1. Введение. Неоптимизирующий компилятор

Запись на курс

Для записи на курс необходимо зарегистрироваться на сайте

compilers.ispras.ru

Содержание занятий

1. Занятия проводятся по понедельникам – первые 2 пары.
2. На первой паре – лекция,
   на второй паре – окончание лекции и короткая (25 минут) контрольная работа (ККР), на которой решение простых задач по теме предыдущей лекции.
3. На сайте – недельные домашние задания (НДЗ – всего за семестр – 4 задания). Срок выполнения задания – 1 неделя.

Оценка за курс

1. За каждую ККР выставляется оценка в пятибалльной системе. Если студент не писал, или не сдал ККР, он получает за эту ККР оценку 0 баллов.
2. Зарегистрировавшиеся на сайте получат в течение семестра 4 недельных домашних задания (НДЗ)
3. Если средняя оценка за ККР окажется не ниже 3 баллов, а средняя оценка за НДЗ – не ниже 4 баллов, студент может быть освобожден от экзамена, и ему в качестве экзаменационной оценки будет выставлена средняя оценка за НДЗ.

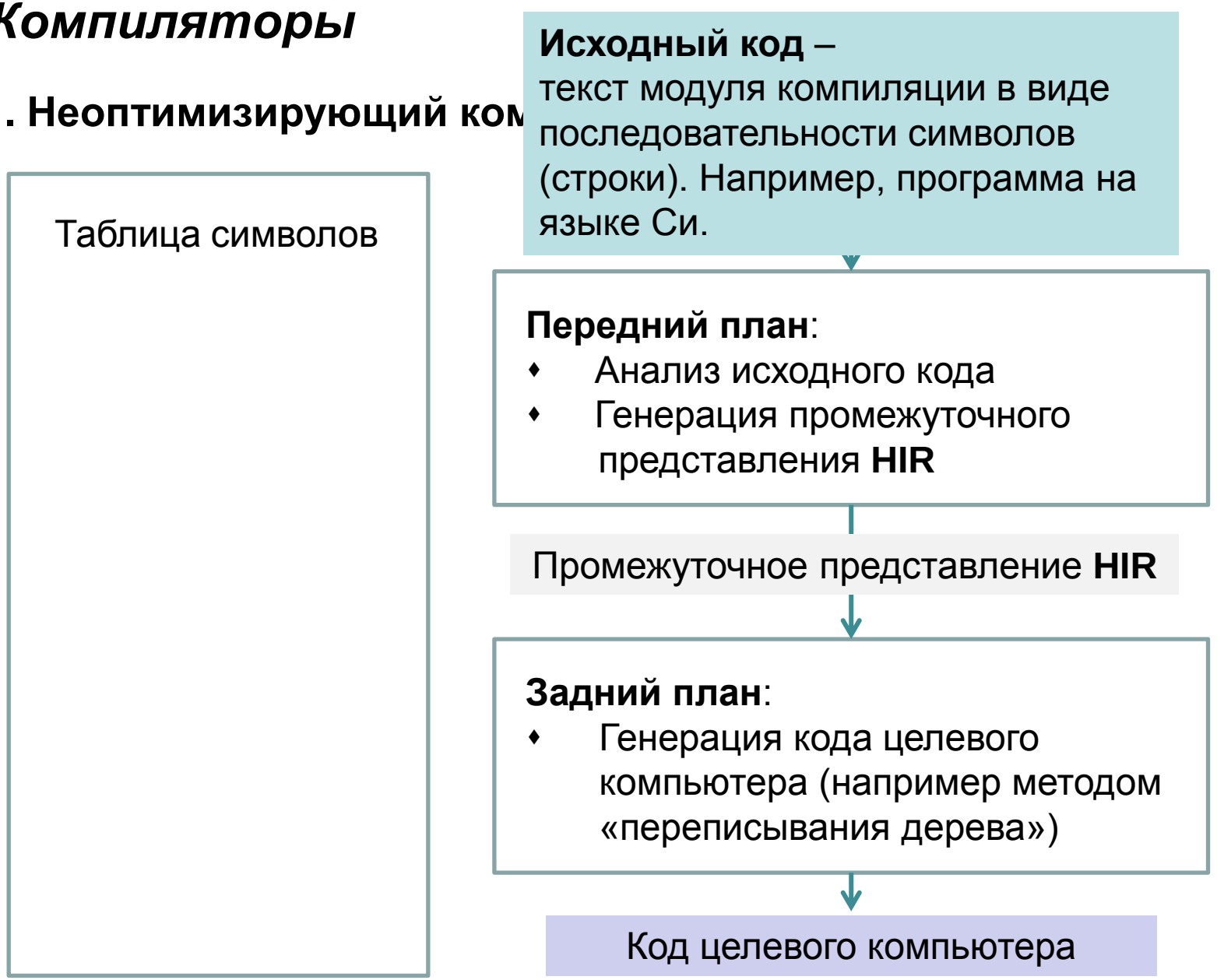
1.1 Компиляторы

1.1.1. Неоптимизирующий компилятор

Структура неоптимизирующего компилятора

1.1 Компиляторы

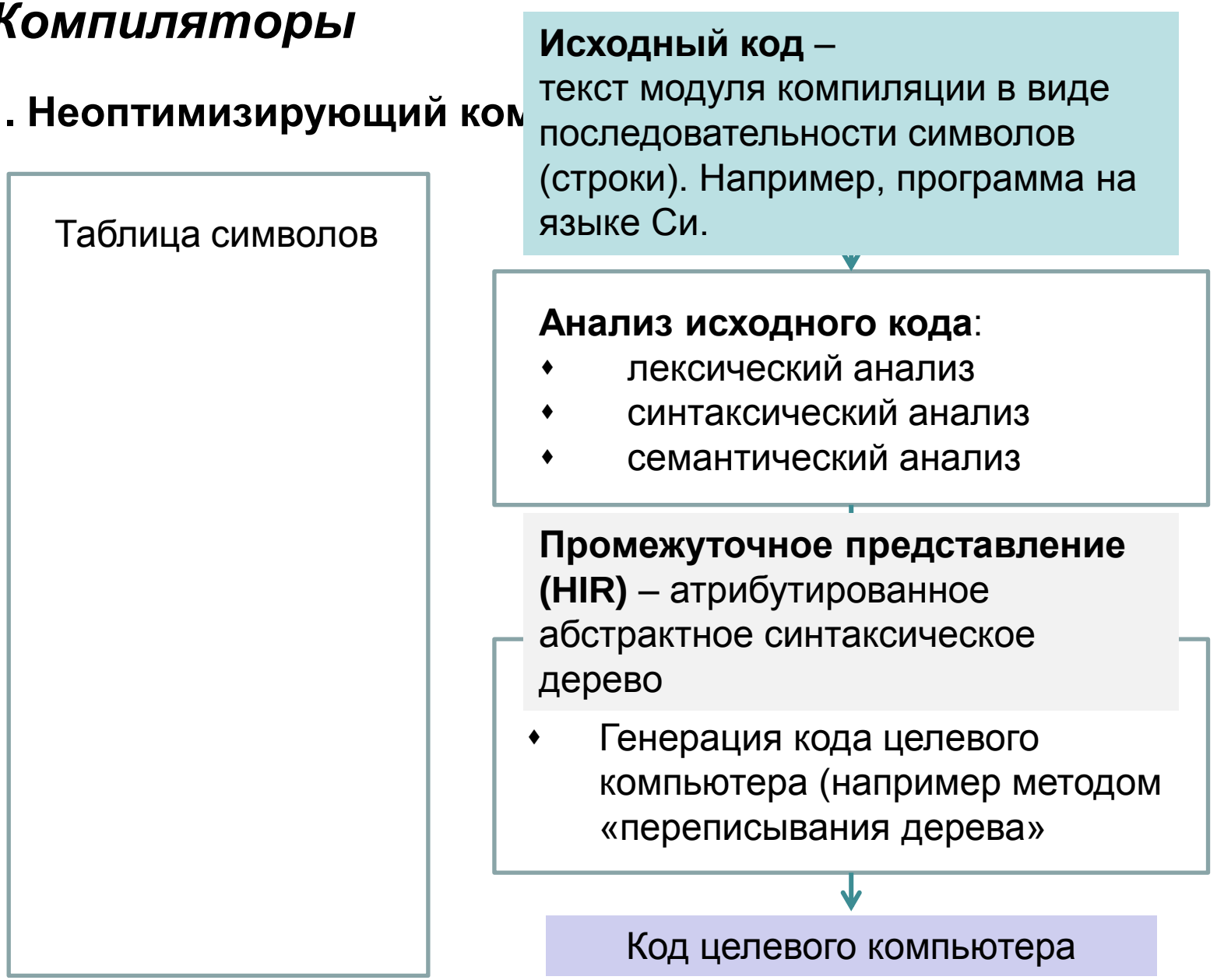
1.1.1. Неоптимизирующий ком



Структура неоптимизирующего компилятора

1.1 Компиляторы

1.1.1. Неоптимизирующий ком



