1. Инструкция внутреннего представления программы -

`current_pos=initial_pos+16*step` — Строка исходной программы

Таблица символов (ТС)

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

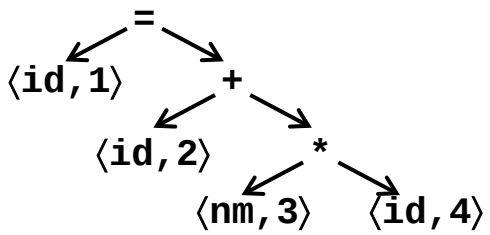
Анализатор лексики

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

Анализатор синтаксиса

Дерево разбора — Дерево разбора

Анализатор семантики



Абстрактное синтаксическое дерево (АСД – Внутреннее представление 1)

Внутреннее представление 1(2):
инструкции вида: **`x ← op, y, z`**
где **`x, y, z`** – абстрактные области памяти, отведенные под соответствующие переменные

Внутреннее представление 2(3):
операции вида: **`OP X, Y, Z,`**
где **`x, Y, Z`** – регистры или адреса ячеек памяти;
операции объединяются в *команды*

Генератор внутреннего представления

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

Внутреннее представление 2

2. Виды инструкций

◊ 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**

◊ 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

◊ Процедуры:	
param x	Передача фактического параметра вызываемой процедуре (если вызываемая процедура имеет n параметров, то инструкции ее вызова предшествует n нструкций *param*
call p, n	Вызов процедуры p, имеющей n параметров
return y	Возврат из процедуры y – необязательное возвращаемое значение

3. Базовый блок (определение)

◊ *Базовым блоком* (или *линейным участком*) называется последовательность следующих друг за другом трехадресных команд, обладающая 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.

4. Как выделить базовый блок

- Строится упорядоченное множество НББ (начало базового блока, либо первая инструкция программы, либо место, куда указывает оператор условного/безусловного перехода, либо первая инструкция после инструкции перехода)
- Каждому НББ соответствует ББ, который определяется как последовательность инструкций, содержащая само НББ и все инструкции до следующего НББ (не включая его) или до конца программы.

5. Множества Input и output для базового блока

◊ Каждый базовый блок задается тройкой объектов: $B = \langle P, Input, Output \rangle$, где P – последовательность инструкций блока B, $Input$ – множество переменных, определения которых достигают блока B, $Output$ –множество переменных, живых после блока B.

6. Локальная оптимизация

Оптимизация в пределах одного базового блока.

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 указанные преобразования можно выполнить за один просмотр ББ, если представить его в виде ориентированного ациклического графа (ОАГ).

- Все задачи локальной оптимизации позволяет решить *метод* локальной *нумерации значений*.
- Избыточные вычисления внутри базового блока автоматически удаляются в процессе построения *ОАГ* (*ориентированного ациклического графа*).
- Множества *Input* и *Output* позволяют фиксировать использование неинициализированных переменных и исключить присваивание значений мертвым переменным.

8. Локальный метод нумерации значений

Представление *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).
- **(7) Номер значения** – это номер строки таблицы значений (ТЗ), в которой определяется это значение
- **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 ← 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. Ориентированный ациклический граф - ОАГ, в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения для базового блока. **a = a + y*(b +(y-z)*b)+(y-z)* b**

