5. Машинно-независимая оптимизация

5.1. Простые оптимизации

5.1.1 Сворачивание констант

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

  Прежде всего это связано с *исключительными ситуациями*.

5.1. Простые оптимизации

5.1.1 Сворачивание констант

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

5.1. Простые оптимизации

5.1.2 Алгебраические упрощения и перегруппировка

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

$$i + 0 = 0 + i = 1 - 0 = i$$

$$0 - i = -i$$

$$i * 1 = 1 * i = \frac{i}{1} = i$$

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

$$-(-i) = i$$

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

- Применение тождеств (b – переменная типа $Boolean$):

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

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

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

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

5.1. Простые оптимизации

5.1.3 Распространение копий

- Пусть в оптимизируемой процедуре есть инструкция копирования

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

- Распространение копий означает замену всех последующих вхождений переменной **x** на переменную **y**.
- Алгоритм распространения копий состоит из двух основных этапов:
  1) Решение задачи анализа потока данных для нахождения множества доступных копий на входе базовых блоков.
  2) Локальное распространение найденных на 1-м этапе копий внутри каждого базового блока (при этом также выполняется распространение локальных копий, действующих только внутри базового блока).

5.1. Простые оптимизации

5.1.3 Распространение копий

- Пусть в оптимизируемой процедуре есть инструкция копирования

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

- Распространение копий означает замену всех последующих вхождений переменной **x** на переменную **y**.
- Для описания структуры потока данных (для задачи распространения копий):
  1. Зададим область определения структуры потока данных (множество значений потока данных)
  2. Определим семейство передаточных функций
  3. Сформулируем уравнения потока данных
- Получившаяся задача распространения копий решается итеративным алгоритмом.

5.1. Простые оптимизации

5.1.3 Распространение копий

- Пусть в оптимизируемой процедуре есть инструкция копирования

$$\mathtt{x} \leftarrow \mathtt{y}$$

- Распространение копий означает замену всех последующих вхождений переменной **x** на переменную **y**.
- Область определения – множество всех команд копирования
- Рассмотрим множество всех команд копирования анализируемой процедуры. Каждая команда копирования описывается четверкой $\langle x, y, b, p \rangle$, где **x** и **y** представляют инструкцию копирования $\mathtt{x} \leftarrow \mathtt{y}$, находящуюся в строке p базового блока b.

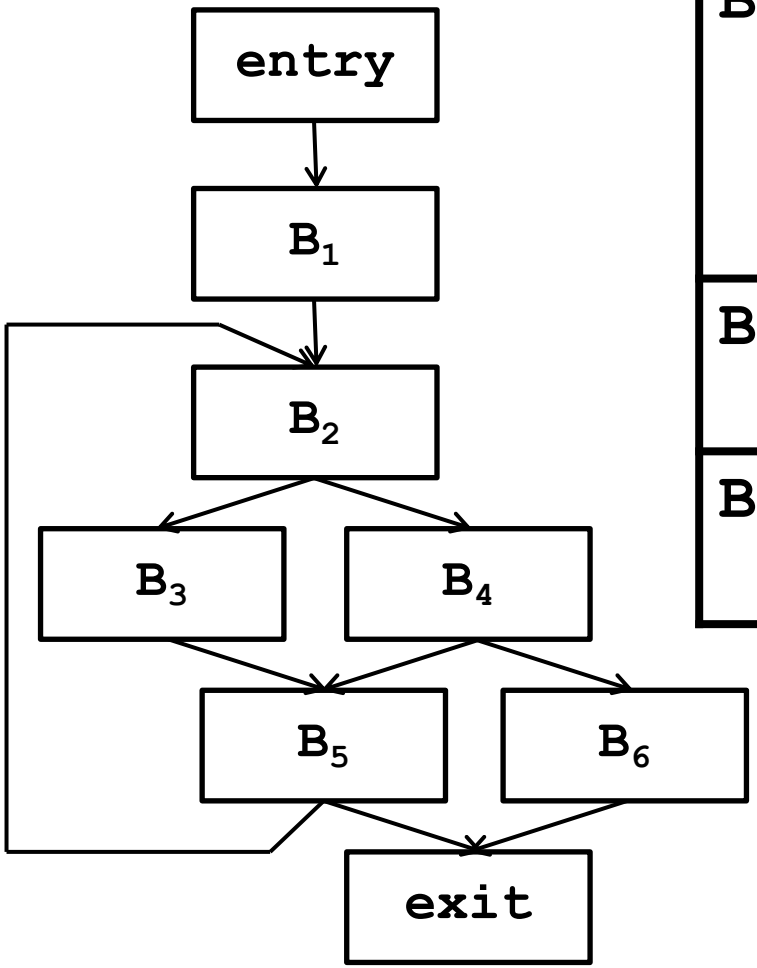
  Множество всех таких четверок обозначим через U.

  Множество U содержит все инструкции копирования анализируемой процедуры.

5.1. Простые оптимизации

5.1.3 Распространение копий

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

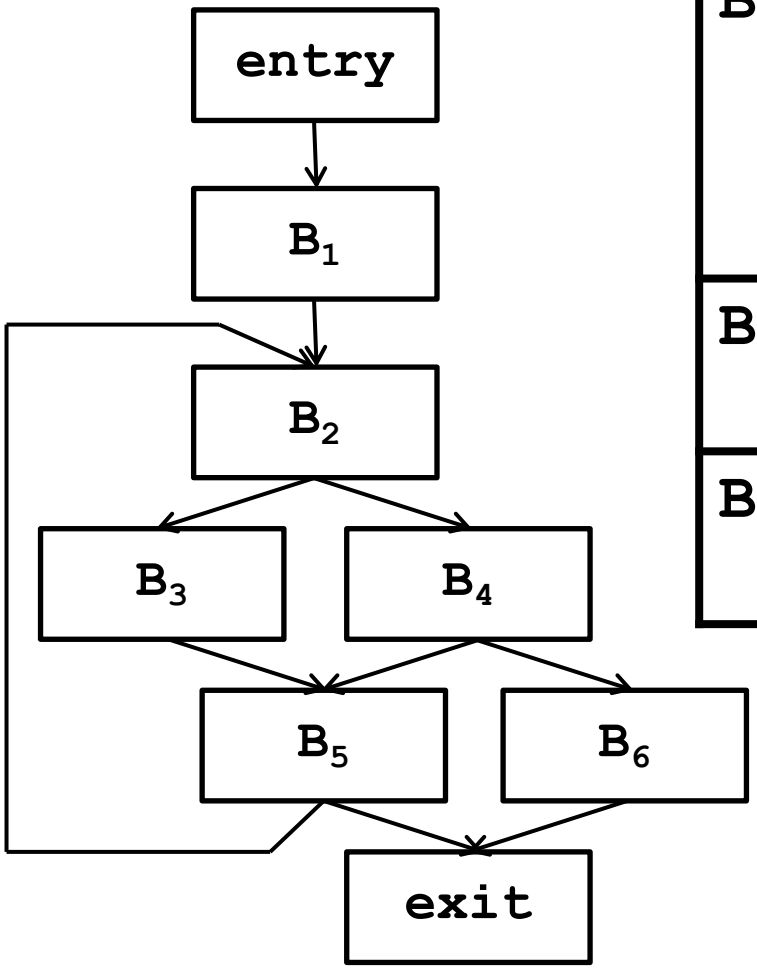


B_1 c ← +,a, b d ← c e ← *,d, d	B_2 f ← +,a, c g ← e a ← +,g, d if(a<c)goto
B_3 h ← +,g, 1	B_4 f ← -,d, g if(f>a)goto
B_5 b ← *,g, a if(f>h)goto	B_6 c ← 2