Структура неоптимизирующего компилятора

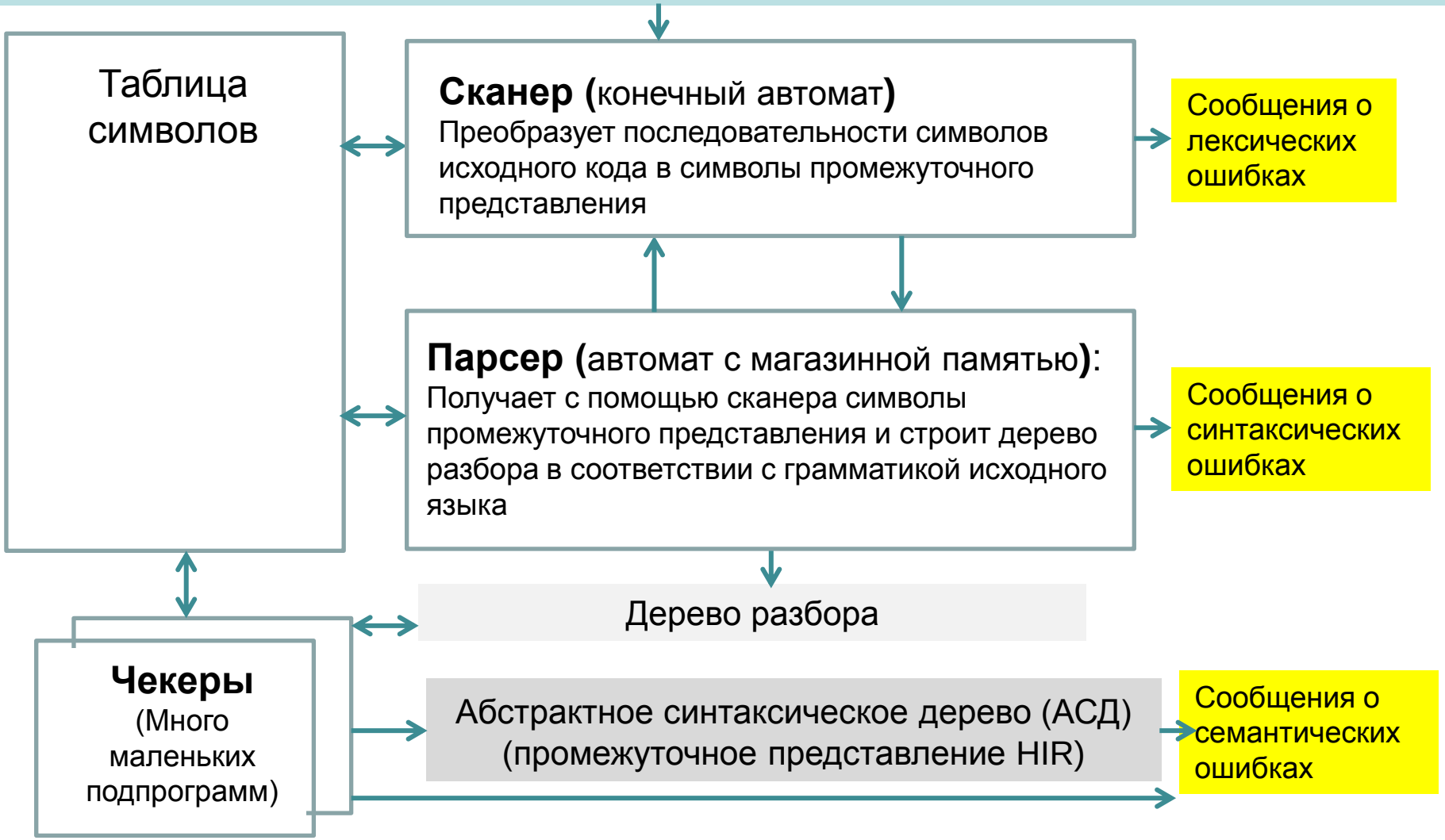
1.1 Компиляторы

1.1.2. Синтаксически управляемая трансляция

1.1 Компиляторы

1.1.2. Синтаксически управляемая трансляция

Исходный код – текст модуля компиляции в виде строки символов

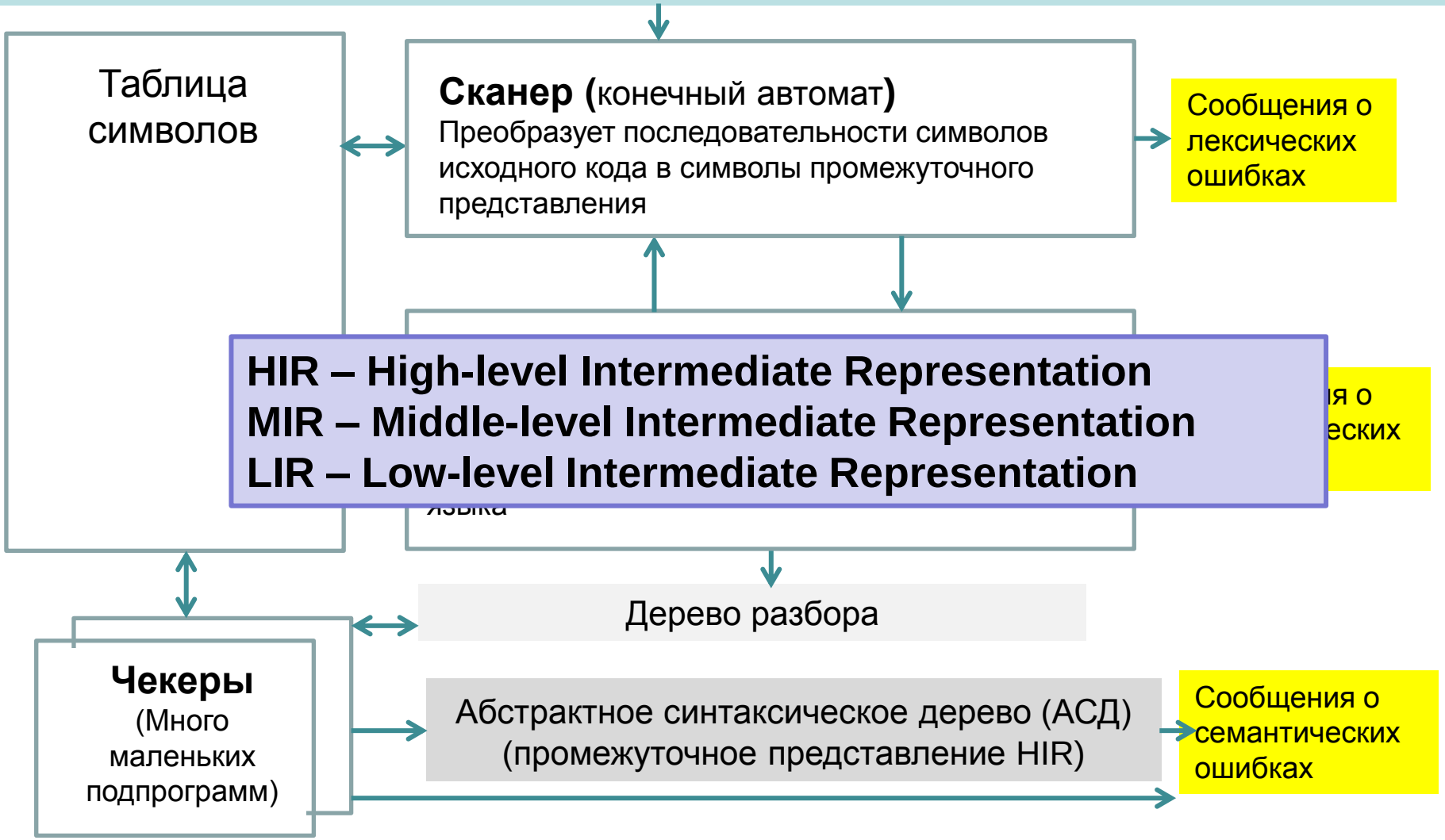


Каждый чекер вычисляет атрибут узлов дерева разбора и проверяет соответствуют ли атрибуты родительских и дочерних узлов; после чего либо строит соответствующую часть АСД, либо выдает сообщение об ошибке

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

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.