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

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

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

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

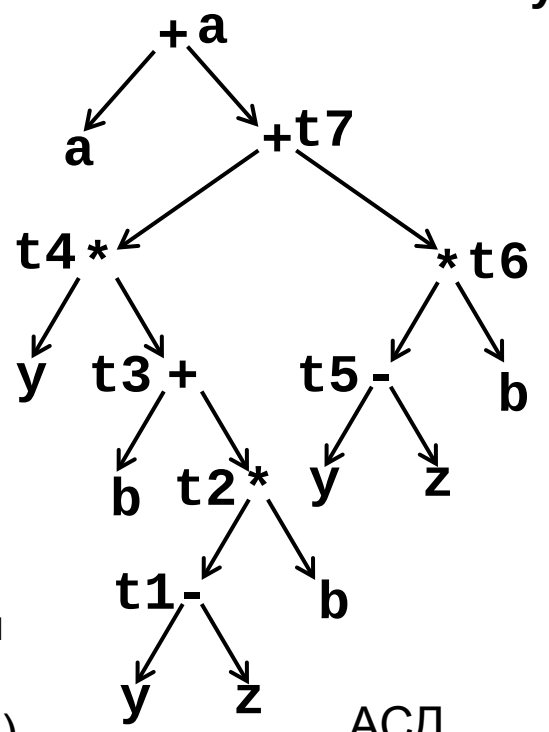
$t5^5 \leftarrow -, y^3, z^4$

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

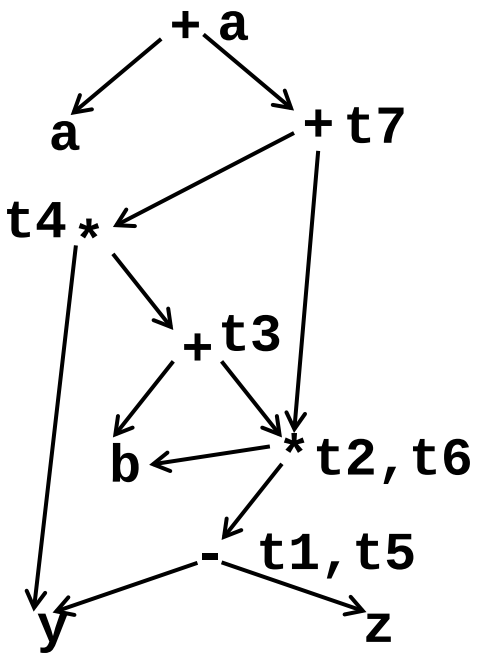
$t7^9 \leftarrow +, t4^8, t6^6$

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

Выражение в промежуточном представлении



АСД



ОАГ

Таблица значений(ТЗ)

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)			

Таблица символов (ТС)

Внутреннее имя	**Внешнее имя**	**Атрибуты**
id1	current_pos	int
nm3	16	int

10. Граф потока управления - множество всех возможных путей исполнения программы, представленное в виде ориентированного графа, вершинами которого являются ББ.

11. Как построить ГПУ

Вход: последовательность трехадресных инструкций.

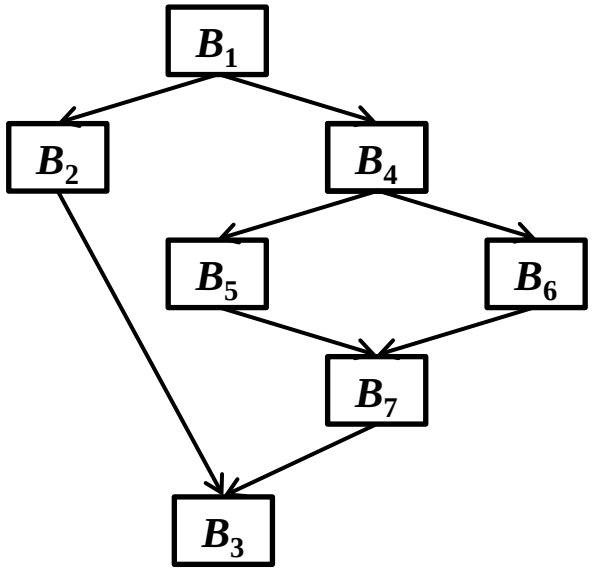
Выход: список базовых блоков для данной последовательности инструкций, такой что каждая инструкция принадлежит только одному базовому блоку.

Метод:

1. Строится упорядоченное множество НББ (начало базового блока, либо первая инструкция программы, либо место, куда указывает оператор условного/безусловного перехода, либо первая инструкция после инструкции перехода)
2. Каждому НББ соответствует ББ, который определяется как последовательность инструкций, содержащая само НББ и все инструкции до следующего НББ (не включая его) или до конца программы.
3. 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.

12. Региональная оптимизация - оптимизация в пределах нескольких базовых блоков (например, суперблока).