5.1. Простые оптимизации

5.1.3 Распространение копий

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



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

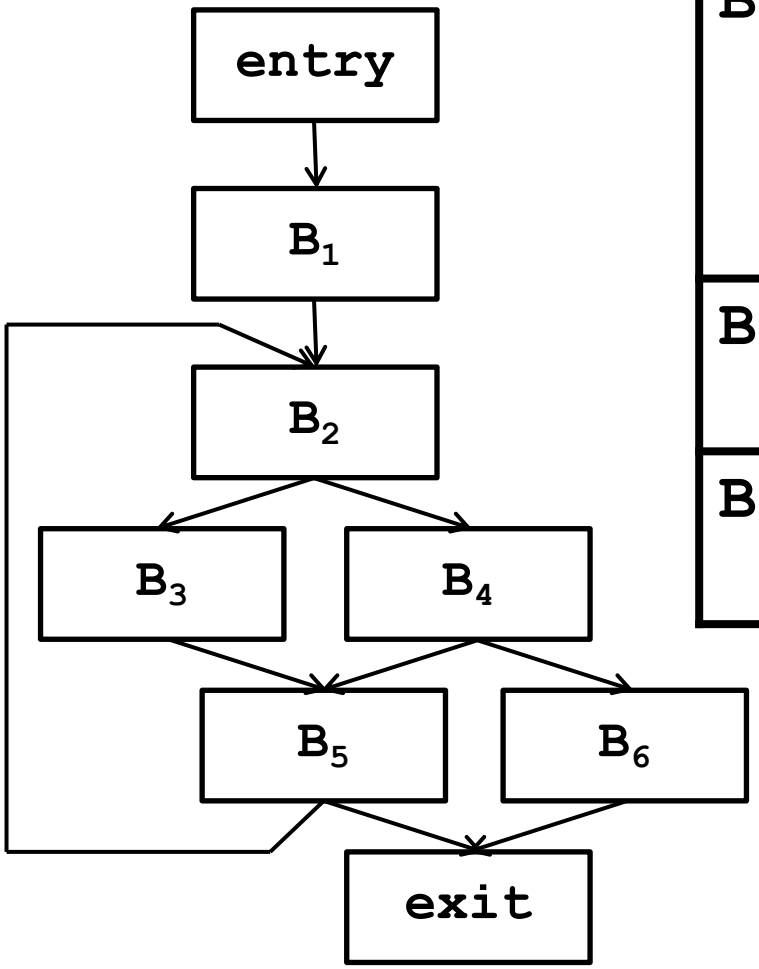
- **g ← e** (2-я строка блока B_2):

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

5.1. Простые оптимизации

5.1.3 Распространение копий

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



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

Следовательно множество U инструкций копирования равно

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

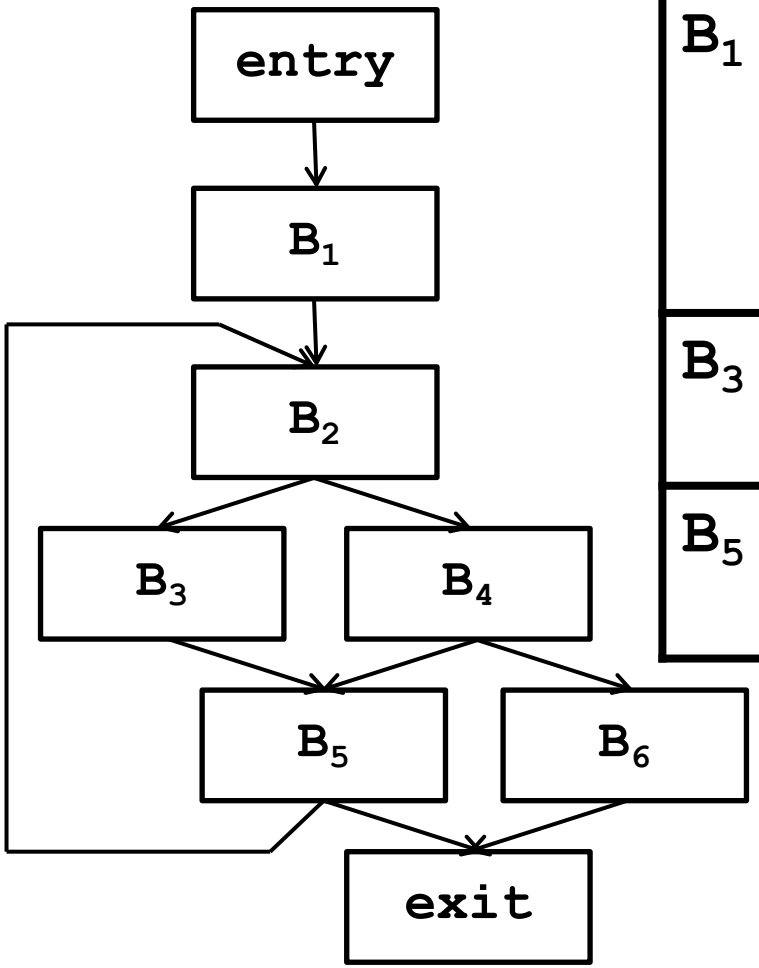
5.1. Простые оптимизации

5.1.3 Распространение копий

- Определим семейство передаточных функций, для этого необходимо ввести вспомогательные множества $copy$ и $kill$
- 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 $\langle x, y, b, p \rangle$), «убиваемых» в блоке b

