6. Машинно-независимая оптимизация

6.1. Простые оптимизации

6.1.1 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*.
- ◊ В случае булевских вычислений исключительные ситуации не возбуждаются и потому таких проблем не возникает.

6.1. Простые оптимизации

6.1.1 Сворачивание констант

- ◊ 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.

6.1. Простые оптимизации

6.1.2 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 $Boolean$):

$$b \vee true = true \vee b = true$$
$$b \vee false = false \vee b = b$$
$$b \,\&\, true = true \,\&\, b = b$$
$$b \,\&\, false = false \,\&\, b = false$$

6.1. Простые оптимизации

6.1.3 Распространение копий

◊ Пусть в оптимизируемой процедуре есть инструкция копирования

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

Распространение копий означает замену всех последующих вхождений переменной **x** на переменную **y**.

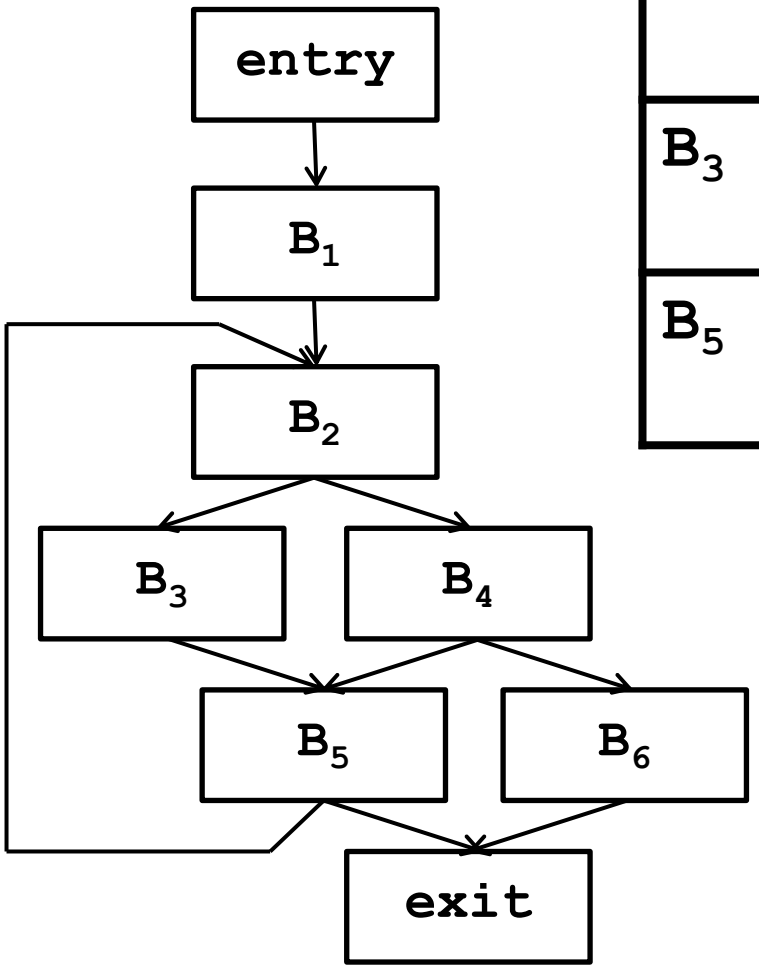
◊ Рассмотрим множество всех команд копирования анализируемой процедуры. Каждая команда копирования описывается четверкой $\langle \mathbf{x}, \mathbf{y}, b, p \rangle$, где **x** и **y** представляют инструкцию копирования $\mathbf{x} \leftarrow \mathbf{y}$, находящуюся в строке p базового блока b.

Множество всех таких четверок обозначим через U. Множество U содержит все инструкции копирования анализируемой процедуры.

◊ Пример. В процедуре, представленной на следующем слайде, имеется две команды копирования

6.1. Простые оптимизации

6.1.3 Распространение копий



B_1 c ← +,a, b d ← c e ← *,d, d	B_2 f ← +,a, c g ← e a ← +,g, d if(a<c)goto
B_3 h ← +,g, 1	B_4 f ← -,d, g if(f>a)goto
B_5 b ← *,g, a if(f>h)goto	B_6 c ← 2

◊ **d ← c** (2-я строка блока B_1):

четверка $\langle$**d**, **c**, B_1, 2$\rangle$

◊ **g ← e** (2-я строка блока B_2):

