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.

◊ **Пример**. Рассмотрим фрагмент программы:

```
Const a = 1.7320;
Const b = 1.4140;
c = a + b;
```

Компилятор заменит его на:

```
Const a = 1.7320;
Const b = 1.4140;
Const c = 3.1460;
```

◊ Оптимизация состоит в том, что ***часть вычислений выполняется во время компиляции и убирается из программы.***

5.1. Простые оптимизации

5.1.1 Сворачивание констант

- **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 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 в соответствующей точке программы выдать во время ее выполнения *сообщение об ошибке*, либо *предупреждение*.

- Наибольшее количество проблем, естественно, возникает, если сворачивается константа одного из плавающих типов.

5.1. Простые оптимизации

5.1.2 Алгебраические упрощения и перегруппировка

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

$$i + 0 = 0 + i = i - 0 = i$$
$$0 - i = -i$$
$$i * 1 = 1 * i = i / 1 = i$$
$$i * 0 = 0 * i = 0$$
$$-(-i) = i$$
$$i + (-j) = i - j$$

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

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

5.1. Простые оптимизации

5.1.3 Распространение копий

◊ Пусть в оптимизируемой процедуре есть инструкция копирования

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

Распространение копий означает замену всех последующих вхождений переменной **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.

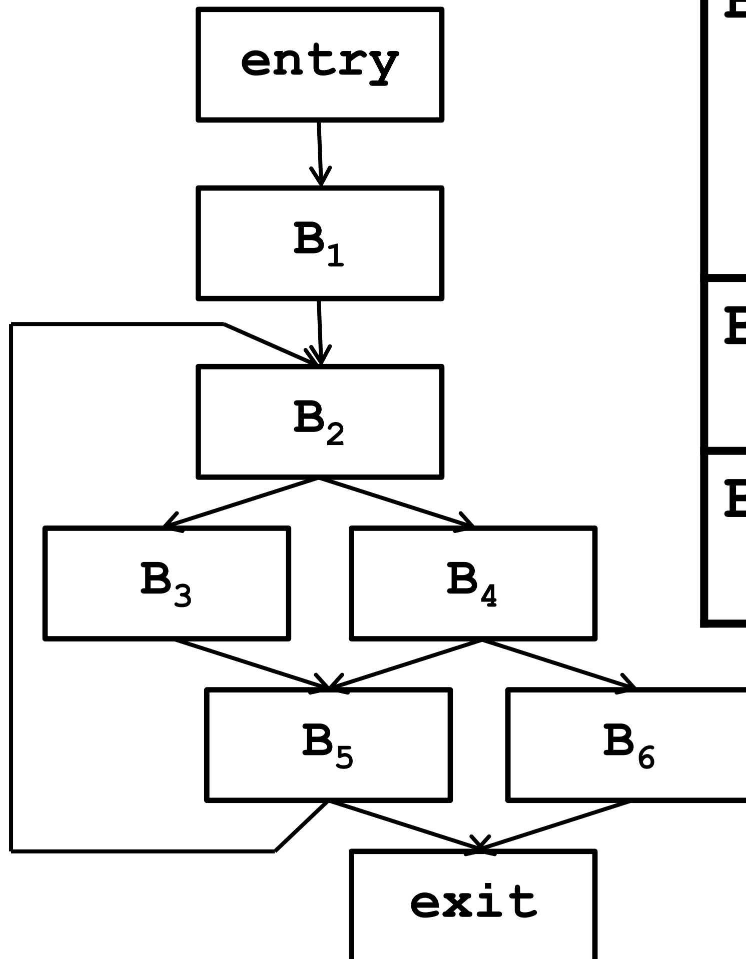
Множество всех таких четверок обозначим через U.

Множество U содержит все инструкции копирования анализируемой процедуры.