5.1. Простые оптимизации

5.1.3 Распространение копий

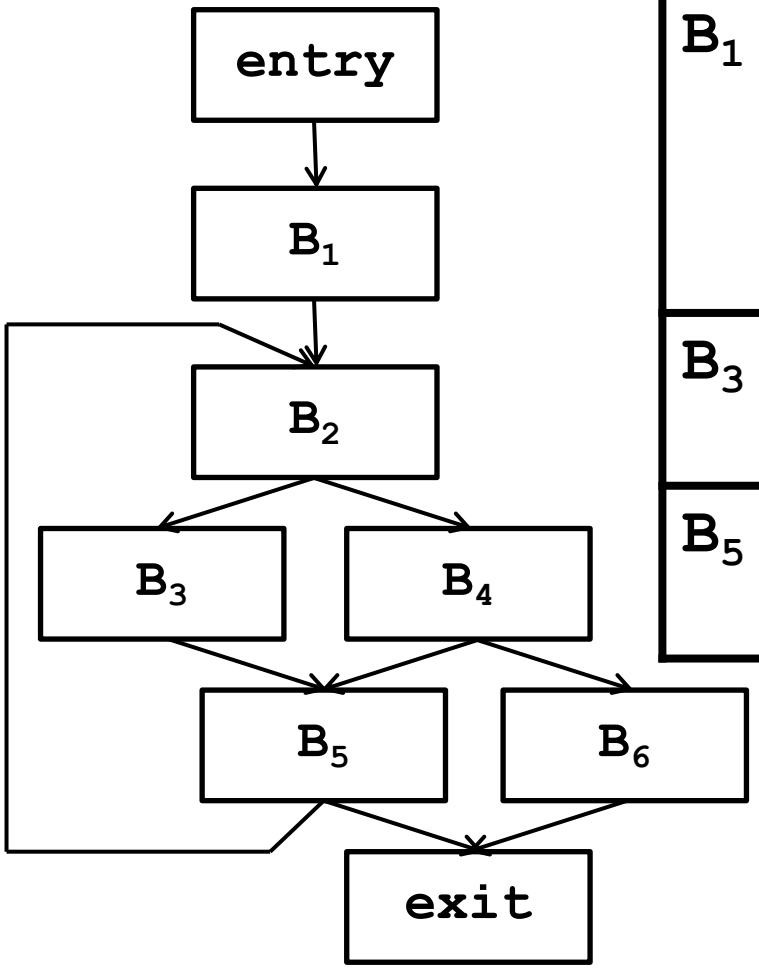


B_1 c ← +,a, b d ← c e ← *,d, d	B_2 f ← +,a, c g ← e a ← +,g, d if(a<c)goto
B_3 h ← +,g, 1	B_4 f ← -,d, g if(f>a)goto
B_5 b ← *,g, a if(f>h)goto	B_6 c ← 2

Определим множества $copy(b)$ для блоков рассматриваемого примера.

5.1. Простые оптимизации

5.1.3 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

Определим множества $copy(b)$ для блоков рассматриваемого примера.

$$copy(B_1) = \{\langle d, c, B_1, 2\rangle\}$$
$$copy(B_2) = \{\langle g, e, B_2, 2\rangle\}$$

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

$$copy(B_3) = \emptyset, copy(B_4) = \emptyset, copy(B_5) = \emptyset, copy(B_6) = \emptyset,$$

5.1. Простые оптимизации

5.1.3 Распространение копий

Определим множества $copy(B)$:
Интерес представляют только такие операции копирования $x \leftarrow y$, для которых ни x, ни y, не переопределяются после копирования до конца этого базового блока B.

Т.е. $copy(B)$ состоит из множества всех таких четверок $\langle x, y, B, p \rangle$, что в базовом блоке B в строке с номером p имеется операция копирования $x \leftarrow y$, причем x и y не переопределяются после копирования в строке p до конца этого базового блока B.

$$copy(B) = \{ \langle x, y, B, p \rangle \}:$$
$$\exists instr_1 = (x \leftarrow y) \in B,$$
$$\nexists instr_2 \succ instr_1 : instr_2 = (dest \leftarrow expr) \in B,$$
$$dest = x \vee dest = y$$

5.1. Простые оптимизации

5.1.3 Распространение копий

Определим множества $kill(B)$:
Операции присваивания в блоке *B*, в левой части которых встречается переменная либо из левой, либо из правой части операции копирования, «убивают» такое копирование.

Если есть присваивание (копия) **a** ← **b**, то

- ниже инструкции

  ```
  a = 1
  ```

  копирование **a** ← **b** уже нельзя использовать, т.к. вместо **a** уже нельзя подставить **b**, которое было выше: теперь **a** нельзя заменить на **b**, а только на 1.
- Аналогично, ниже инструкции

  ```
  b = 2
  ```

  копирование **a** ← **b** распространять нельзя, т.к. присваивание было выполнено при другом значении **b**.

Таким образом, присваивание, в левой части которого встречается либо левая, либо правая часть операции копирования, «убивает» такое копирование.

5.1. Простые оптимизации

5.1.3 Распространение копий

Определим множества $kill(B)$:
Операции присваивания в блоке B, в левой части которых встречается переменная либо из левой, либо из правой части операции копирования, «убивают» такое копирование.

Таким образом, $kill(B)$ состоит из таких операций копирования (*встречающихся в каком-либо множестве* $copy(b)$*, без учета достижимости в потоке управления*) $x \leftarrow y$, у которых либо левая, либо правая часть операции копирования встречается в блоке *B* в качестве левой части какого-либо присваивания:

$$kill(B_{kill}) = \{ \langle x, y, b_{copy}, p \rangle \} \subseteq \bigcup_{b \in CFG} copy(b):$$

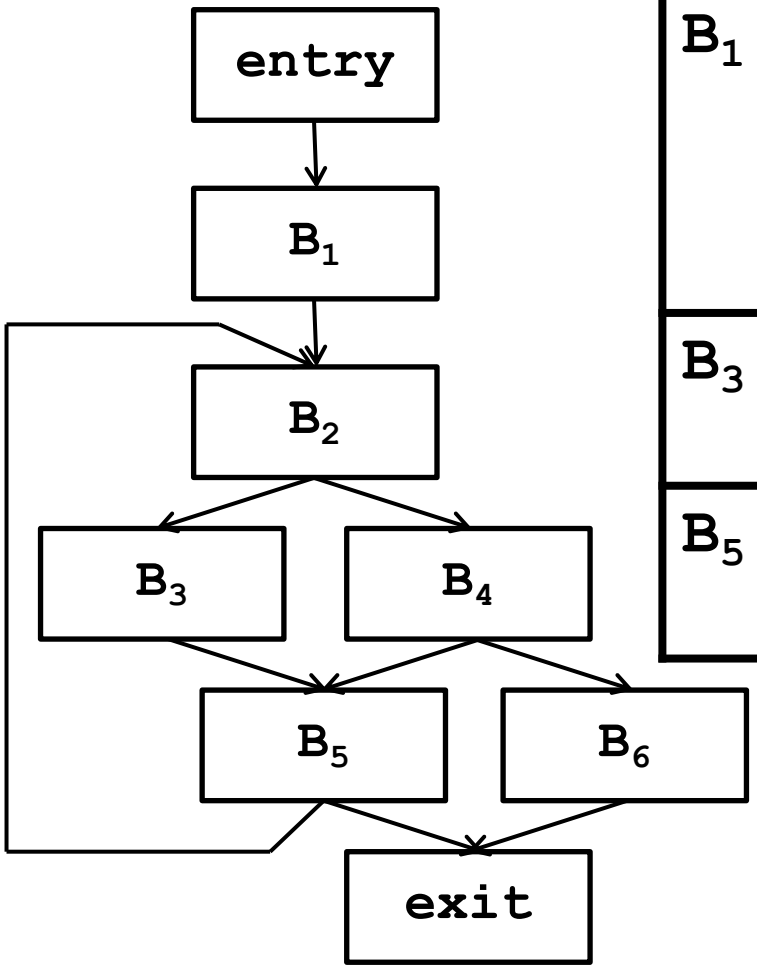
$$\exists instr = (dest \leftarrow expr) \in B_{kill}: dest = x \vee dest = y$$