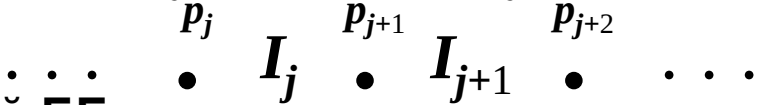
13. Суперблок - расширенный базовый блок E определяется как множество базовых блоков B_1, B_2, …, B_n, где у блока B_1 может быть несколько предшественников, а каждый из блоков B_i, $2 \le i \le n$ имеет в суперблоке единственного предшественника. Блоки $B_i \in E$ формируют дерево с корнем B_1. У суперблока E может быть несколько выходов на блоки (или суперблоки), не входящие в состав E.



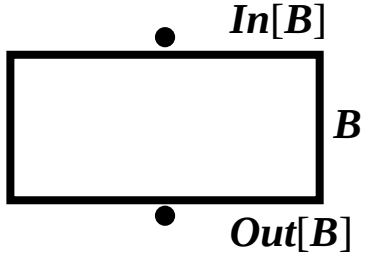
Можно выделить три суперблока: $\{B_1, B_2, B_4, B_5, B_6\}$, $\{B_7\}$ и $\{B_3\}$. Блоки B_7 и B_3 имеют по два предшественника, поэтому их нельзя включить в суперблок больших размеров

14. Глобальная оптимизация - оптимизация в пределах процедуры.

15. Точки программы (*…, p_j , p_{j+1} , p_{j+2} , …*) расположены между ее инструкциями (*…, I_j , I_{j+1} , …*) и характеризуют соответствующие состояния программы.



16. Входной точкой ББ является входная точка его первой инструкции. **Выходной точкой ББ** является выходная точка его последней инструкции.



17. Состояние программы – множество значений всех переменных программы, включая переменные в кадрах стека времени выполнения, находящихся ниже текущей вершины стека.

18-20. Передаточная функция

Пара состояний во входной и выходной точках фрагмента определяет передаточные функции этого фрагмента: передаточная функция прямого обхода (f) переводит состояние во входной точке в состояние в выходной точке, а передаточная функция обратного обхода (f^b) переводит состояние в выходной точке в состояние во входной точке.

Для инструкций: $Out[I_j] = f_{I_j}(In[I_j]), \quad In[I_j] = f^b_{I_j}(Out[I_j])$

Для ББ: 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 $I_1,\ldots,I_n$: $f_B(x) = f_{I_n}(f_{I_{n-1}}(\ldots f_{I_1}(x)\ldots)) = (f_{I_1} \circ f_{I_2} \circ \ldots \circ f_{I_n})(x)$ или $f_B = f_{I_1} \circ f_{I_2} \circ \ldots \circ f_{I_n}$ и, соотв., $f^b_B = f^b_{I_n} \circ f^b_{I_{n-1}} \circ \ldots \circ f^b_{I_1}$

21. Определением переменной *x* называется инструкция, которая присваивает значение переменной *x.* Каждое новое определение переменной *x* ***убивает*** ее предыдущее определение.

22. Использованием переменной *x* является инструкция, одним из операндов которой является переменная *x.*

23. 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. Для достигающих определений передаточная функция f_I инструкции *I* может быть записана в виде: $f_I(x) = gen_I \cup (x - kill_I)$

24. Поток данных – все возможные наборы значений переменных, вычисленные в различных точках программы.

25. Живые переменные - переменные, используемые в базовых блоках, в которые попадает управление после выхода из исследуемого базового блока.

25. Живые переменные - переменные, используемые в базовых блоках, в которые попадает управление после выхода из исследуемого базового блока.