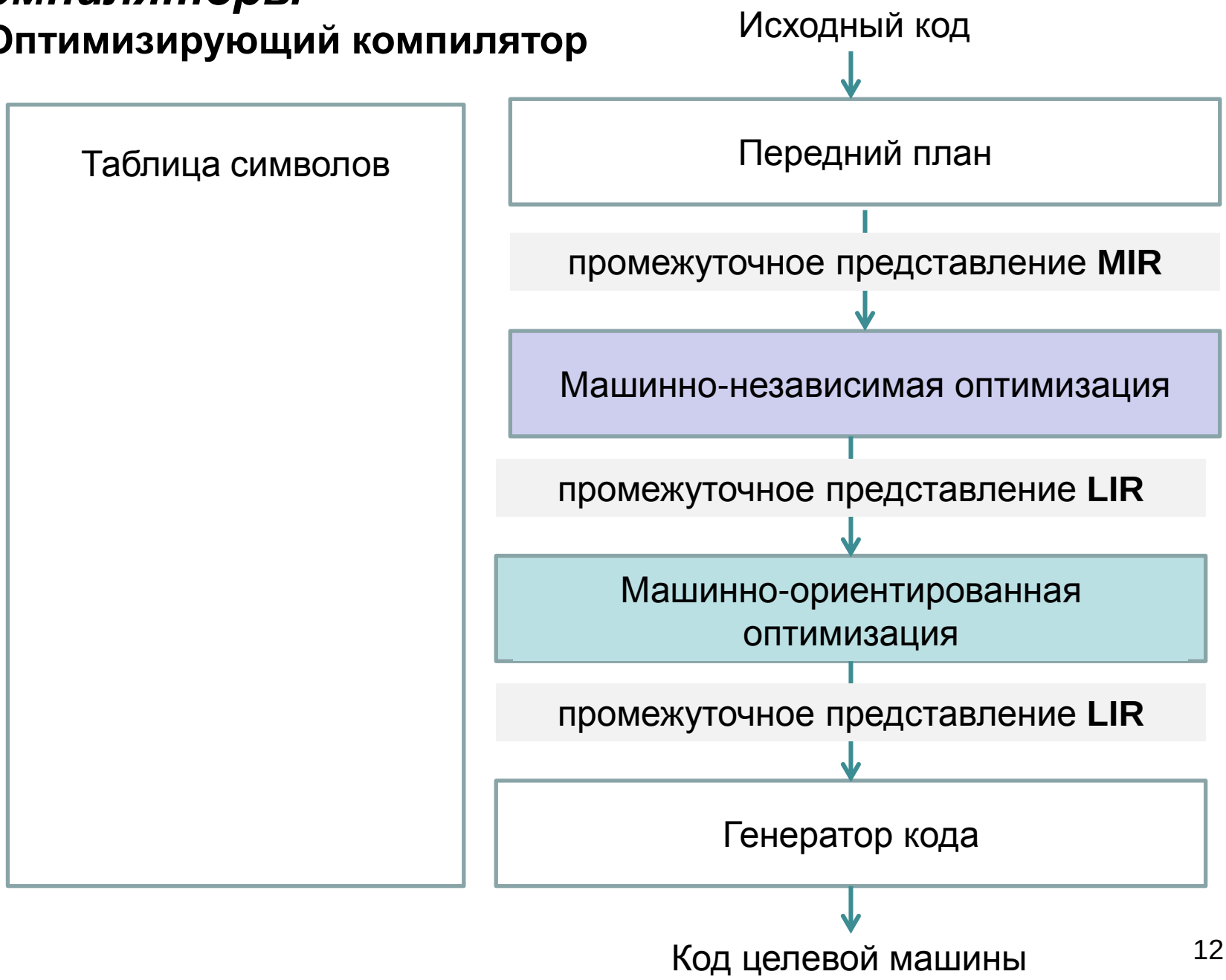
◊ Передний и задний планы компилятора изучаются в курсе «Конструирование компиляторов», а также в таких курса как «Теория алгоритмических языков», «Теория автоматов» и др.
В данном курсе считается, что студенты все это знают. Для тех, кто забыл могу рекомендовать учебное пособие