Замечание. Согласно такому формальному определению, любое копирование также «убивает» и само себя, т.е. $\forall B$: $copy(B) \subseteq kill(B)$. Мы ранее сталкивались с таким для задач ***RD*** и ***LV***, и это не влияло на результат. Можно также внести дополнительное ограничение:

$$\langle x, y, b_{copy}, p \rangle \notin copy(B_{kill}), b_{copy} = B_{kill}$$

5.1. Простые оптимизации

5.1.3 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

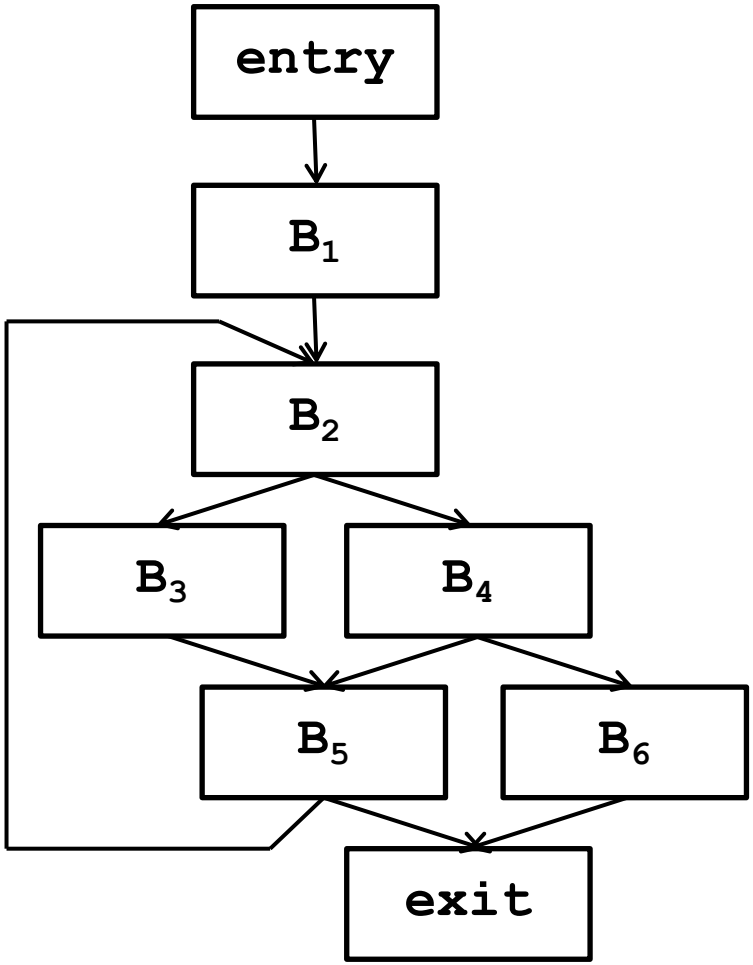
Определим множества $kill(b)$

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

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

5.1. Простые оптимизации

5.1.3 Распространение копий



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

b	copy(b)	kill(b)
entry	$\emptyset$	$\emptyset$
B_1	$\{\langle d, c, B_1, 2\rangle\}$	$\{\langle g, e, B_2, 2\rangle\}$
B_2	$\{\langle g, e, B_2, 2\rangle\}$	$\emptyset$
B_3	$\emptyset$	$\emptyset$
B_4	$\emptyset$	$\emptyset$
B_5	$\emptyset$	$\emptyset$
B_6	$\emptyset$	$\{\langle d, c, B_1, 2\rangle\}$
exit	$\emptyset$	$\emptyset$

5.1. Простые оптимизации

5.1.3 Распространение копий

- Система уравнений составляется по аналогии с системой уравнений для достигающих определений:
  - в передаточной функции сначала из множества инструкций копирования удаляются инструкции «убитые» в блоке b, потом в него добавляются инструкции копирования блока b

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

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

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

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

5.1. Простые оптимизации

5.1.3 Распространение копий

- Алгоритм локального распространения копий внутри базового блока
  - За один проход по командам базового блока заменяет использование копии переменной самой переменной
  - Информация о доступных локальных командах копирования хранится в хеш-таблице ACP – Available Copy instructions
  - Хеш-таблица ACP ассоциирует переменную с командой копирования, описываемой четверкой $\langle x, y, b, p \rangle$
    - x – переменная копия
    - y – копируемая переменная
    - b – базовый блок, содержащий команду копирования
    - p – номер команды копирования в базовом блоке
  - Алгоритм использует вспомогательные функции
    - `copyValue(operand, ACP)` – используя ACP возвращает переменную по её копии (т.е. правую часть по левой части копирования)
    - `removeACP(ACP, var)` – удаляет команды копирования из таблицы доступных команд, для которых выполняется условие: $\langle x, y, b, p \rangle$: `x == var || y == var`
    - `isAssignment(i)` – возвращает предикат, определяющий является ли команда `i` командой присваивания
    - `isVariable(operand)` – возвращает предикат, определяющий является ли операнд переменной

5.1. Простые оптимизации

5.1.3 Распространение копий

- Алгоритм локального распространения копий внутри базового блока

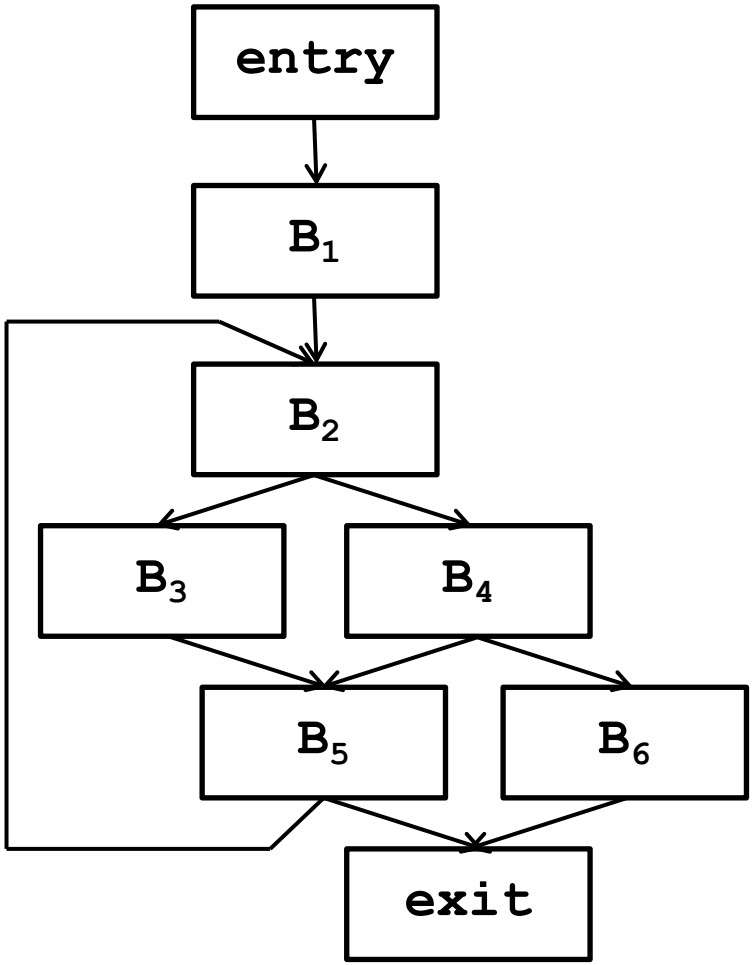
```
localCopyPropagation(b: BasicBlock,
                     ACP: Available Copy Instructions) {
   foreach (i: Instruction in b) {
      match (i) {
      // во всех операциях dest ← OP, arg1, arg2 для каждой копии
      // x ← y из ACP заменить все вхождения x на y в аргументах
      |binexp arg1, arg2| => i.rhs[0] = copyValue(arg1, ACP)
                             i.rhs[1] = copyValue(arg2, ACP)
      |unexp arg1|        => i.rhs[0] = copyValue(arg1, ACP)
      }

      if (hasLhs(i)) {           // если операция i имеет вид x = … , то
         removeACP(ACP, i.lhs)   // удалить все копирования, у которых
                                 // x встречается либо в левой, либо в
                                 // правой части
      }
      // если операция имеет вид x = y, то добавить найденное копирование в ACP
      if (isAssignment(i) && isVariable(i.rhs[0])) {
         ACP += {i.lhs, i.rhs[0]}
      }
   }
}
```