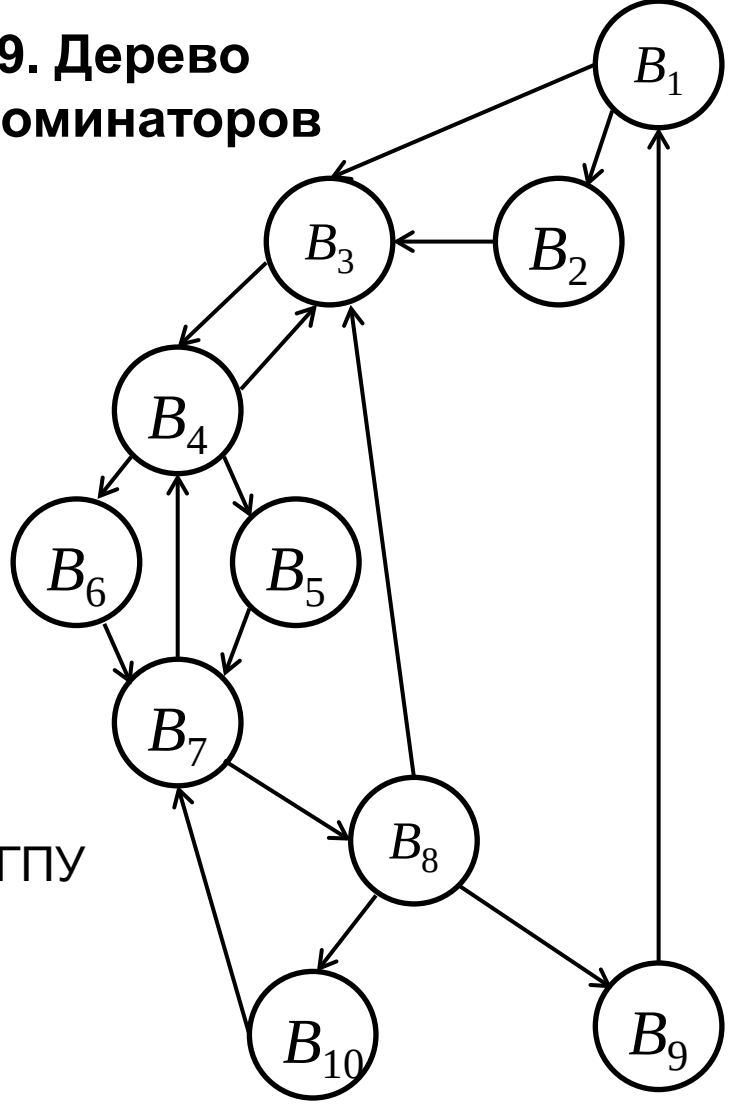
26-27. Существует несколько разновидностей **бесполезного кода** (инструкции которого не влияют на результат вычислений): **мертвый код** – инструкции, результат которых не используется в дальнейших вычислениях; **недостижимый код** – инструкции, которые не содержатся ни в одном реальном пути выполнения; **избыточный код** – инструкции, повторно вычисляющие уже вычисленные значения (напр., доступные выражения или инвариантные вычисления в циклах).

28. Доминаторы. В ГПУ вершина *d* является **доминатором** вершины *n* (этот факт записывается как *d dom n* или *d = Dom*(*n*)), если любой путь от вершины *Entry* до вершины *n* проходит через вершину *d.* Каждая вершина *n* является доминатором самой себя, так как путь от *Entry* до *n* проходит через *n.*

Вершина *i* ГПУ является **непосредственным доминатором** вершины *n* (*i idom n*), если 1) *i dom n, 2*) не существует вершины *m*, $m \neq i$, $m \neq n$, такой что *i dom m* и *m dom n.*

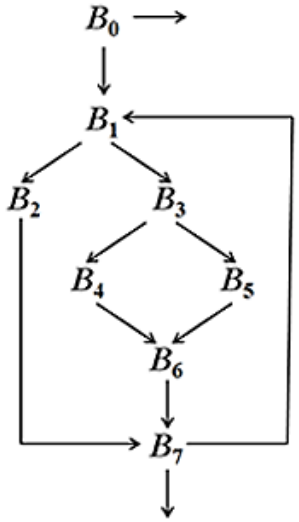
Вершина *s* ГПУ является **строгим доминатором** вершины *n* (*s sdom n*), если *s dom n* и $s \neq n$.