5.1. Простые оптимизации

5.1.3 Распространение копий

◊ В процедуре, ГПУ которой представлен на рисунке две команды копирования

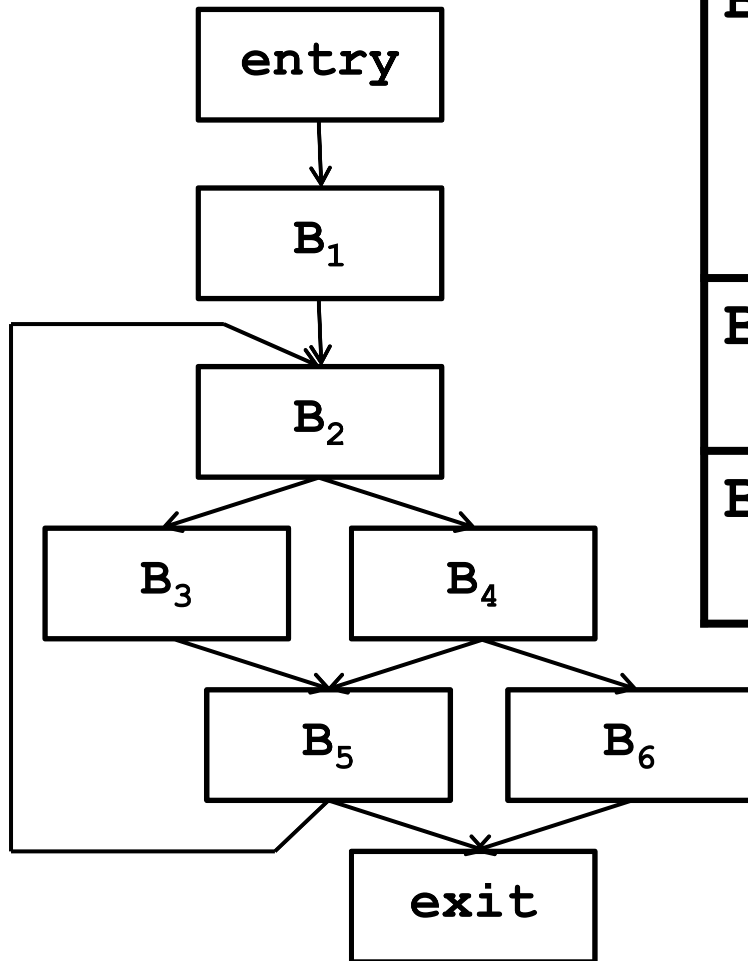


B_1 c ← +,a, b d ← c e ← *,d, d	B_2 f ← +,a, c g ← e a ← +,g, d if(a<c)goto
B_3 h ← +,g, 1	B_4 f ← -,d, g if(f>a)goto
B_5 b ← *,g, a if(f>h)goto	B_6 c ← 2

5.1. Простые оптимизации

5.1.3 Распространение копий

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



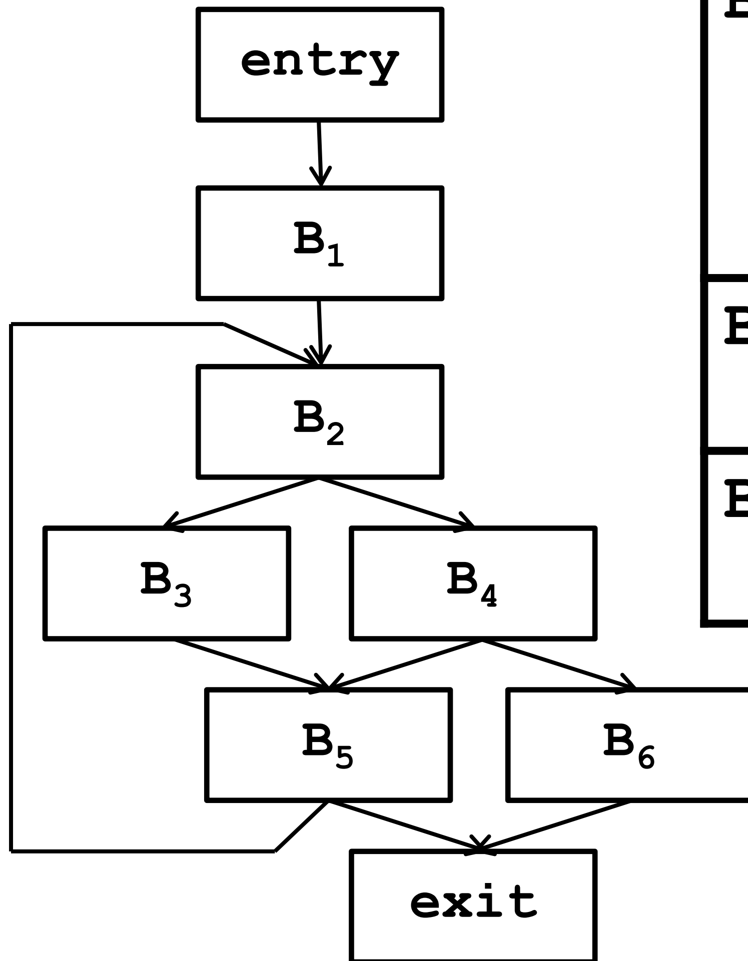
B_1 c ← +,a, b d ← c e ← *,d, d	B_2 f ← +,a, c g ← e a ← +,g, d if(a<c)goto
B_3 h ← +,g, 1	B_4 f ← -,d, g if(f>a)goto
B_5 b ← *,g, a if(f>h)goto	B_6 c ← 2