5.1. Простые оптимизации

5.1.4 Распространение копий

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

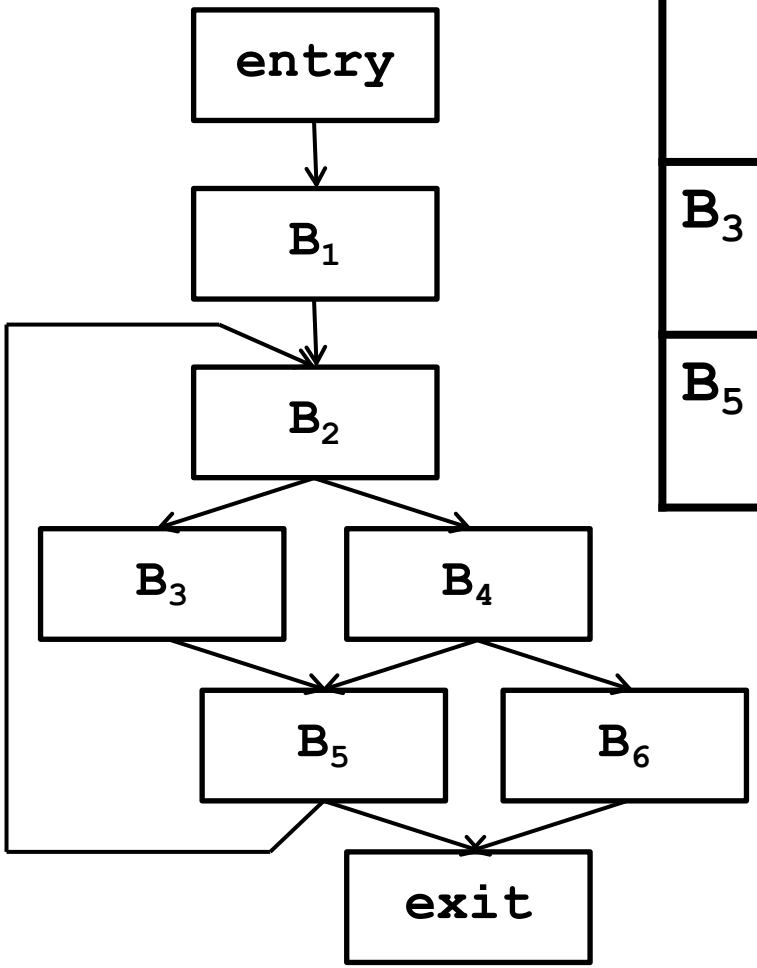


b	$\mathbf{In_{CP}(b)}$
Entry	$\emptyset$
B_1	$\emptyset$
B_2	$\{\langle d, c, B_1, 2\rangle\}$
B_3	$\{\langle d, c, B_1, 2\rangle, \langle g, e, B_2, 2\rangle\}$
B_4	$\{\langle d, c, B_1, 2\rangle, \langle g, e, B_2, 2\rangle\}$
B_5	$\{\langle d, c, B_1, 2\rangle, \langle g, e, B_2, 2\rangle\}$
B_6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

5.1. Простые оптимизации

5.1.4 Распространение копий

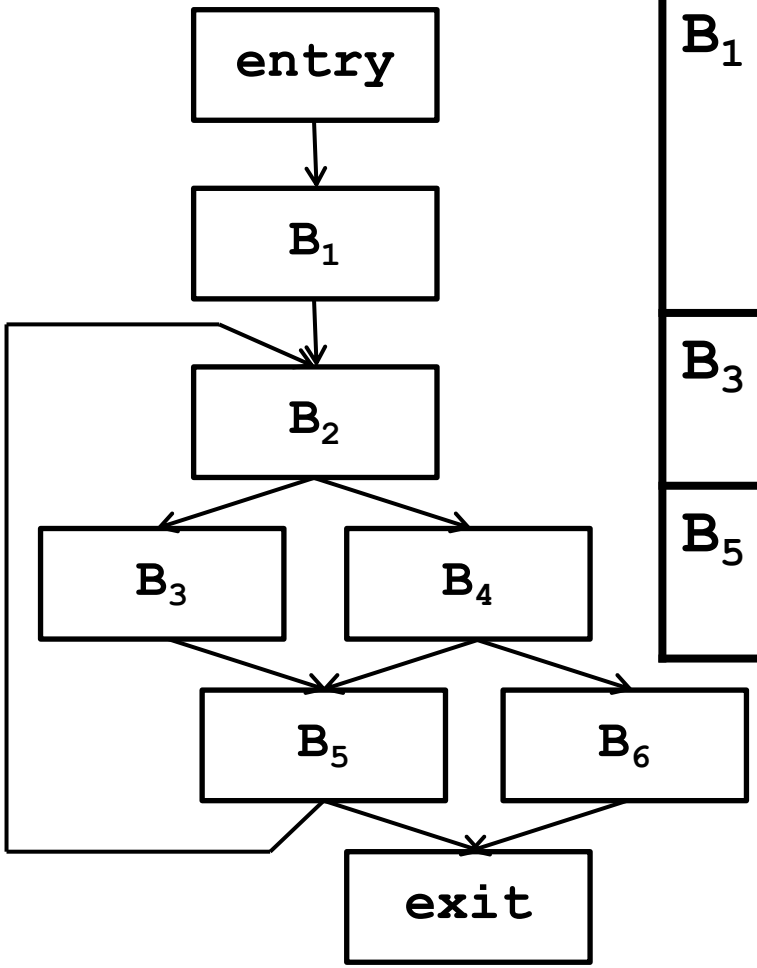
- Исходная программа:



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

5.1. Простые оптимизации

5.1.3 Результат



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

5.3. Исключение бесполезного кода

5.3.1 Постановка задачи

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

5.3. Исключение бесполезного кода

5.3.2 Алгоритм *Mark & Sweep*.

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

5.3. Исключение бесполезного кода

5.3.3 Поток управления: переходы и ветвления.

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

5.3. Исключение бесполезного кода

5.3.3 Поток управления: переходы и ветвления.

- При удалении бесполезных инструкций необходимо учитывать *поток управления*.
- ✓ – полезные инструкции (по определению).

```
int foo(int x, int y, int z) {
  int t;
  if (x > 42) {
    printf("Hello\n");
  }
  t = y + z;
  return t;
}
```

5.3. Исключение бесполезного кода

5.3.3 Поток управления: переходы и ветвления.