четверка $\langle$**g**, **e**, B_2, 2$\rangle$

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

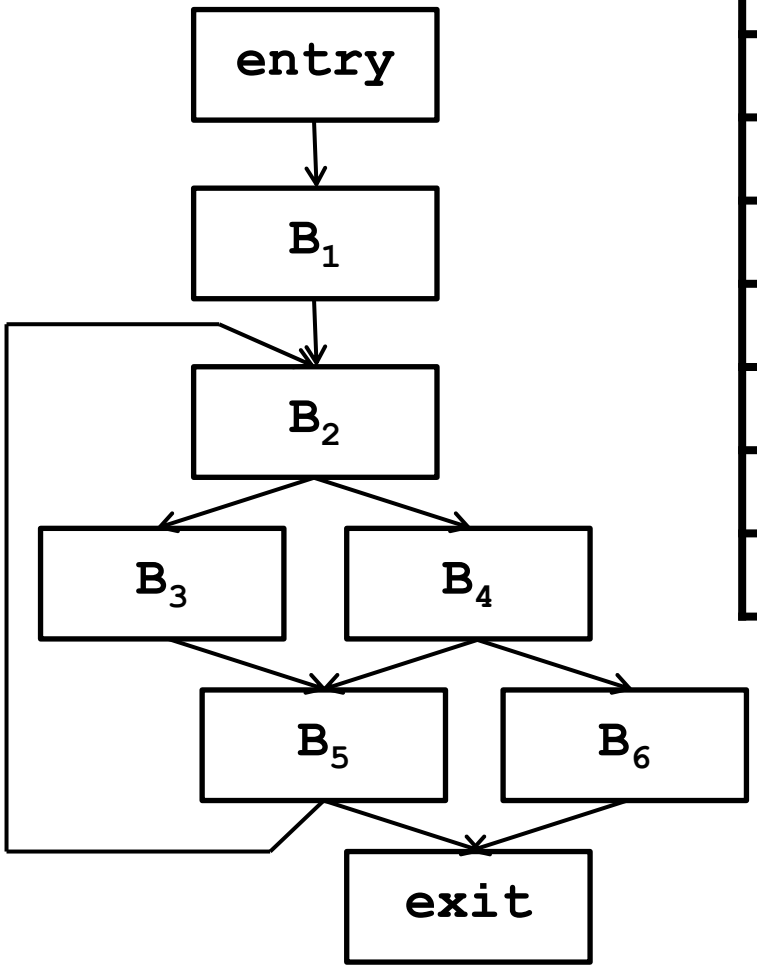
6.1. Простые оптимизации

6.1.3 Распространение копий

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

6.1. Простые оптимизации

6.1.3 Распространение копий



Блок	copy	kill
entry	∅	∅
B_1	{⟨d, c, B_1, 2⟩}	{⟨g,e,B_2,2⟩}
B_2	{⟨g,e,B_2,2⟩}	∅
B_3	∅	∅
B_4	∅	∅
B_5	∅	∅
B_6	∅	{⟨d, c, B_1, 2⟩}
exit	∅	∅

⟨g,e,B_2,2⟩ убивается в блоке B_1, так как в 3-ей строке этого блока переопределяется **e**

⟨d, c, B_1, 2⟩ убивается в блоке B_6, так как единственная инструкция этого блока переопределяет **c**

6.1. Простые оптимизации

6.1.3 Распространение копий

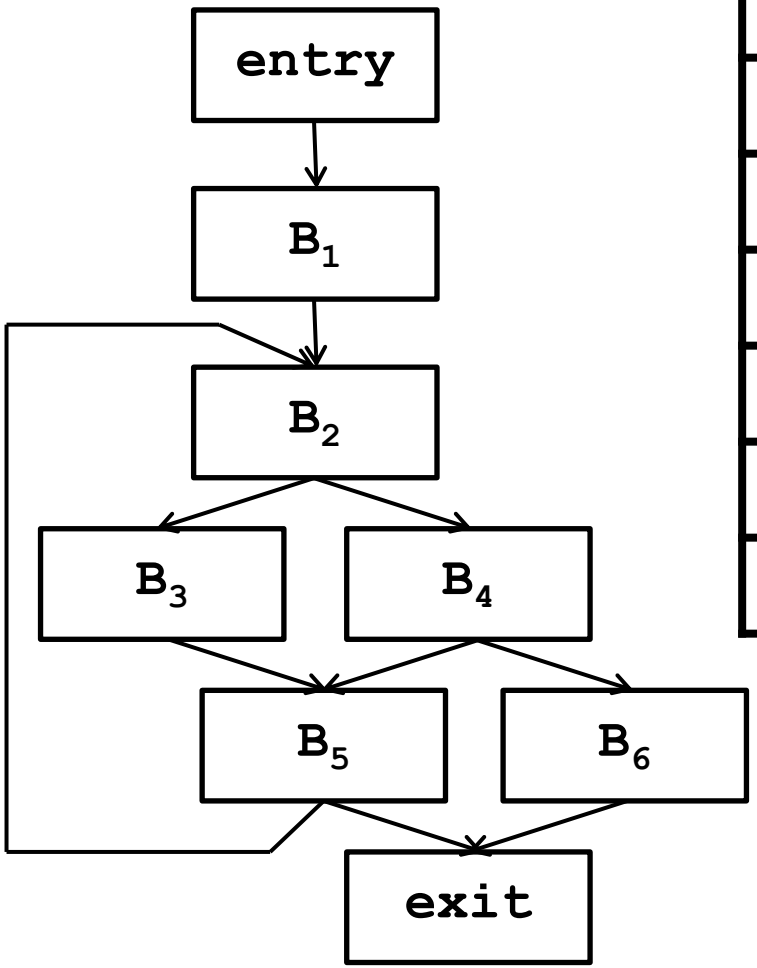
- Система уравнений составляется по аналогии с системой уравнений для достигающих определений:
  - сначала из множества инструкций копирования удаляются инструкции «убитые» в блоке b, потом в него добавляются инструкции копирования блока b.
  - Второе уравнение содержит операцию пересечения, так как по всем путям должны приходить одинаковые копии.