- **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 инструкции 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

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

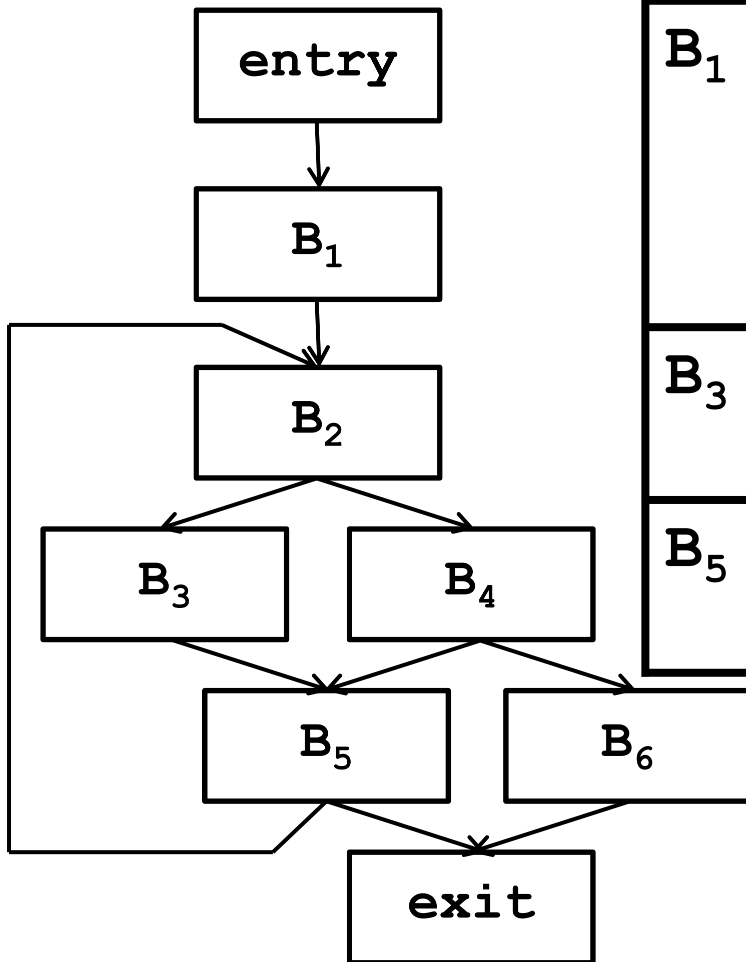
5.1. Простые оптимизации

5.1.3 Распространение копий

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

5.1. Простые оптимизации

5.1.3 Распространение копий

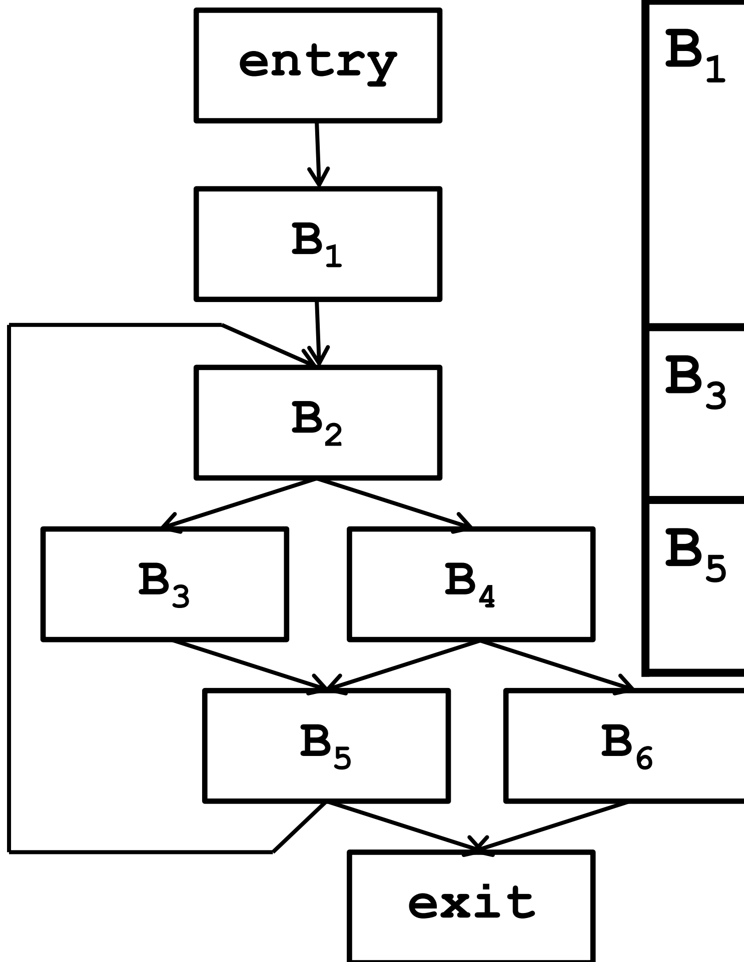


B_1 c ← +,a, b d ← c e ← *,c, c	B_2 f ← +,a, c g ← e a ← +,e, c if(a<c)goto
B_3 h ← +,e, 1	B_4 f ← -,c, e if(f>a)goto
B_5 b ← *,e, a if(f>h)goto	B_6 c ← 2

