию можно формально определить следующим образом. Пусть  $S$  –  $n$ -мерный многогранник. Проекцией  $S$  на первые  $m$  его измерений является множество точек  $(x_1, x_2, \dots, x_m)$ , таких что для некоторых  $x_{m+1}, x_{m+2}, \dots, x_n$ , вектор  $[x_1, x_2, \dots, x_n]$  принадлежит  $S$ . Проекцию можно вычислить с помощью <sup>39</sup>исключения Фурье-Мюккина, следующим образом:

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения



#### Проекция

Генерация границ цикла лишь одно из многочисленных применений проекции. Проекцию можно формально определить следующим образом. Пусть  $S$  –  $n$ -мерный многогранник. Проекцией  $S$  на первые  $m$  его измерений является множество точек  $(x_1, x_2, \dots, x_m)$ , таких что для некоторых  $x_{m+1}, x_{m+2}, \dots, x_n$ , вектор  $[x_1, x_2, \dots, x_n]$  принадлежит  $S$ . Проекцию можно вычислить с помощью метода *исключения Фурье-Моцкина*, следующим образом:

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения



Алгоритм 17.11: Исключение Фурье-Моцкина.

ВХОД: Многогранник  $S$  в пространстве переменных  $x_1, x_2, \dots, x_n$ . То есть,  $S$  – *множество линейных ограничений*, содержащих переменные  $x_i$ . Одна из заданных переменных  $x_m$  считается переменной, которую следует исключить.

ВЫХОД: Многогранник  $S'$  в пространстве переменных  $x, \dots, x_{m-1}, x_{m+1}, \dots, x_n$  (т.е., всех переменных  $S$ , за исключением  $x_m$ ), который является проекцией  $S$  на измерения, отличные от  $m$ -ого.

МЕТОД: Пусть  $C$  – все ограничения из  $S$ , содержащие  $x_m$ . Выполнить следующее:

Для каждой пары (нижняя граница, верхняя граница  $x_m$ ) из  $C$ , такой что

$$L \leq c_1 x_m$$

$$c_2 x_m \leq U$$

создать новое ограничение

$$c_2 L \leq c_1 U$$



## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения



Отметим, что  $s_1$  и  $s_2$  – целые, но  $L$  и  $U$  могут быть выражениями относительно переменных, отличных от  $x_m$ .

Если целые  $s_1$  и  $s_2$  имеют общий делитель, поделить обе части на этот делитель.

Если новое ограничение не может быть удовлетворено, значит решение  $S$  не существует; т.е., оба многогранника  $S$  и  $S'$  пусты.

$S'$  определяется как множество ограничений  $S - C$ , плюс все ограничения, созданные на шаге 1.

Попутно отметим, что если у  $x_m$  имеется  $u$  нижних границ и  $v$  верхних границ, исключение  $x_m$  порождает до  $uv$  неравенств, но не более того.

Ограничения, добавленные в шаг (1) алгоритма 17.11, соответствуют применению ограничений  $C$  к остальным переменным системы.

Следовательно, решение в  $S'$  имеется тогда, и только тогда, когда существует по крайней мере одно соответствующее решение в  $S$ . Для заданного решения в  $S'$  диапазон, соответствующий  $x_m$ , может быть найден заменой всех переменных в ограничениях  $C$ , кроме  $x_m$ , на их фактические значения.

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Пример 17.12: Рассмотрим неравенства, определяющие итерационное пространство на рис. 17.11. Пусть требуется применить исключение Фурье-Моцкина, чтобы спроектировать двумерное пространство на ось  $j$ . Имеется одна нижняя граница для  $i$ :  $0 \leq i$  и две верхние границы:  $i \leq j$  и  $i \leq 5$ . Это порождает два ограничения:  $0 \leq j$  и  $0 \leq 5$ . Последнее тривиально верно, и его можно опустить. Первое определяет нижнюю границу  $j$ , а исходная верхняя граница  $j \leq 7$  определяет его верхнюю границу. □

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

#### ◇ Генерация границ цикла

После того, как было определено исключение Фурье-Мюллена, алгоритм генерации границ циклов с помощью итераций над выпуклым многогранником (Алгоритм 17.13) стал очевидным. Вычисляются границы циклов в порядке, от самого внутреннего цикла к внешним. Все неравенства, содержащие переменную самого внутреннего цикла записываются в виде нижней или верхней границы переменной. Потом все проектируется в пространство, не включающее размерности, соответствующей самому внутреннему циклу, в результате получается многогранник, имеющий на одно измерение меньше. Итерации повторяются до тех пор, пока границы всех переменных циклов не будут найдены.