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

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

6.1. Простые оптимизации

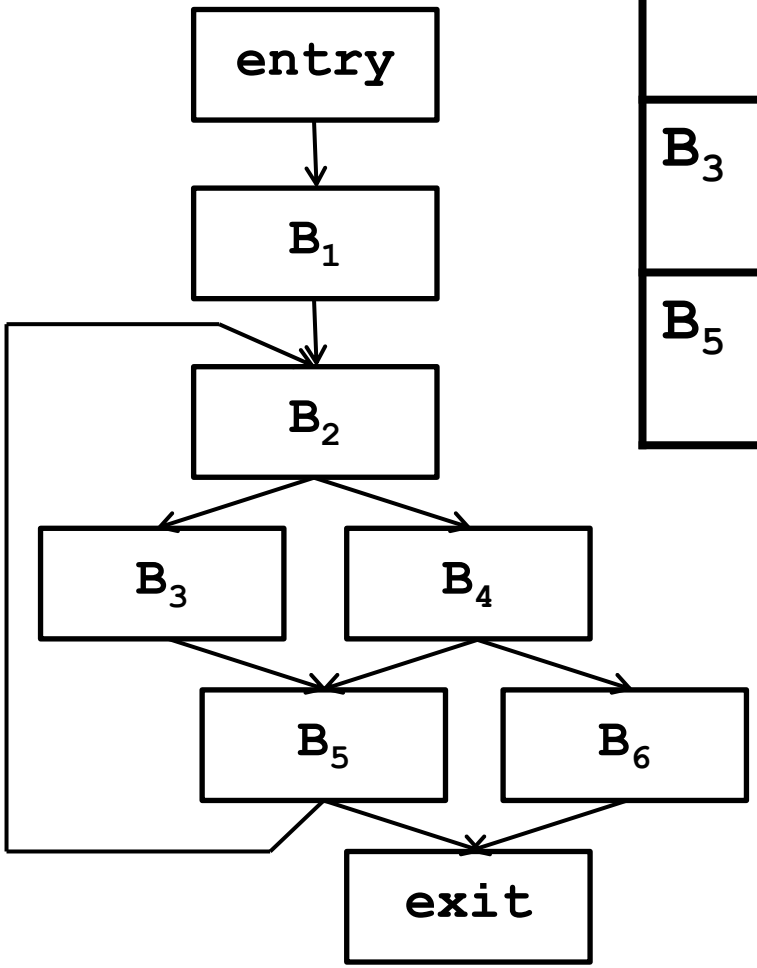
6.1.4 Распространение копий



Блок	In_{CP}
Entry	$\varnothing$
B_1	$\varnothing$
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	$\{\langle g, e, B_2, 2\rangle\}$

6.1. Простые оптимизации

6.1.3 Распространение копий



B_1 c ← +,a, b d ← c e ← *,c, c	B_2 f ← +,a, c g ← e a ← +,e, c if(a<c)goto
B_3 h ← +,e, 1	B_4 f ← -,c, e if(f>a)goto
B_5 b ← *,e, a if(f>h)goto	B_6 c ← 2

6.2. Оптимизация циклов

6.2.1 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 (их, а также обратно направленные дуги любят великие Фортранные программисты, но и у них они постепенно выходят из моды).

6.2. Оптимизация циклов

6.2.2 Натуральные циклы

◊ **Определение**. *Натуральным* (или *естественным*) *циклом* называется цикл со следующими свойствами:

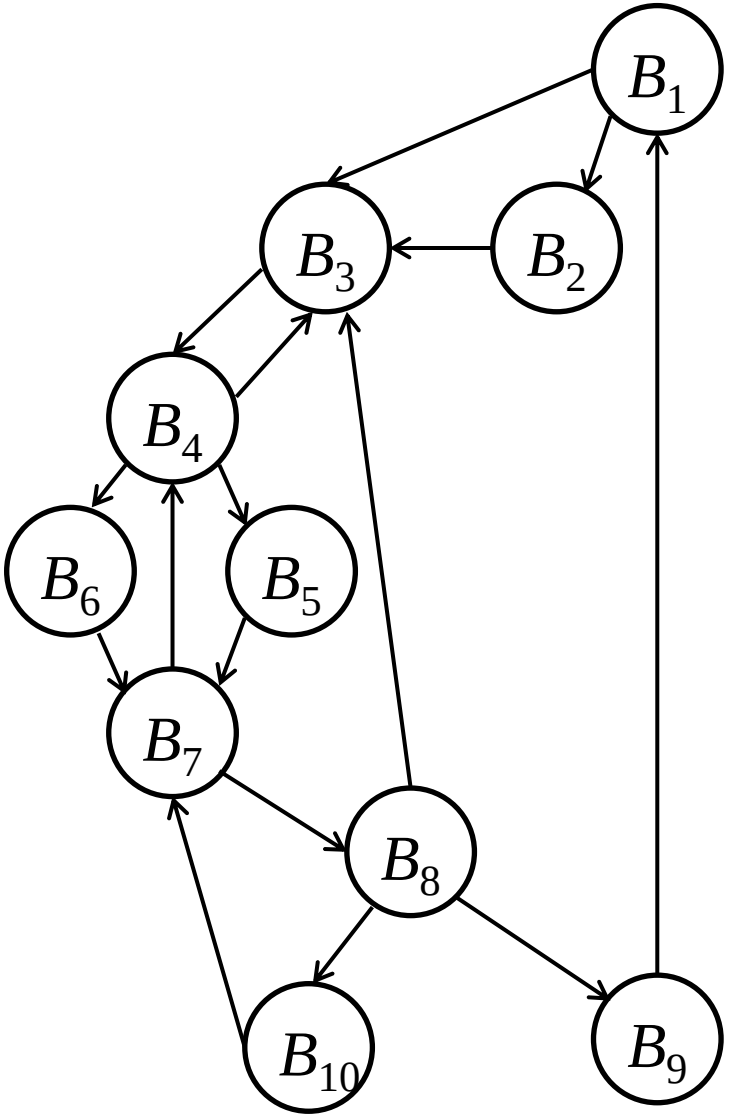
- ◈ Цикл имеет единственный входной узел, называемый его *заголовком*,
- ◈ Существует обратное ребро, ведущее в заголовок цикла

◊ **Определение**. *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*).

6.2. Оптимизация циклов

6.2.3 Натуральные циклы. Пример.