29. Дерево доминаторов

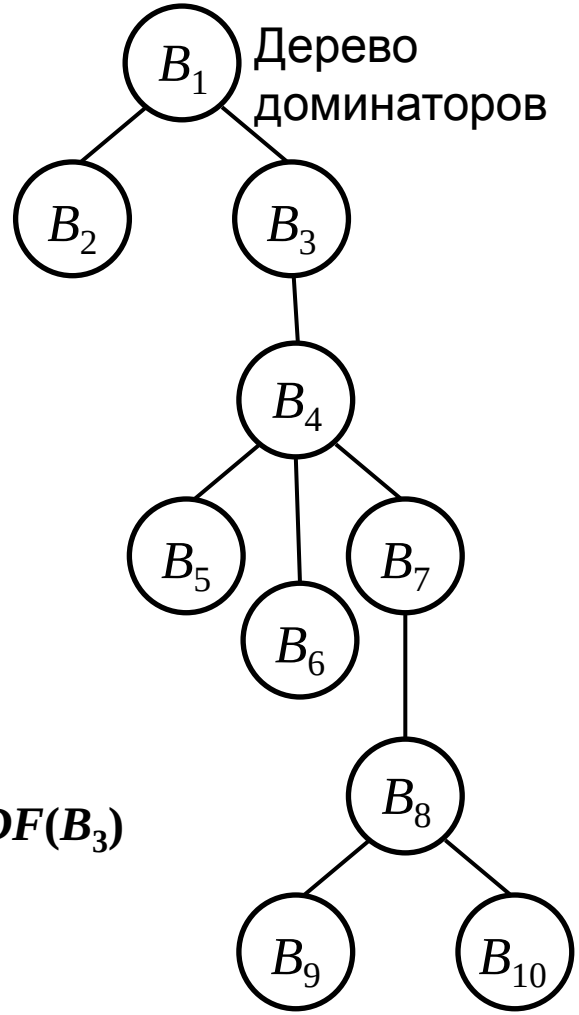


ГПУ

Соединив дугами каждый блок с его непосредственным доминатором, получим дерево доминаторов (*Entry* это блок B_1).



$B_7 \in DF(B_3)$

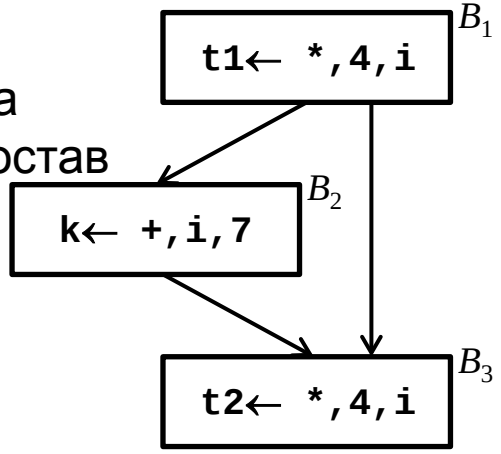


30. Граница доминирования *n (DF*(*n*)) - множество узлов *m*, удовлетворяющих условиям: 1) *n* является доминатором предшественника *m*, т.е. $p \in Pred(m)$ & $n \in Dom(p)$, 2) *n* не является строгим доминатором *m*, т.е. $n \notin (Dom\ (m) - \{m\})$.

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.

31. Постдоминатор: в ГПУ вершина *p* постдоминирует над вершиной *n* (*p* = *Postdom* (*n*)), если каждый путь из вершины *n* в вершину *exit* проходит через вершину *p*. Выходной узел постдоминирует над всеми блоками графа. Постдоминаторы ГПУ – это доминаторы его обратного графа.

32. Обратная граница доминирования (*RDF*(*n*)) вершины $n \in G$ это обычная граница доминирования в обратном графе G^R. **Обратным графом** ориентированного графа $G = \langle N, E \rangle$ называется ориент. граф $G^R = \langle N, E^R \rangle$, у которого направления всех ребер противоположны.



33. 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*. //Пусть точка *p* – вход в блок B_3. На рисунке //присваивание *не убивает* выражение ***, 4, i**. //Поэтому **t2 ← *, 4, i** можно заменить на **t2 ← t1**

34-39. Полурешетка

Полурешетка – это абстрактная алгебраическая структура, над элементами которой определена абстрактная операция $\wedge$ (мы будем называть ее «сбор»), обладающая свойствами операций $\cup$ и $\cap$.