- При удалении бесполезных инструкций необходимо учитывать *поток управления*.
- ✓ – полезные инструкции (по определению).
- ✓ – инструкции и переменные, участвующие в вычислении полезных переменных, также полезны (т.е. инструкции, от которых зависят по данным полезные, также полезны)

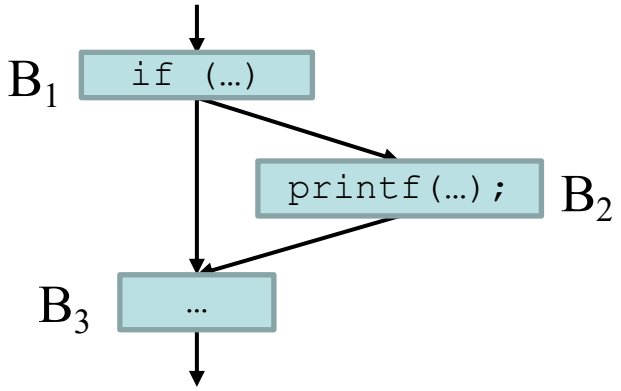
```
int foo(int x, int y, int z) {
  int t;
  if (x > 42) {
    printf("Hello\n");
  }
  t = y + z;
  return t;
}
```

5.3. Исключение бесполезного кода

5.3.3 Поток управления: переходы и ветвления.

- При удалении бесполезных инструкций необходимо учитывать *поток управления*.
- ✓ – полезные инструкции (по определению).
- ✓ – инструкции и переменные, участвующие в вычислении полезных переменных, также полезны (т.е. инструкции, от которых зависят по данным полезные, также полезны)
- ✓ – инструкции, от которых зависят по управлению полезные инструкции, также полезны

```
int foo(int x, int y, int z) {
  int t;
  if (x > 42) {
    printf("Hello\n");
  }
  t = y + z;
  return t;
}
```



$RDF(B_2) = B_1$

5.3. Исключение бесполезного кода

5.3.4 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 $n \in N$ это обычная граница доминирования в обратном графе G^R.

5.3. Исключение бесполезного кода

5.3.5 Зависимость по управлению.

◊ По определению, вершина m ГПУ *зависит по управлению* от вершины n тогда, и только тогда, когда:

- существует непустой путь T от n до m, такой что $\forall k \in T - \{n\}: m = Postdom(k)$, т.е. если выполнение программы пошло по пути T, то, чтобы достичь $exit$, оно обязательно пройдет через m.
- m не является строгим постдоминатором n: у n может быть несколько выходов, так что помимо T возможны и другие пути, проходящие через n, но потом ведущие не в m, а в другие вершины.

◊ Обратная граница доминирования позволяет определять границы зависимостей по управлению.

5.3. Исключение бесполезного кода

5.3.5.1 Эквивалентность по управлению

◊ **Определение**. Два базовых блока B_i и B_j *эквивалентны по управлению*, если B_i выполняется тогда, и только тогда, когда выполняется B_j .

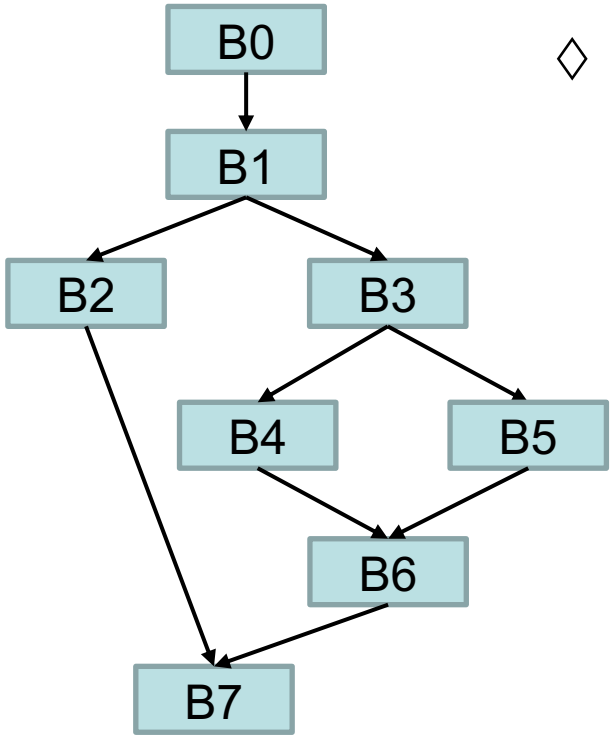
◊ **Утверждение**. Если выполняются соотношения:

$B_i = Dom(B_j)$ и $B_j = Postdom(B_i)$

то базовые блоки B_i и B_j эквивалентны по управлению

5.3. Исключение бесполезного кода

5.3.5.2 Зависимость и эквивалентность по управлению. Примеры.

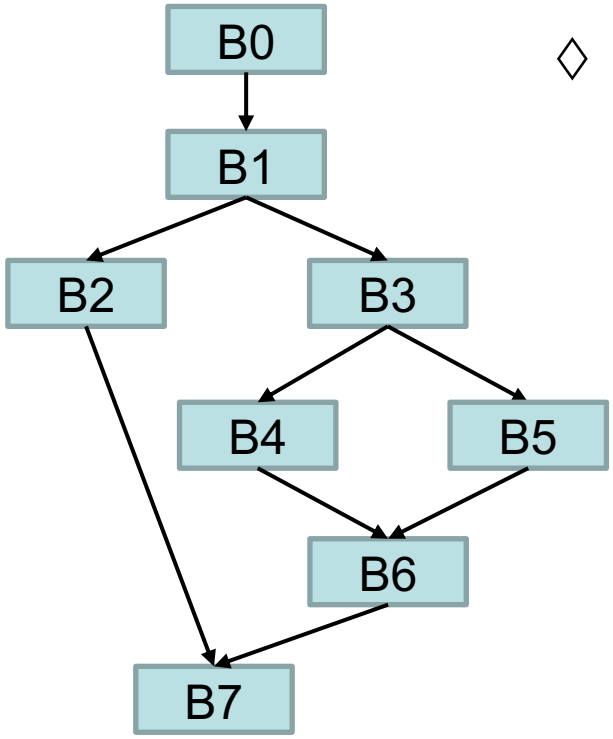


- **B3** (а также B6) зависит по управлению от **B1**, т. к. в B1 принимается решение о выборе пути дальнейшего выполнения (и попадания в B3 и B6)
- С точки зрения определения зависимости по управлению:
  - существует путь (B1, B3], в котором B3 является постдоминатором для всех узлов в пути, кроме первого (B1), и B3 не является постдоминатором B1 =>
    B3 зависит по управлению от B1
  - аналогично,
    B5 зависит по управлению от B3

- нет пути, исходящего из B1, в котором B4 или B5 являются постдоминаторами для узлов этого пути => **B4 и B5 не зависят по управлению от B1, хотя и зависят от B3, а B3 зависит от B1** => зависимость по управлению не транзитивна

5.3. Исключение бесполезного кода

5.3.5.2 Зависимость и эквивалентность по управлению. Примеры.



- ◊ B3 (а также **B6**) зависит по управлению от **B1**, т. к. в B1 принимается решение о выборе пути дальнейшего выполнения (и попадания в B6)
- ◊ С точки зрения определения зависимости по управлению:
  - ◊ существуют пути (вообще говоря, все пути, начинающихся в B1, проходящие через B3, и заканчивающихся в B6):
    (B1, B3, …, b_i, …, B6] – в них
    B6 будет постдоминатором для всех промежуточных узлов b_i (не считая B1)
  - ◊ B6 не является постдоминатором B1
- ◊ **B1 входит в обратную границу доминирования B6 (и B3)**

◊ B6 не зависит по управлению от B3 – эти блоки **эквивалентны по управлению**: если выполнился код в B3, то обязательно выполнится и B6, и наоборот: если выполнился B6, то прежде был выполнен и B3.