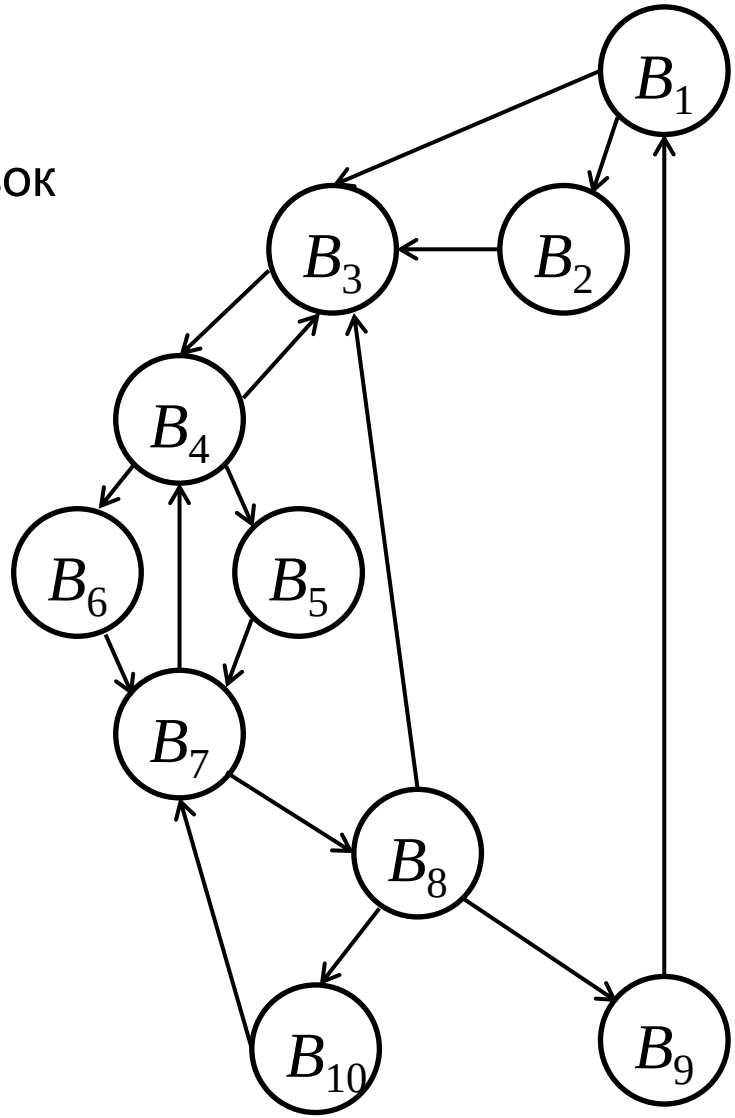
- На рисунке справа – пять обратных дуг: $\langle B_{10}, B_7\rangle$, $\langle B_7, B_4\rangle$, $\langle B_4, B_3\rangle$, $\langle B_8, B_3\rangle$, $\langle B_9, B_1\rangle$
- Обратной дуге $\langle B_{10}, B_7\rangle$ соответствует натуральный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и B_{10} можно достичь вершины B_{10}, не проходя через B_7.
- Обратной дуге $\langle B_7, B_4\rangle$ соответствует натуральный 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



6.2. Оптимизация циклов

6.2.3 Натуральные циклы. Пример.

◊ У натуральных 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 этот цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$



6.2. Оптимизация циклов

6.2.3 Натуральные циклы. Пример.

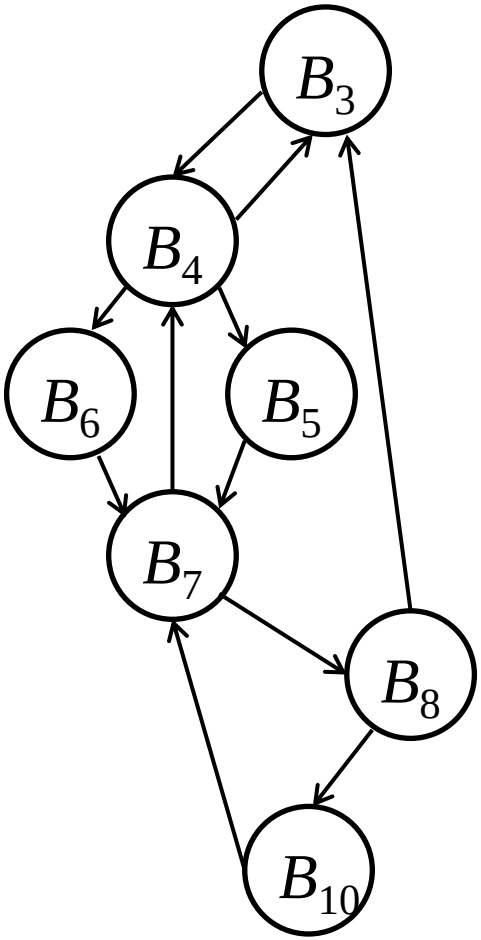
◊ У натуральных циклов обратных дуг

$\langle B_4, B_3 \rangle$ 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 этот объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$



6.2. Оптимизация циклов

6.2.4 Алгоритм построения натурального цикла по обратной дуге

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

Обратная дуга $e = \langle n, d \rangle \in E$

Выход: подграф $C \subseteq G$, являющийся натуральным 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 графе потока (направления дуг заменены на противоположные).

(4) все узлы, посещенные на шаге (3), добавляются в C.

6.2. Оптимизация циклов

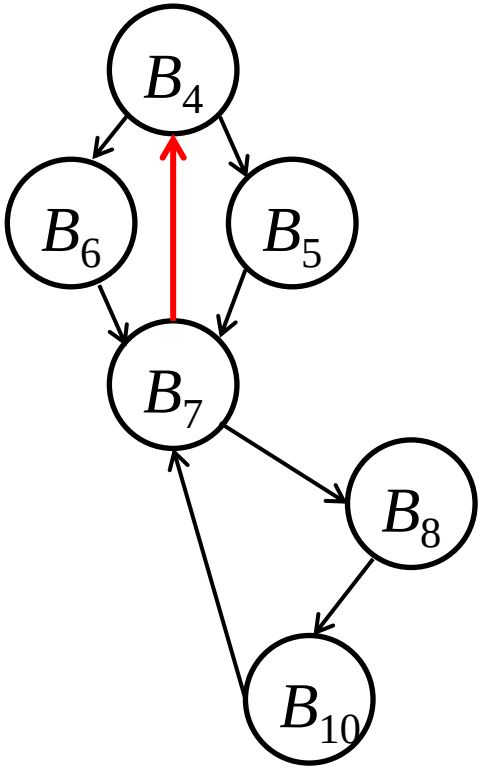
6.2.5 Построение натурального цикла по обратной дуге. Пример