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ **Алгоритм 17.13:** Вычисление границ для заданного порядка переменных.  
ВХОД:  $A$  выпуклый многогранник  $S$  над переменными  $v_1, \dots, v_n$ .  
ВЫХОД: Множество нижних границ  $L_i$  и верхних границ  $U_i$  для каждого  $v_i$ , выраженных только  $v_j'$ , для  $j < i$ .  
МЕТОД: Алгоритм описан ниже на псевдокоде. □
- $S_n = S$ ;  
/\* Используется алгоритм 17.11 для нахождения границ \*/  
for (  $i = n$ ;  $i > 1$ ;  $i--$  ) {  
     $L_i$  = все нижние границы  $V_i$  в  $S_i$   
     $U_i$  = все верхние границы  $V_i$  в  $S_i$ ,  
     $S_{i-1}$  = Ограничения, полученные после применения  
        Алгоритма 17.11, для исключения  $v_i$  из  
        ограничений на ограничения  $S_i$ ,  
} /\* Удаление избыточностей \*/  
 $S' = \emptyset$ ;  
for (  $i = 1$ ;  $i \leq n$ ;  $i++$  ) {  
Из  $L_i$  и  $U_i$  удаляются любые ограничения из  $S'$ ;  
Все оставшиеся в  $L_i$  и  $U_i$  ограничения на  $v_i$   
добавляются в  $S'$ ;  
}

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

◇      [0, 2], [1, 3], [2, 4], [3, 5], [4, 6], [5, 7]  
      [0, 3], [1, 4], [2, 5], [3, 6], [4, 7]  
      [0, 4], [1, 5], [2, 6], [3, 7]  
      [0, 5], [1, 6], [2, 7]  
      [0, 6], [1, 7]  
      [0, 7]

Рис. 17.16: Диагональное упорядочение итерационного пространства с рис. 17.11

**Пример 17.14:** Применим алгоритм 17.13 для генерации границ циклов с обходом итерационного пространства с рис. 17.11 по вертикали. Порядок переменных  $j, i$ . Алгоритм генерирует следующие границы:

$L_i:$      0  
 $U_i:$      5,  $j$   
 $L_j:$      0  
 $U_j:$      7

Необходимо удовлетворить всем ограничениям, поэтому границей  $i$  является  $\min(5, j)$ . В этом примере нет избыточностей. □

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

#### ◇ 17.3.6 Смена осей

Отметим, что просмотры итерационного пространства по горизонталям и вертикалям, рассмотренные выше, всего лишь два наиболее обычных способа обращения к итерационному пространству. Существует много других возможностей; например, можно просматривать итерационное пространство примера 17.6 по диагоналям, как это рассматривается в примере 17.15.

◇ **Пример 17.15:** Можно просматривать итерационное пространство, представленное на рис. 17.11 по диагоналям, используя порядок, показанный на рис. 17.16. Разность между координатами  $j$  и  $i$  на каждой диагонали является константой, начиная с 0 и до 7. Таким образом, нужно определить новую переменную  $k = j - i$  и просматривать итерационное пространство в лексикографическом порядке относительно  $k$  и  $j$ . Подставив  $i = j - k$  в неравенства, получаем:

$$\begin{array}{rclcl} 0 & \leq & j - k & \leq & 5 \\ j - k & \leq & j & \leq & 7 \end{array}$$

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Чтобы определить границы циклов для порядка, описанного выше, можно применить алгоритм 17.13 к вышеприведенной системе неравенств с переменными упорядочения  $k$ ,  $j$ .

$Lj: k$

$Uj: 5 + k, 7$

$Lk: 0$

$Uk: 7$

- ◇ Эти неравенства позволяют составить следующий код, заменяя в обращениях к массивам  $i$  на  $j - k$ .

```
for (k = 0; k <= 7; k++)
 for (j = k; j <= min(5+k, 7); j++)
 z[j, j - k] = 0;
```

## 17.4 Пространство итераций

### 17.4.2. Управление порядком выполнения

- ◇ Вообще можно менять оси многогранника, создавая новые индексные переменные цикла, которые представляют собой аффинные комбинации исходных переменных, и определяя упорядочение этих переменных. Трудной задачей является выбор правильных осей, чтобы удовлетворить зависимостям по данным, добиваясь параллельности и локальности. Эта задача обсуждается, начиная с раздела 17.7. Здесь же мы установили, что когда оси выбраны, генерация требуемого кода не вызывает затруднений, как это показано в примере 17.15.
- ◇ Существует много других порядков обхода итераций, к которым неприменимы эти методы. Например, может потребоваться посетить нечетные строки пространства итераций до того, как будут посещены четные строки. Или может потребоваться начать итерации из середины пространства итераций и продвигаться к краям. Однако для приложений с аффинными функциями доступа, описанные методы покрывают большую применяемых порядков итераций.