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

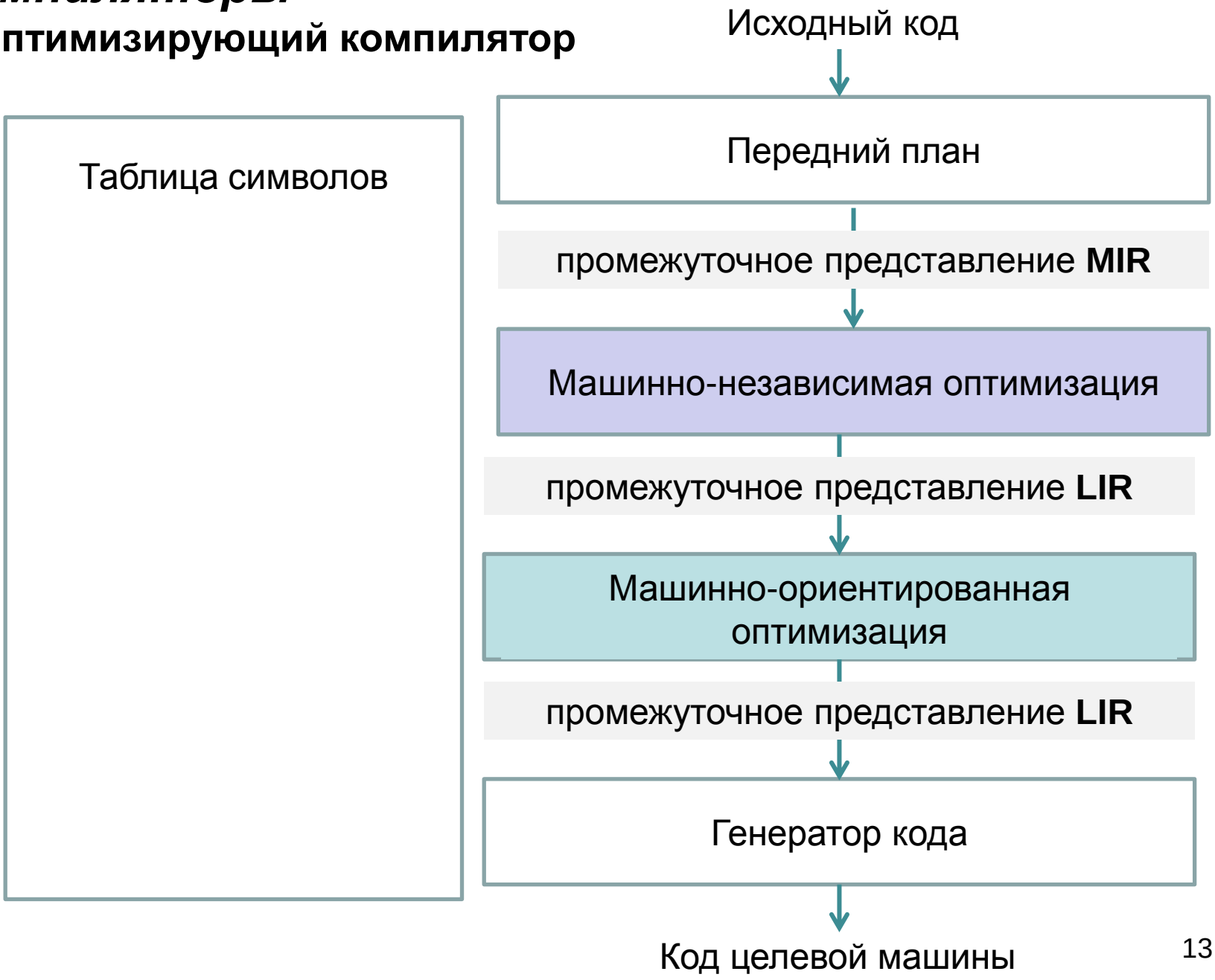
1.1 Компиляторы

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).

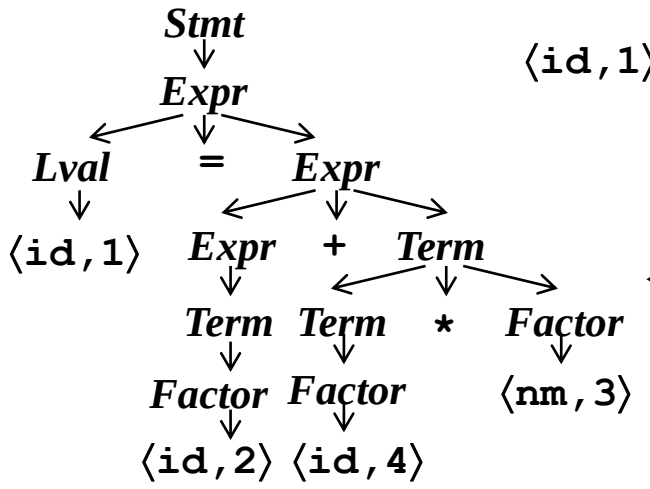
1.3 Генерация промежуточного представления MIR

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

Анализатор лексики

⟨id,1⟩⟨=⟩⟨id,2⟩⟨+⟩⟨nm,3⟩⟨*⟩⟨id,4⟩ Последовательность токенов

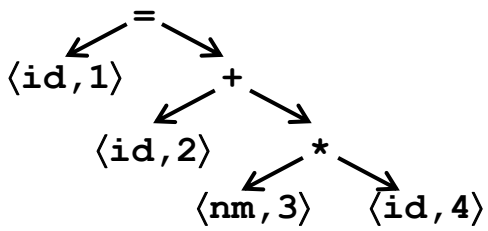
Анализатор синтаксиса



Дерево разбора

Дерево разбора

Анализатор семантики



Абстрактное синтаксическое дерево (**HIR**)

Генератор промежуточного представления

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

MIR

Таблица символов

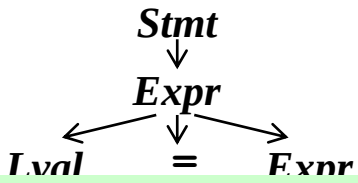
Внутреннее имя	Внешнее имя	Атрибуты
id1	current_pos	int
id2	initial_pos	int
nm3	16	int
id4	step	int

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, 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**