◊ Применим алгоритм 6.2.4 для построения натурального 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 $Pred\ (B_7) = \{B_5, B_6, B_{10}\}$

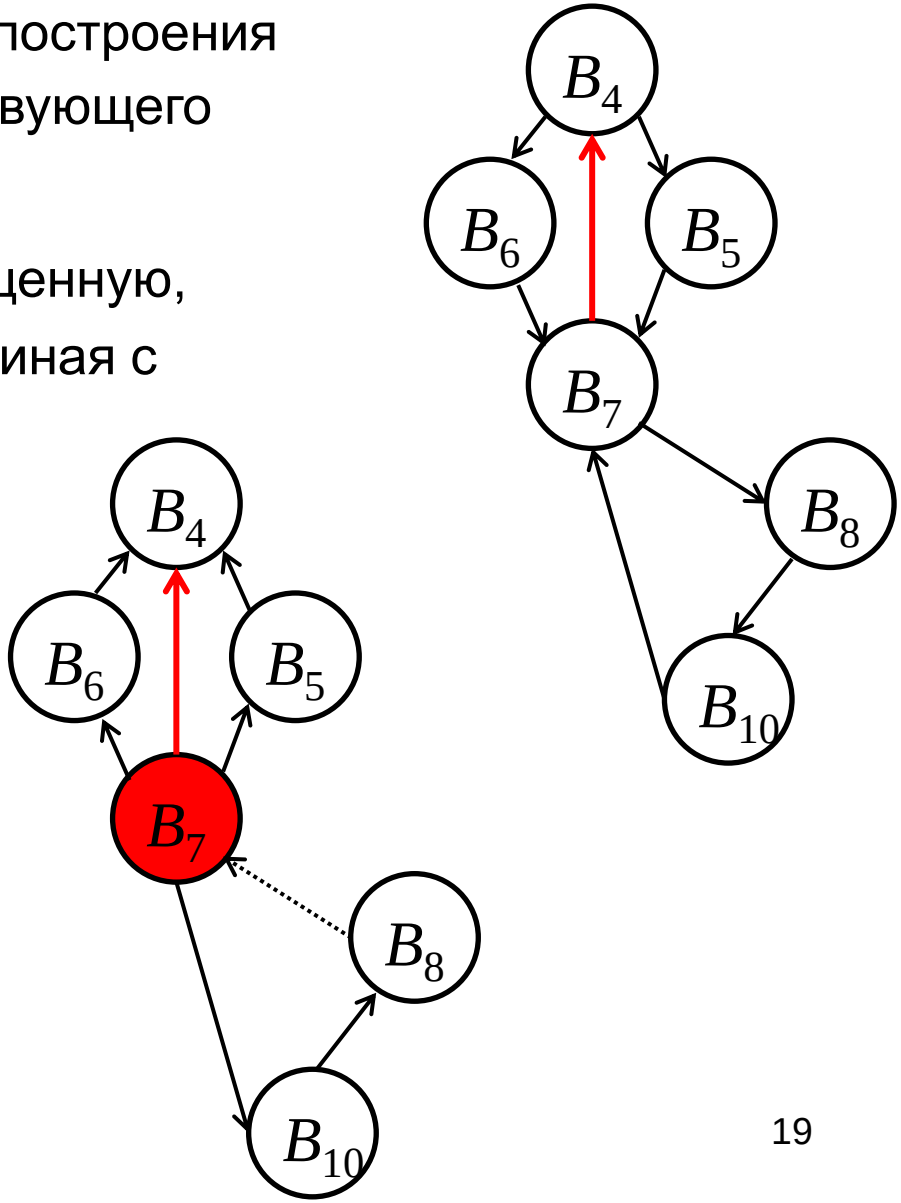


6.2. Оптимизация циклов

6.2.5 Построение натурального цикла по обратной дуге. Пример

◊ Применим алгоритм 6.2.4 для построения натурального 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.



6.2. Оптимизация циклов

6.2.6 Перемещение кода, инвариантного относительно цикла

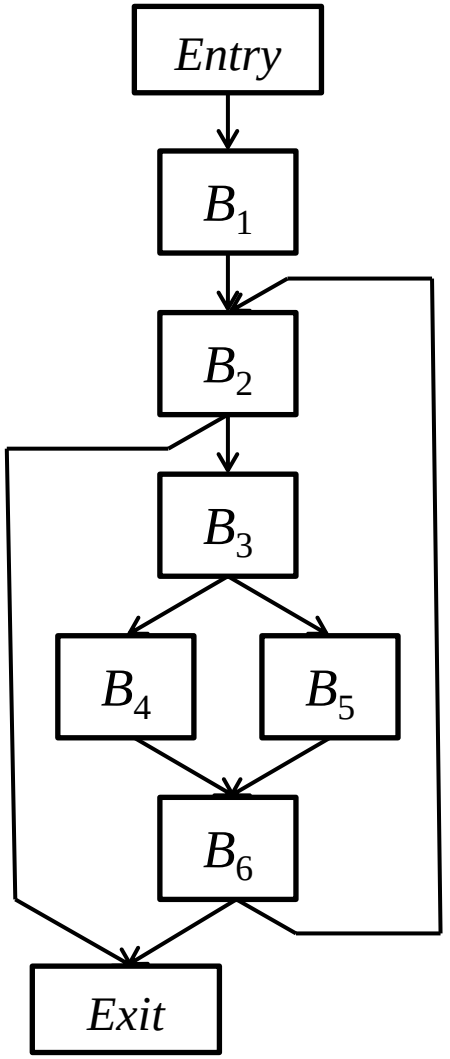
- Инструкция *инвариантна относительно цикла*, если она удовлетворяет одному из следующих условий:
  - ее операнды – константы
  - все определения операндов, достигающие инструкции находятся вне цикла
  - внутри цикла имеется в точности одно определение операнда, но оно само инвариантно относительно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

6.2. Оптимизация циклов

6.2.6 Перемещение кода, инвариантного относительно цикла

◊ Рассмотрим пример: цикл с заголовком B_2.