Опр. Полурешетка представляет собой множество L, на котором определена бинарная операция «*сбор*» $\wedge$, такая, что для всех x, y и $z \in L$:

$x \wedge x = x$ (*идемпотентность*), $x \wedge y = y \wedge x$, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$.

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

Полурешеточное отношение частичного порядка: для всех пар $x, y \in L$ определим отношение $\leq$: $x \leq y$ тогда и только тогда, когда $x \wedge y = x$

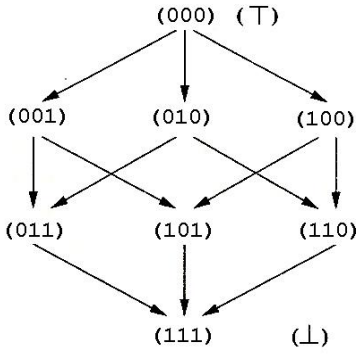
(1) *Рефлексивность* $\leq$ следует из идемпотентности $\wedge$: $x \leq x \Leftrightarrow x \wedge x = x$

(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 $y \wedge z = y$; $(x \wedge z) = x \Leftrightarrow x \leq z$;

$(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) - (x \wedge y) = x$.

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. Диаграмма полурешетки $\langle U, \cup \rangle$, $|U| = 8$: элемент множества U представляется битовым 3-вектором.

40-42. 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, $\wedge$ - операция сбора.

Семейство передаточных функций F называется **замкнутым**, если:1) F содержит тождественную функцию I: $\forall x \in L$: $I(x) = x$, 2) F замкнуто относительно композиции: $\forall f, g \in F \Rightarrow h(x) = g(f(x)) \in F$.

Структура потока данных $\langle D, F, L, \wedge \rangle$ называется **монотонной**, если

$$\forall x, y \in L, \forall f \in F \quad f(x \wedge y) \leq f(x) \wedge f(y).$$

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

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

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

43. 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.

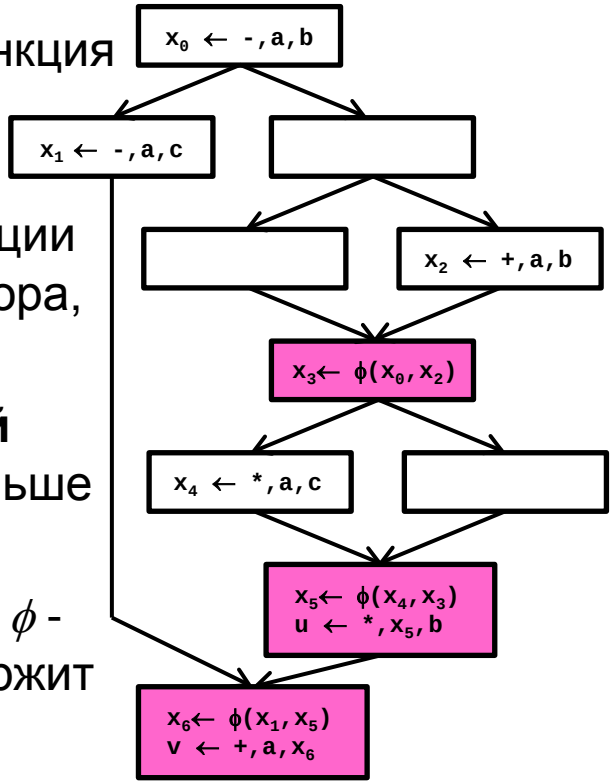
44. Пусть в оптимизируемой процедуре есть инструкция копирования **x ← y. Распространение копий** означает замену всех последующих вхождений переменной **x** на переменную **y**. Каждая команда копирования описывается четверкой $\langle$**x**, **y**, b, $p\rangle$, b – блок, p – строка. Далее определяются и анализируются множество *copy*(b) команд копирования и множество *kill*(b) переопределений **y** …

45. Форма статического единственного присваивания (SSA) позволяет в каждой точке программы объединить 1) информацию об имени переменной

2) с информацией о текущем значении этой переменной (или, что то же самое, с информацией о том, какое из определений данной переменной определяет ее текущее значение в данной точке).