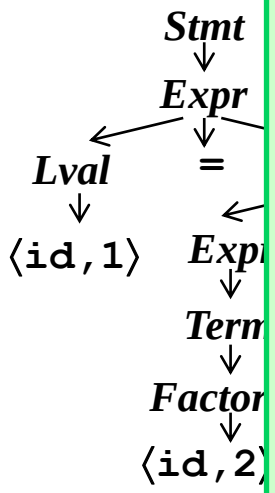
представления

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

MIR

1.3 Генерация промежуточного представления MIR

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



ность токенов

нзбора

нтаксическое
во
)

Промежуточное представление **MIR** включает

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

Пример «четверки»: `*, t1, nm3, id4`

или `t1 ← *, nm3, id4`

Таблица символо

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

Генератор промежуточного представления

MIR

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

1.4. Промежуточное представление MIR

1.4.1. Определение промежуточного представления MIR

◊ Инструкции присваивания	(**x**, **y** и **z** – имена (адреса) переменных или константы):
x ← op, y, z	выполнить бинарную операцию **op** над операндами **y** и **z** и поместить результат в **x**
x ← op, y	выполнить унарную операцию **op** над операндом **y** и поместить результат в **x**
x ← y	скопировать значение переменной **y** в переменную **x**
x[i] ← y	поместить значение **y** в **i**-ю по отношению к **x** ячейку памяти
x ← 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:	
`goto L`	Безусловный переход: следующей будет выполнена инструкция с меткой L
`ifTrue x goto L`	Условный переход: если x истинно, следующей будет выполнена инструкция с меткой L
`ifFalse x goto L`	Условный переход: если x ложно, следующей будет выполнена инструкция с меткой L

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`**

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

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 средину 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)	i ← -, m, 1	(16)	t7 ← *, 4, i
(2)	j ← n	(17)	t8 ← *, 4, j
(3)	tl ← *, 4, n	(18)	t9 ← a[t8]
(4)	v ← a[tl]	(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] ← tl4
(14)	t6 ← *, 4, j	(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)	`i ← -, m, 1`	(16)	`t7 ← *, 4, i`
(2)	`j ← n`	(17)	`t8 ← *, 4, j`
(3)	`tl ← *, 4, n`	(18)	`t9 ← a[t8]`
(4)	`v ← a[tl]`	(19)	`a[t7] ← t9`
(5)			
(6)			
(7)			
(8)			
(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] ← tl4`
(14)	`t6 ← *, 4, j`	(29)	`t15 ← *, 4, n`
(15)	`x ← a[t6]`	(30)	`a[t15] ← x`

Напоминаю: НББ по определению это:

- первая инструкция программы
- помеченная инструкция программы
- инструкция, следующая за инструкцией перехода

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)	tl ← *, 4, n		(18)	t9 ← a[t8]	
(4)	v ← a[tl]		(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] ← tl4	
(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)	i ← -, m, 1	НББ	(16)	t7 ← *, 4, i	
(2)	j ← n		(17)	t8 ← *, 4, j	
(3)	tl ← *, 4, n		(18)	t9 ← a[t8]	
(4)	v ← a[tl]		(19)	a[t7] ← t9	
(5)	L1: i ← +, i, 1	НББ	(20)	t10 ← *, 4, j	

Началами базовых блоков являются инструкции с номерами:
(1), (5), (9). (13), (14), (23). Алгоритм 1.5.3 позволяет построить следующие ББ

(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] ← tl4	
(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