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark*.

- На первом проходе (*Mark*) в каждом базовом блоке n:
  - В *Worklist* добавляются все «точно полезные» инструкции программы (`return`, `printf`, запись в глобальные переменные), эти инструкции помечаются, с остальных пометка снимается
- выбирается очередная инструкция из *Worklist*
  - для этой инструкции
    - помечаются ветви, по которым «приходят» операнды инструкции (операнды полезны, так как используются в полезной инструкции)
    - посещаются все блоки $b \in RDF(n)$ и помечается каждая ветвь, ведущая к этим блокам.
    - Каждая помеченная ветвь помещается в *Worklist*.
- Проход завершается, когда *Worklist* становится пустым.

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark* (псевдокод)

```
Mark( )
WorkList ← ∅;
for each инструкции i  //(x ← op, y, z, или ветвление)
        убрать пометку у i
        if (i полезная) пометить i
        WorkList ← WorkList ∪ {i}
while (WorkList ≠ ∅)
        remove i from WorkList
        for each u ∈ uses_in_rhs(i)
                if (def(u) не помечена)
                    пометить def(u)
                    WorkList ← WorkList ∪ {def(u)}
        for each block b ∈ RDF(block(i))
                пусть j - инструкция условного перехода,
                          которой заканчивается блок b
                if (j не помечена)
                    пометить j
                    WorkList ← WorkList ∪ {j}
```

Для обычных инструкций `i` вида $x \leftarrow op, y, z$:

uses_in_rhs(i) = { y, z }

Если i – условный переход:

uses_in_rhs(i) – все переменные, используемые в условии

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark* (псевдокод)

```
Mark( )
WorkList ← ∅;
for each инструкции i  //(x ← op, y, z, или ветвление)
        убрать пометку у i
        if (i полезная) пометить i
        WorkList ← WorkList ∪ {i}
while (WorkList ≠ ∅)
        remove i from WorkList
        for each u ∈ uses_in_rhs(i)
                if (def(u) не помечена)
                        пометить def(u)
                        WorkList ← WorkList ∪ {def(u)}
                пометить j
                WorkList ← WorkList ∪ {j}
```

Для обычных инструкций `i` вида $x \leftarrow op, y, z$:

uses_in_rhs(i) = { y, z }

Если i – условный переход:

uses_in_rhs(i) – все переменные, используемые в условии

Замечание:
Для наглядности, в псевдокоде данного алгоритма приводятся **def(u)**, но на самом деле здесь подразумеваются ***все*** определения **u**.
В этом смысле алгоритм упрощается, если его выполнять на внутреннем представлении в SSA-форме (см. следующую лекцию)

5.3. Исключение бесполезного кода

5.3.6. Проход *Mark* (псевдокод)

```
Mark( )
WorkList ← ∅;
for each инструкции i  //(x ← op, y, z, или ветвление)
      убрать пометку у i
      if (i полезная) пометить i
      WorkList ← WorkList ∪ {i}
while (WorkList ≠ ∅)
```

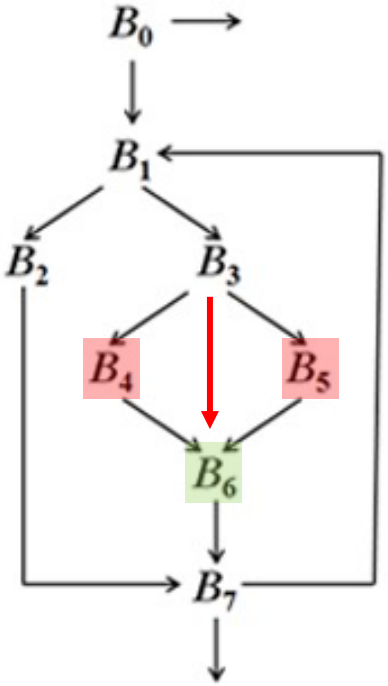
Почему нужно помечать ветви:
Если **i** вычисляет что-то полезное, то и ветвь **j**, которая ведет к **block(i)**, также полезная (а также переменные, участвующие в вычислении условия для перехода по **j**). Здесь мы помечаем ветви блоков, от которых **block(i)** зависит по управлению.

```
      for each block b ∈ RDF(block(i))
            пусть j - инструкция условного перехода,
                      которой заканчивается блок b
            if (j не помечена)
                пометить j
                WorkList ← WorkList ∪ {j}
```

5.3. Исключение бесполезного кода

5.3.6 Проход *Sweep*

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

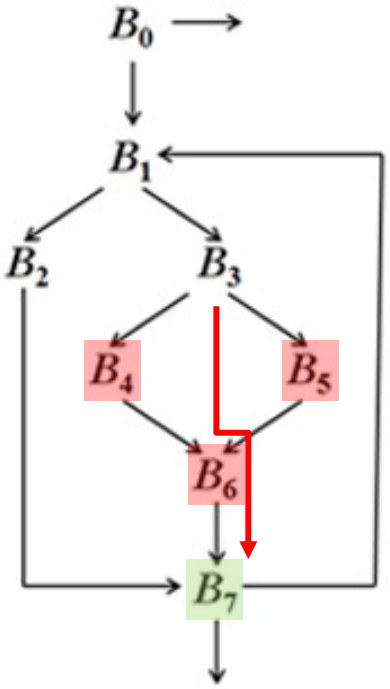


- На втором проходе (*Sweep*) в каждый блок, с которого начинается непомеченная ветвь, помещается безусловный переход на его помеченный постдоминатор.
  Это правильно, так как если ветвь не помечена, потомки блока вплоть до его непосредственного постдоминатора, не могут содержать полезных инструкций,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

5.3. Исключение бесполезного кода

5.3.6 Проход *Sweep*

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



- Сказанное справедливо и для непосредственного непомеченного постдоминатора ветвления.
  Чтобы найти ближайший помеченный постдоминатор, можно двигаться вверх по дереву постдоминаторов, пока не найдется помеченный блок.
  Поиск обязательно закончится,
  так как по определению блок *Exit* помечен.

5.4. Исключение недостижимого кода

5.4.1 Постановка задачи

- Иногда ГПУ содержит *недостижимый код*. Компилятор должен найти недостижимые базовые блоки и исключить их.
- Две причины недостижимости блока:
  - в ГПУ отсутствует путь, ведущий к базовому блоку;
  - путь, достигающий блока, может быть невыполнимым для любых возможных входных данных
    (например, **`if(argc || 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.

5.4. Исключение недостижимого кода

5.4.2 Анализ достижимости

- Проход $Mark$ сначала помечает каждый блок b как «недостижимый», потом он начинает обход ГПУ с $entry$ и помечает как «достижимый» каждый блок, которого он может достичь.

- Если все ветвления и переходы определяются однозначно, то все блоки, помеченные как недостижимые, действительно недостижимы и могут быть удалены на проходе $Sweep$.

- В случае неоднозначных условий ветвления, компилятор должен сохранить любой блок, достижимый ветвлением или переходом.