Хотелось бы, чтобы программа в SSA-форме удовлетворяла двум условиям: а) каждое определение переменной имеет индивидуальное имя; б) каждое использование переменной достигало только одно определение.

46. ϕ-функция - «функция» объединения значений. 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.



47. Построенная SSA-форма называется **максимальной SSA-формой**, так как она обычно содержит намного больше ϕ-функций, чем необходимо (в каждой точке сбора).

48. Частично усеченная SSA-форма содержит меньше ϕ-функций, чем максимальная (но, к сожалению, она содержит не минимально возможное количество ϕ-функций).

Чтобы не всегда вставлять ϕ-функции необходимо **для каждой точки сбора** уметь выяснять, какие переменные нуждаются в ϕ-функциях ИЛИ **для каждого определения переменной** уметь находить множество всех точек сбора, которые нуждаются в ϕ-функциях для значения, порожденного этим определением.

49. Натуральный (или **естественный) цикл** - цикл со следующими свойствами:

1) цикл имеет единственный входной узел, называемый его **заголовком**,
2) существует обратное ребро, ведущее в заголовок цикла

Натуральный цикл обратного ребра $\langle B_i, B_k \rangle$ 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**).

50. Инструкция **инвариантна относительно цикла**, если она удовлетворяет одному из следующих условий: 1) ее операнды – константы, 2) все определения операндов, достигающие инструкции, находятся вне цикла, 3) внутри цикла имеется в точности одно определение операнда, но оно само инвариантно относительно цикла.

51. Бесполезный код – см. 26-27.

53. Машинно-независимая оптимизация: 1) простые оптимизации: сворачивание констант, алгебраические упрощения и перегруппировка, распространение копий; 2) оптимизация циклов; 3) исключение бесполезного кода; 4) оптимизация потока управления.

Машинно-ориентированная оптимизация: генерация кода: 1) выбор команд, 2)распределение регистров, 3) планирование кода.

-Входной поток: промежуточное представление исходной программы (последовательность трехадресных инструкций) + таблица символов.

-Выходной поток: объектный код (последовательность команд целевого процессора.

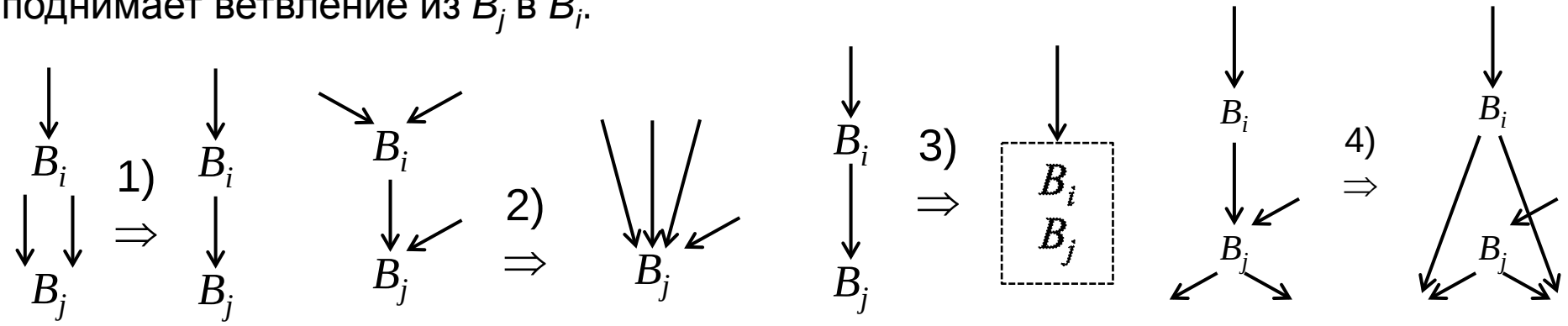
52. Оптимизация потока управления: некоторые оптимизации могут иметь побочный эффект, изменяющий форму ГПУ, добавляя в него бесполезные блоки и дуги. Если компилятор содержит такие оптимизации, он должен также содержать проход, упрощающий ГПУ, исключая бесполезный поток управления. Функция **Clean()**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

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_i , то ветвление заменяется безусловным переходом на блок B_i

2) удалить пустой блок: если блок B_i содержит только инструкцию перехода, то он поглощается своим последователем – блоком B_j.