B_1 `b←2` `i←1`	B_2 `if(i>100)`	B_3 `a←b+1` `c←2` `if(i%2=0)`
B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(a<2)`



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

◊ Инструкции **`a←b+1, c←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.

6.2. Оптимизация циклов

6.2.6 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.

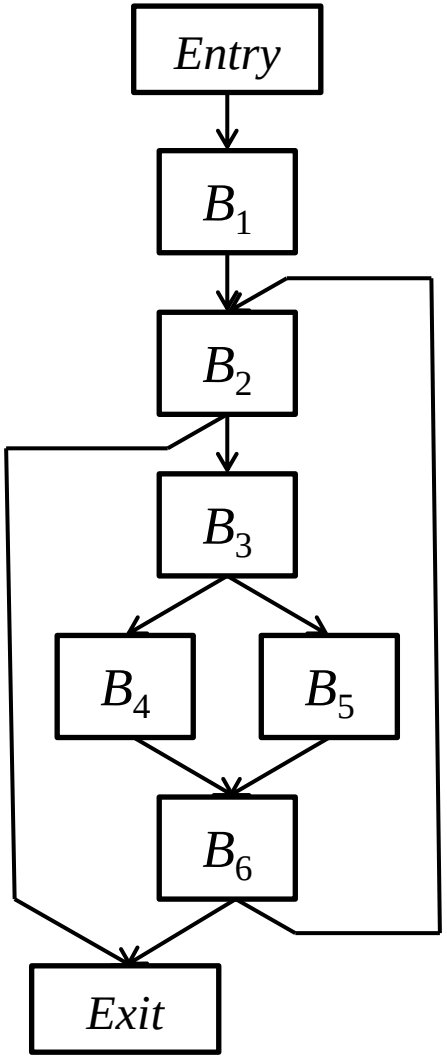
6.2. Оптимизация циклов

6.2.6 Перемещение кода, инвариантного относительно цикла

B_1 `b←2`
`i←1`
	B_2 `if(i>100)`	B_3 `a←b+1`
		`c←2`
		`if(i%2=0)`

B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(a<2)`

Исходный цикл



6.2. Оптимизация циклов

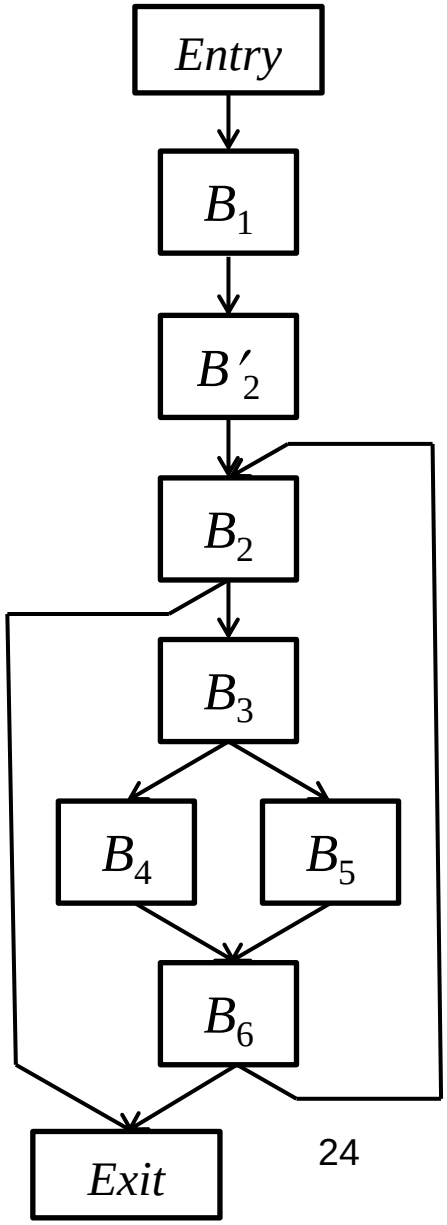
6.2.6 Перемещение кода, инвариантного относительно цикла

B_1 `b←2` `i←1`	B_2 `if(i>100)`	B'_2

	B_3 `a←b+1`
	`c←2`
	`if(i%2=0)`	

B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(a<2)`

После добавления предзаголовка (B'_2)



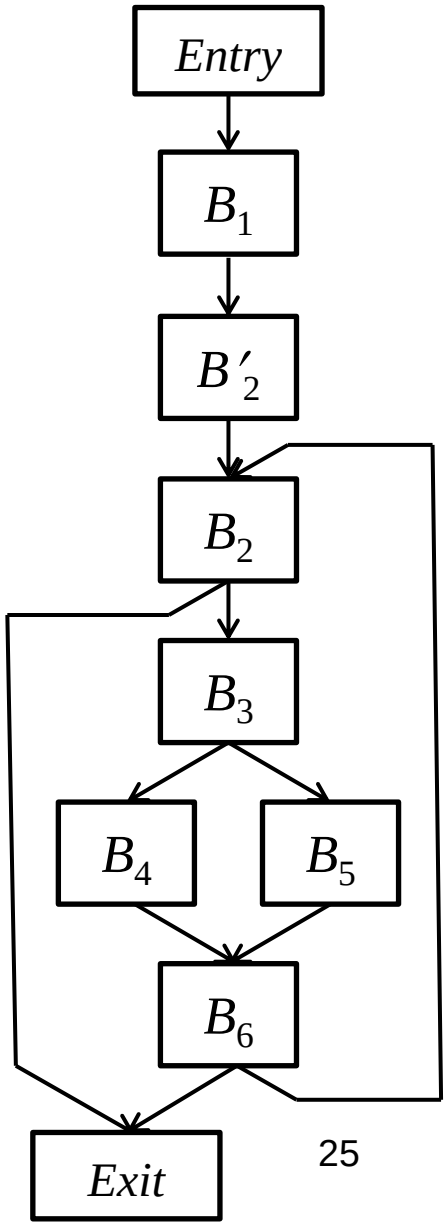
6.2. Оптимизация циклов

6.2.6 Перемещение кода, инвариантного относительно цикла

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

Выделение инвариантных инструкций:

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

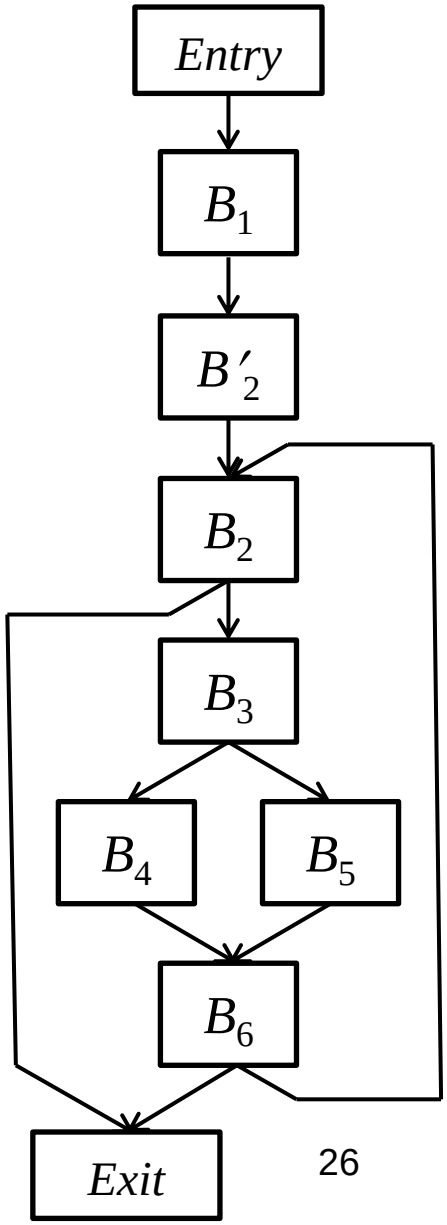


6.2. Оптимизация циклов

6.2.6 Перемещение кода, инвариантного относительно цикла

B_1 `b←2` `i←1`	B_2 `if(i>100)`	B'_2 `a←b+1` `c←2` `t1←a<2`
	B_3 `if(i%2=0)`	
B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(t1)`

После вынесения инвариантного кода в предзаголовок



6.3. Исключение бесполезного кода

6.3.1 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 (в частности, недостижимый и мертвый) код

6.3. Исключение бесполезного кода

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

6.3. Исключение бесполезного кода

6.3.3 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.

6.3. Исключение бесполезного кода

6.3.4 Постдоминаторы

◊ *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 ($p = Postdom\ (n)$) , 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.

6.3. Исключение бесполезного кода

6.3.5 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 $\forall k \in T - \{n\}: m = Postdom(k)$, 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.

6.3. Исключение бесполезного кода

6.3.6. 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.

6.3. Исключение бесполезного кода

6.3.6. 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.

◈

6.3. Исключение бесполезного кода

6.3.6. Проход *Mark* (псевдокод)

```
Mark( )
WorkList ← ∅;
for each инструкции i (x ← op, y, z ::: пометка)
      убрать пометку у i
      if (i полезная) пометить i
      WorkList ← WorkList ∪ {i}
while (WorkList ≠ ∅)
      remove i from WorkList
      if (def(y) не помечена)
            пометить def(y)
            WorkList ← WorkList ∪ {def(y)}
      if (def(z) не помечена)
            пометить def(z)
            WorkList ← WorkList ∪ {def(z)}
      for each block b ∈ RDF(block(i))
            пусть j ветвь, оканчивающаяся в b
            if (j не помечена)
                  пометить j
                  WorkList ← WorkList ∪ {j}
```

6.3. Исключение бесполезного кода

6.3.6 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.

6.3. Исключение бесполезного кода

6.3.6 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.

6.4. Исключение недостижимого кода

6.4.1 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){...}`)
- В этом курсе 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.

6.4. Исключение недостижимого кода

6.4.2 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 непомеченные блоки действительно недостижимы и могут быть удалены на проходе $Sweep$.
- ◊ В случае неоднозначных условий ветвления, компилятор должен сохранить любой блок, достижимый ветвлением или переходом.

6.5. Оптимизация потока управления

6.5.1. Постановка задачи

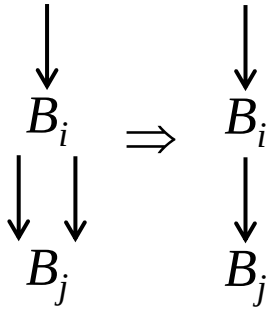
- ◊ Некоторые оптимизации могут иметь побочный эффект, изменяющий форму ГПУ, добавляя в него бесполезные блоки и дуги.
  Если компилятор содержит такие оптимизации, он должен также содержать проход, упрощающий ГПУ, исключая бесполезный поток управления.
- ◊ Функция *Clean* обрабатывает непосредственно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

6.5. Оптимизация потока управления

6.5.2. 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.



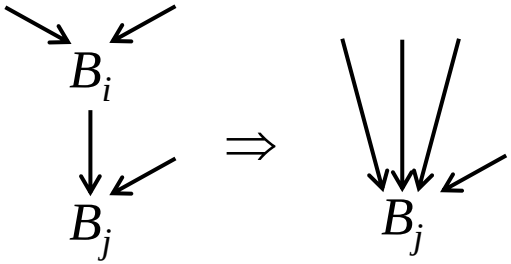
Такая ситуация может возникнуть в результате других оптимизаций (**Например**, у B_i могло быть два последователя, каждый из которых заканчивался переходом на B_j. Если другие оптимизации убрали из этих блоков все инструкции, то второе из основных преобразований могло породить левый граф, показанный на рисунке).