Блок A

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

Блок B

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

Блок C

```
(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 E

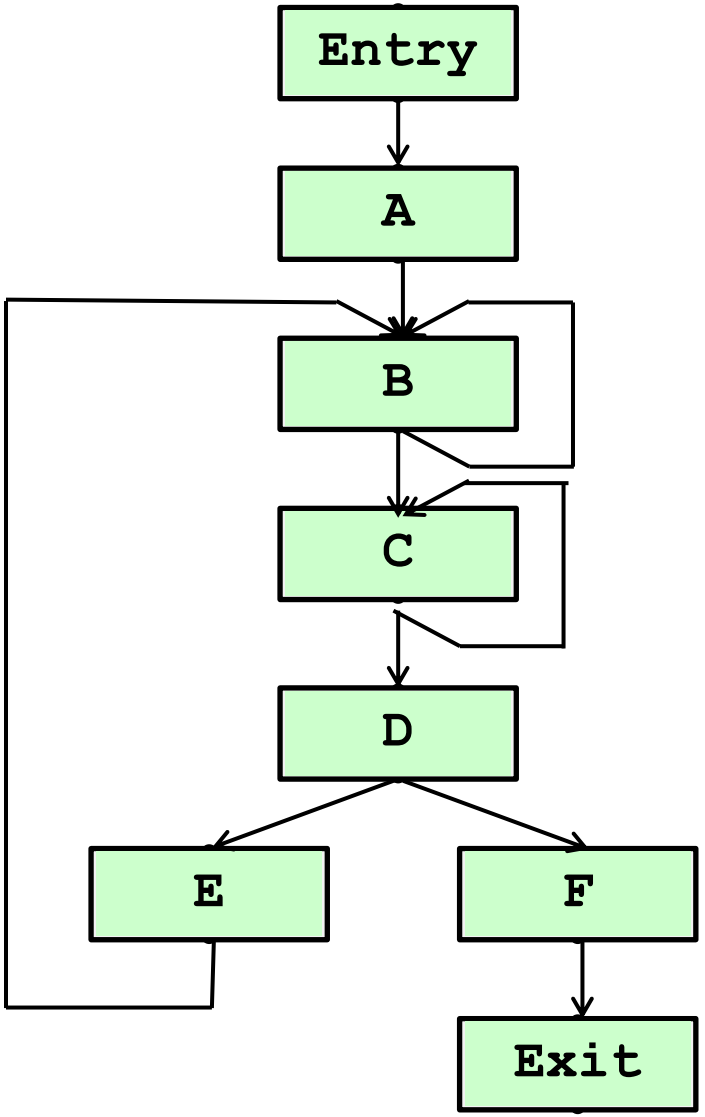
```
(14)   t6 ← *, 4, i
(15)   x ← a[t6]
(16)   t7 ← *, 4, i
(17)   t8 ← *, 4, j
(18)   t9 ← a[t8]
(19)   a[t7] ← t9
(20)   t10← *, 4, j
(21)   a[t10] ← x
(22)   goto L1
```

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 – 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).

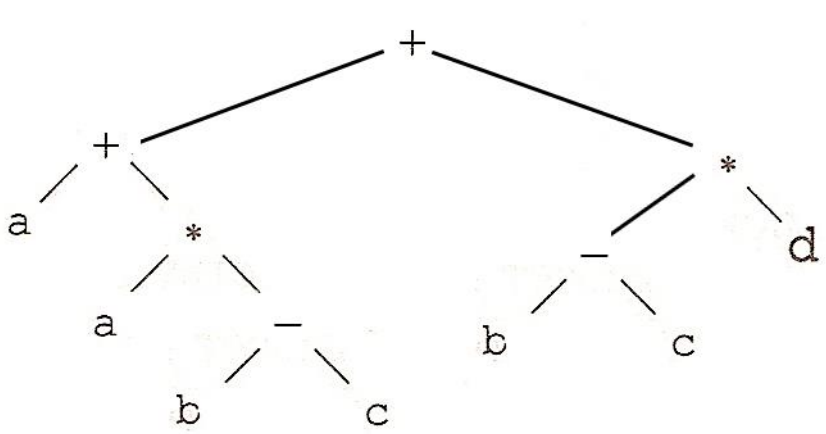
1.6 Локальная оптимизация (*оптимизация базовых блоков*)

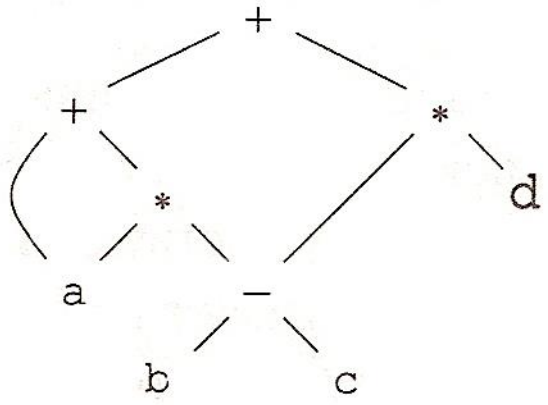
1.6.2 Представление базового блока в виде ориентированного ациклического графа

◊ **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

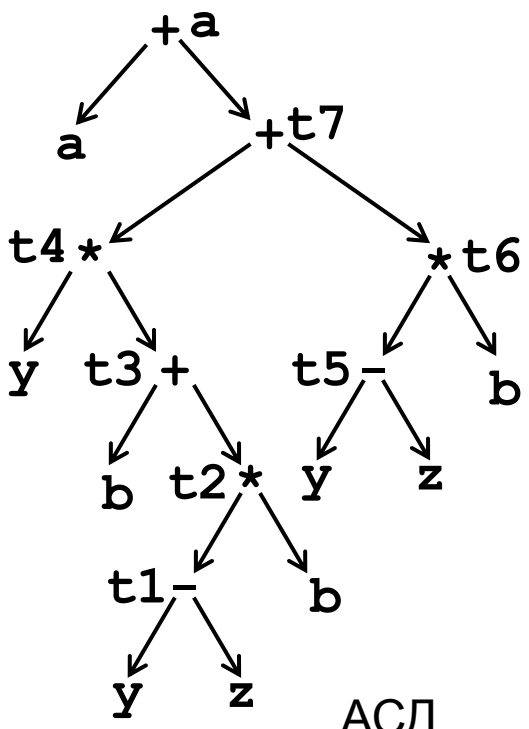
1.6.2 Представление базового блока в виде ориентированного ациклического графа

◊ **Пример**. Выражение в исходном коде:

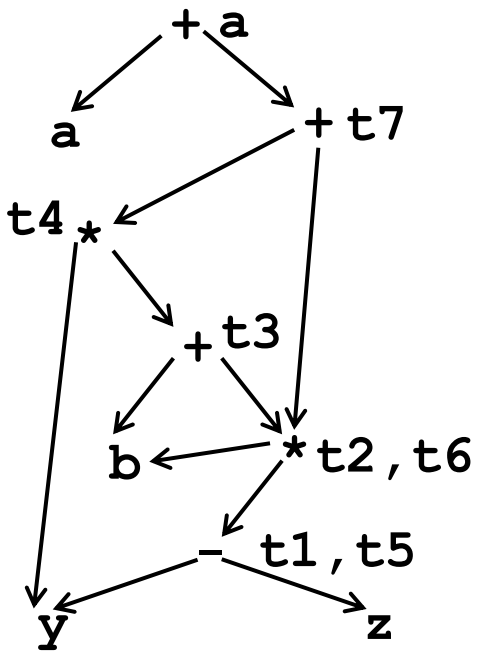
```
a = a + y*(b +(y-z)*b)+(y-z)* b
```