5.5. Оптимизация потока управления

5.5.1. Постановка задачи

- Некоторые оптимизации могут иметь побочный эффект, изменяющий форму ГПУ, добавляя в него бесполезные блоки и дуги.
  Если компилятор содержит такие оптимизации, он должен также содержать проход, упрощающий ГПУ,
  исключая бесполезный поток управления.

- Функция *Clean* обрабатывает непосредственно ГПУ оптимизируемой процедуры, упрощая его.

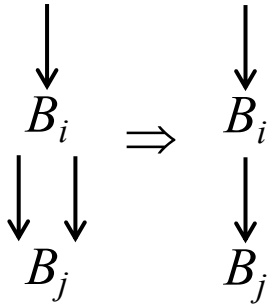
5.5. Оптимизация потока управления

5.5.2. Основные преобразования. Преобразование 1.

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

1. *Свернуть избыточную ветвь:*

   Если последние инструкции блока B_i реализуют ветвление, и обе ветви выполняют условный переход на один и тот же блок B_j, то ветвление заменяется безусловным переходом на блок B_j.



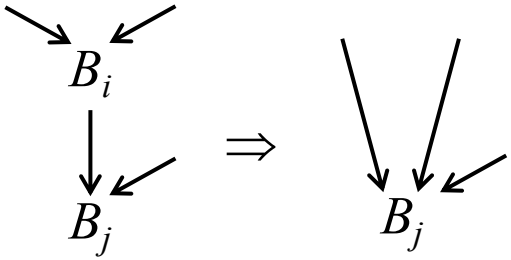
Такая ситуация может возникнуть в результате других оптимизаций (**Например**, у B_i могло быть два последователя, каждый из которых заканчивался переходом на B_j. Если другие оптимизации убрали из этих блоков все инструкции, то второе из основных преобразований могло породить левый граф, показанный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

5.5. Оптимизация потока управления

5.5.2. Основные преобразования. Преобразование 2.

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

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

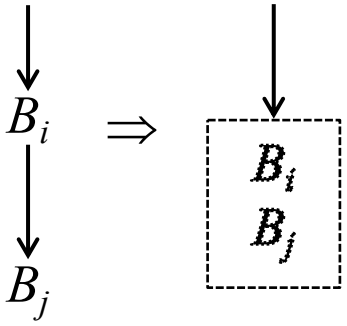


Так как у B_i всего один последователь, B_j, преобразование перенаправляет дуги, входящие в B_i, к B_j и исключает B_i из $Pred(B_j)$, что упрощает ГПУ и ускоряет выполнение.

5.5. Оптимизация потока управления

5.5.2. Основные преобразования. Преобразование 3.

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



5.5. Оптимизация потока управления

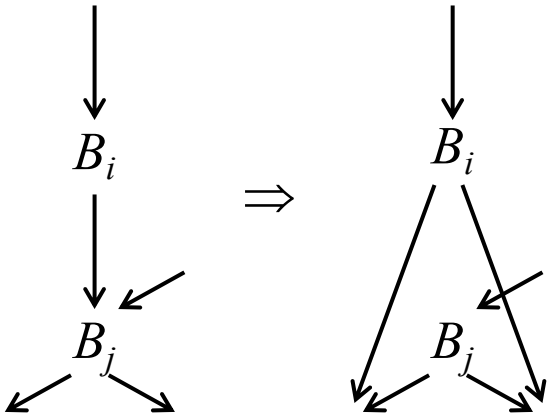
5.5.2. Основные преобразования. Преобразование 4.

4. *Подъём ветвлений*. В ситуации, когда блок B_i, который оканчивается переходом в пустой блок B_j, а блок B_j оканчивается ветвлением, переход в конце блока заменяется на копию ветвления из блока B_j.

Такое преобразование поднимает ветвление из B_j в B_i.

Ситуация может возникнуть, если другие оптимизации удалят все операции из B_j, оставив только ветвление.

К ГПУ добавится ребро. Объединить B_i и B_j нельзя, так как у B_j есть еще предшественники (если бы их не было, B_i и B_j уже были бы объединены Преобразованием 3).



5.5. Оптимизация потока управления

5.5.3. Функция `Clean()`

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

OnePass()
for each block 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 */
```

5.5. Оптимизация потока управления

5.5.3. Функция `Clean()`

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

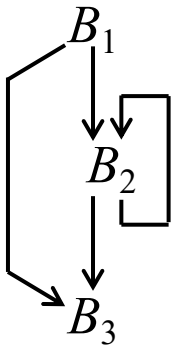
- В начале каждой итерации выполняется новая нумерация блоков, так как после каждого применения четверки преобразований ГПУ может сильно измениться.

- Функция **`Clean()`** не может самостоятельно удалить пустой цикл (цикл с пустым телом). Это показывает следующий пример.

5.5. Оптимизация потока управления

5.5.4. Пример

- Рассмотрим процедуру, ГПУ которой изображен на рисунке.
  Пусть блок B_2 пуст.
  Ни одно из преобразований функции $Clean()$
  не может удалить пустой блок B_2 :
  - ветвление в конце B_2 не избыточно;
  - B_2 не завершается безусловным 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.



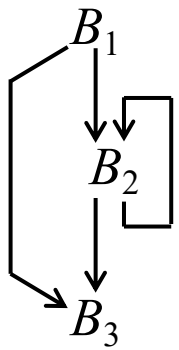
Исходный ГПУ

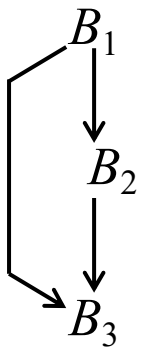
5.5. Оптимизация потока управления

5.5.4. Пример

- Однако если исходный ГПУ предварительно обработать с помощью $Mark\ \&\ Sweep$, рассматриваемый пустой цикл удастся удалить.

- Блоки B_1 и B_3 содержат полезные инструкции, а блок B_2 пуст (не считая операции ветвления), 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.



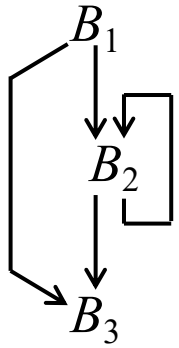


Исходный ГПУ ГПУ после $Mark\ \&\ Sweep$

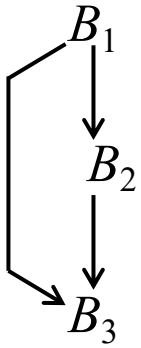
5.5. Оптимизация потока управления

5.5.4. Пример

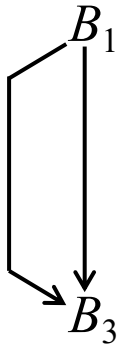
- *Clean* загоняет B_2 в B_1, в результате получается ГПУ, изображенный на третьем рисунке слева.
- Теперь ветвление в конце B_1 становится избыточным, и *Clean* заменяет его переходом, в результате получается ГПУ, изображенный на четвертом рисунке слева.
- Наконец, если окажется, что B_1 – единственный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



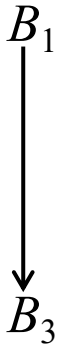
Исходный ГПУ



ГПУ после *Mark & Sweep*



ГПУ после удаления B_2.



ГПУ после сворачивания избыточной ветви.