6.5. Оптимизация потока управления

6.5.2. 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.

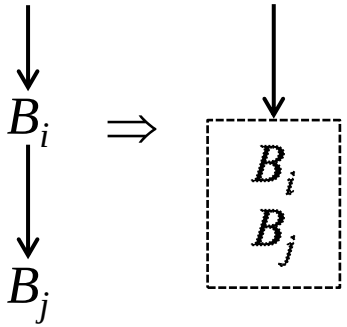


Так как у B_i всего один последователь, B_j, преобразование перенаправляет дуги, входящие в B_i, к B_j и исключает B_i из $Pred(B_j)$, что упрощает ГПУ и ускоряет выполнение.

6.5. Оптимизация потока управления

6.5.2. Основные преобразования. Преобразование 3.

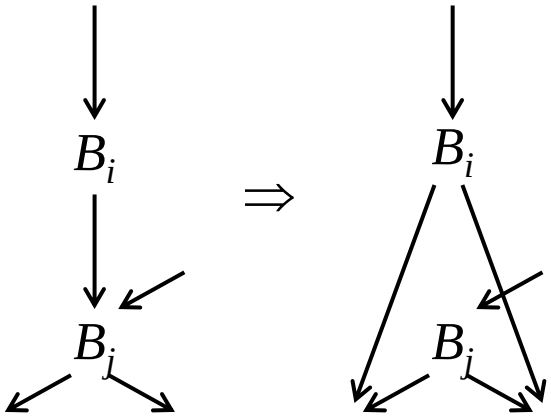
- 3. *Объединение блоков:* Если имеется блок B_i, который оканчивается переходом на B_j, у которого всего один предшественник, B_i, он может объединить эти блоки как показано на рисунке внизу, что позволяет исключить переход из B_i в B_j.



6.5. Оптимизация потока управления

6.5.2. 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).



6.5. Оптимизация потока управления

6.5.3. Функция Clean()

```
Clean()
while ГПУ продолжает изменяться
compute
  Postorder
  OnePass()

OnePass()
for each block Bi              || in postorder
   if (Bi оканчивается ветвлением)
     if (обе цели одинаковы)
        заменить ветвление на переход                    /* 1 */
   if (Bi оканчивается переходом на Bj)
     if (Bi пуст)
        заменить все переходы на Bi переходами на Bj     /* 2 */
     if (Bj имеет только одного предшественника)
        совместить Bi и Bj                               /* 3 */
     if (Bj пуст и оканчивается ветвлением)
        заменить Bi переход
       на копию ветвления из Bj                          /* 4 */
```

6.5. Оптимизация потока управления

6.5.3. Функция `Clean()`

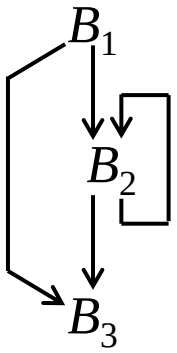
- ◊ Функция **`Clean()`** многократно вызывает функции **`Postorder()`** (обычная нумерация блоков) и **`OnePass()`** (однократный проход), выполняя последовательность преобразований 1 – 4 итеративно до тех пор, пока ГПУ оптимизируемой процедуры продолжает изменяться.
- ◊ В начале каждой итерации выполняется 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.

6.5. Оптимизация потока управления

6.5.4. 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.



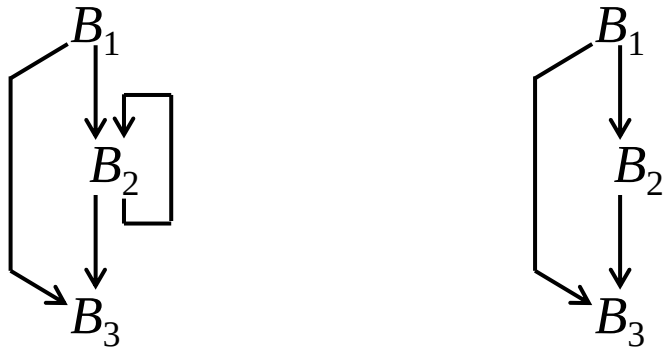
Исходный ГПУ

6.5. Оптимизация потока управления

6.5.4. 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.

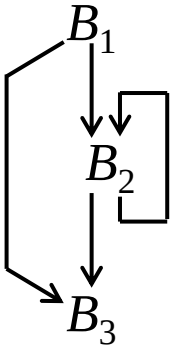


Исходный ГПУ ГПУ после $Mark\ \&\ Sweep$

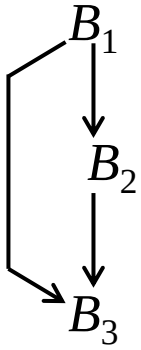
6.5. Оптимизация потока управления

6.5.4. Пример

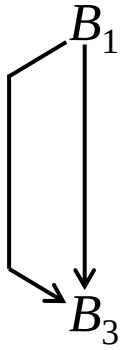
- ◊ *Clean* загоняет B_2 в B_1, в результате получается ГПУ, изображенный на третьем рисунке слева.
- ◊ Теперь ветвление в конце B_1 становится избыточным, и *Clean* заменяет его переходом, в результате получается ГПУ, изображенный на четвертом рисунке слева.
- ◊ Наконец, если окажется, что B_1 – единственный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



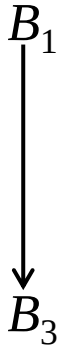
Исходный ГПУ



ГПУ после *Mark & Sweep*



ГПУ после удаления B_2.



ГПУ после сворачивания избыточной ветви.