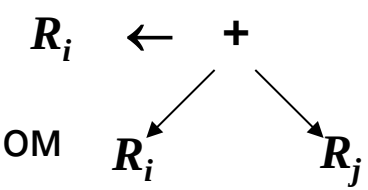
3) объединение блоков: если имеется блок B_i, который оканчивается переходом на B_j, у которого всего один предшественник, B_i, он может объединить эти блоки как показано на рисунке внизу, что позволяет исключить переход из B_i в B_j.

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.



54. Метод переписывания дерева.

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* ← *template* {*action*}, где *replacement* – отдельный узел, *template* – шаблон (поддерево), *action* – действие (фрагмент генерируемого кода, соответствующий шаблону). Множество правил преобразования дерева именуется схемой трансляции дерева. **{ADD Ri, Ri, Rj}**



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**.

54. Гнездо циклов *** - композиция нескольких циклов (один в другом)**.**

◊ *Распределение регистров*

отображает неограниченное множество имен (псевдорегистров) на конечное множество физических регистров целевой машины, Это NP-полная задача

55. Локальное распределение регистров (1)

◊ *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 к другим 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

55. Локальное распределение регистров (2)

Реализация функции *getReg*

- Выбор регистра R_y для операнда **y**
  - 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 представляет собой переменную x, вычисляемую 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**

55. Локальное распределение регистров (3)

Реализация функции *getReg*

- Выбор регистра R_x для результата **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

  x ← y

  Сначала выбирается R_y, как и для операнда инструкции **x ← op, y, z**, после чего полагается $R_x = R_y$.

56. Глобальное распределение регистров

- ◊ При распределении регистров моделируется состязание за место на регистрах целевой машины.
  Рассмотрим два различных интервала жизни LR_i и LR_j. Если в программе существуют команды, во время которых и LR_i, и LR_j актуальны, то 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 интервалам жизни, а дуги соединяют интервалы жизни, 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.

57. Планирование кода

◊Цель планирования кода – выбрать такую последовательность команд, которая, не меняя семантики программы, обеспечит оптимальное использование особенностей архитектуры целевого процессора

◊ Прежде всего, это необходимость обеспечить правильно использование возможностей параллельного выполнения команд, реализованных в аппаратуре данного целевого процессора

◊ 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 машины.

59-60. Зависимости по управлению

◊ Команда **`s1`** *зависит 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`**

62. 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 по данным:
  - *Истинная зависимость*: **чтение после записи**.
    Если команда C_1 записывает значение в некоторую ячейку памяти (или на регистр), 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.

63. Переименование значений, чтобы избежать антизависимостей.

Значения переименовываются таким образом, чтобы каждое значение имело уникальное имя. Это нужно для того, чтобы найти и те расписания, которые были бы исключены из-за антизависимостей.

64. Требование консервативности анализа

◊ **Консервативность**. Компилятор обязан считать, что две команды могут обращаться к одним и тем же ячейкам, если он не может доказать обратное.

◊ **Пример**: Рассмотрим код

```
1. a ← 1
2. *p ← 2
3. b ← a
```

Сразу можно обнаружить истинную зависимость между командами 1 и 3.

Больше зависимостей как будто нет, но это только в том случае, если *компилятор может доказать,* что указатель **p** не может указывать на **a**.

В противном случае, 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.

65. В чем состоит анализ потока данных ***

При анализе потока данных рассматриваются множества переменных для описания состояния и такие операции как объединение и пересечение множеств.
Структурой потока данных называется четверка <D, F, L, ∧> , где
D – направление анализа (Forward или Backward), F – семейство передаточных функций, L – поток данных, ∧ - операция сбора.
На выходе получаем значения из L для In[B] и Out[B] для каждого блока B в графе потока.

66-69. Дуги 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.

70. 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 полезные инструкции.
  - на втором проходе (*Sweep*) непомеченные инструкции удаляются.