```
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

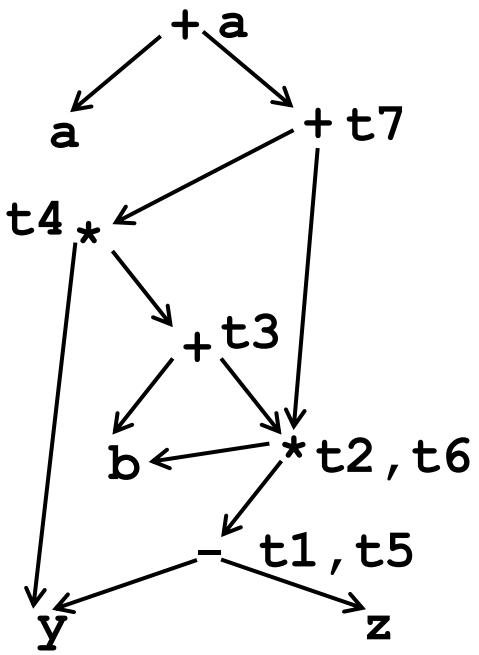
1.6.2 Представление базового блока в виде ориентированного ациклического графа

◊ **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

1.6.3 Метод нумерации значений – алгоритм построения ОАГ

Представим *ОАГ* в виде таблицы значений

- ◊ Каждая строка таблицы значений представляет один узел ОАГ.
- ◊ Первое поле каждой записи представляет собой код операции
- ◊ Каждой правой части операции

  ⟨**op, left, right**⟩,

  где **op** – код операции, а **left** и **right** – левый и правый операнды, соответствует ее *сигнатура*

  ⟨**op, #left, #right**⟩,

  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

1.6.4. Алгоритм построения ОАГ для базового блока B

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

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

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

```
for each "t_i ← 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

1.6.3 Представление *ОАГ* в виде таблицы значений

◊ Таблица значений рассматриваемого примера имеет вид:

$t1^5 \leftarrow -, y^3, z^4$

$t2^6 \leftarrow *, t1^5, b^2$

$t3^7 \leftarrow +, b^2, t2^6$

$t4^8 \leftarrow *, y^3, t3^7$

$t5^5 \leftarrow -, y^3, z^4$

$t6^6 \leftarrow *, t5^5, b^2$

$t7^9 \leftarrow +, t4^8, t6^6$

$a^{10} \leftarrow +, a^1, t7^9$

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. Пример

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

(a) Блок E
до оптимизации

```
t1^5 ← -, y^3, z^4
t2^6 ← *, t1^5, b^2
t3^7 ← +, b^2, t2^6
t4^8 ← *, y^3, t3^7
t5^5 ← -, y^3, z^4
t6^6 ← *, t5^5, b^2
t7^9 ← +, t4^8,t6^6
a^10 ← +, a^1, t7^9
```

(c) Блок E
после нумерации значений

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

(c) Блок E
после оптимизации

1.6 Локальная оптимизация

1.6.6 Удаление общих подвыражений

◊ Общие подвыражения обнаруживаются при построении узлов *ОАГ* с помощью алгоритма 1.6.4 (нумерация значений).

1.6.7 Сворачивание констант

◊ *Сворачивание констант* заключается в вычислении констант в процессе компиляции и замене константных выражений их значениями. Например, выражение 2 * 3.14 можно заменить значением 6.28.

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)

Дорогие	Дешевые
x^2	x × x
2 × x	x + x
x/2	x × 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 (**Op**), 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,

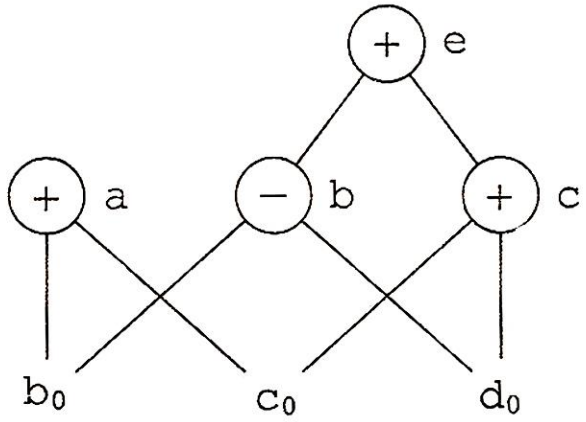
$Input$ – множество переменных, определенных до входа в блок B,

$Output$ – 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
```




```
a ← +, b, c
b ← -, b, d
```

Базовый блок B до удаления мертвого кода

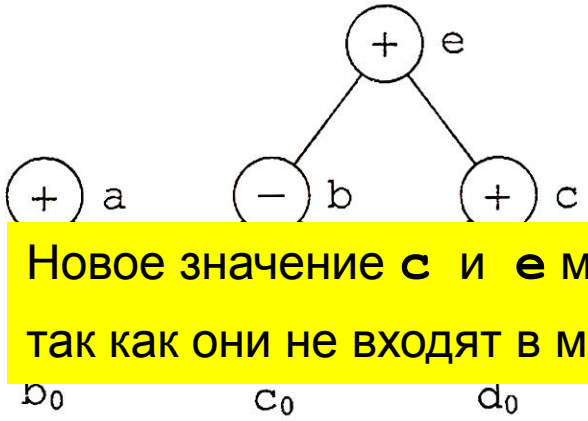
ОАГ базового блока B

Базовый блок B после удаления мертвого кода

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
```




```
a ← +, b, c
b ← -, b, d
```

Новое значение **c** и **e** можно не вычислять, так как они не входят в множество *Output*

Базовый блок B до удаления мертвого кода

ОАГ базового блока B

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: $\mathbf{a} \leftarrow +, b_0, c_0$

$\mathbf{d} \leftarrow -, a, d_0$

$\mathbf{c} \leftarrow +, d, c_0$

$\mathbf{b} \leftarrow d$

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

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

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.