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 ← *,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

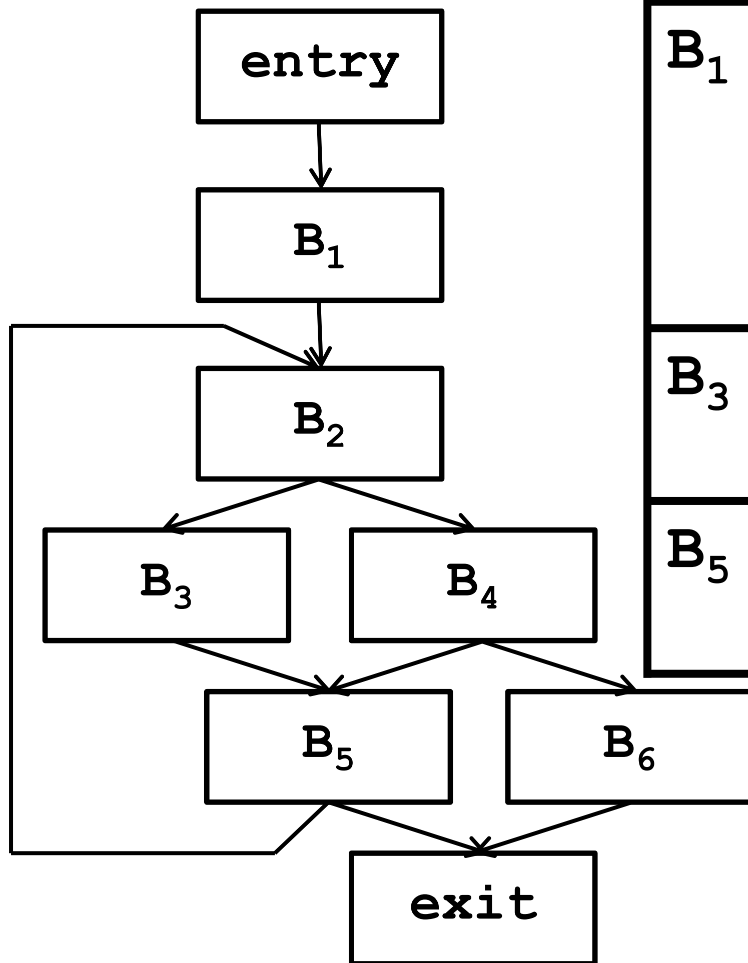
Определим множества $copy(b)$ для блоков рассматриваемого примера.

$copy(\mathbf{B_1}) = \{\langle \mathbf{d}, \mathbf{c}, \mathbf{B_1}, \mathbf{2} \rangle\}$

$copy(\mathbf{B_2}) = \{\langle \mathbf{g}, \mathbf{e}, \mathbf{B_2}, \mathbf{2} \rangle\}$

5.1. Простые оптимизации

5.1.3 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

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

$copy(\mathbf{B_1}) = \{\langle \mathbf{d}, \mathbf{c}, \mathbf{B_1}, \mathbf{2} \rangle\}$

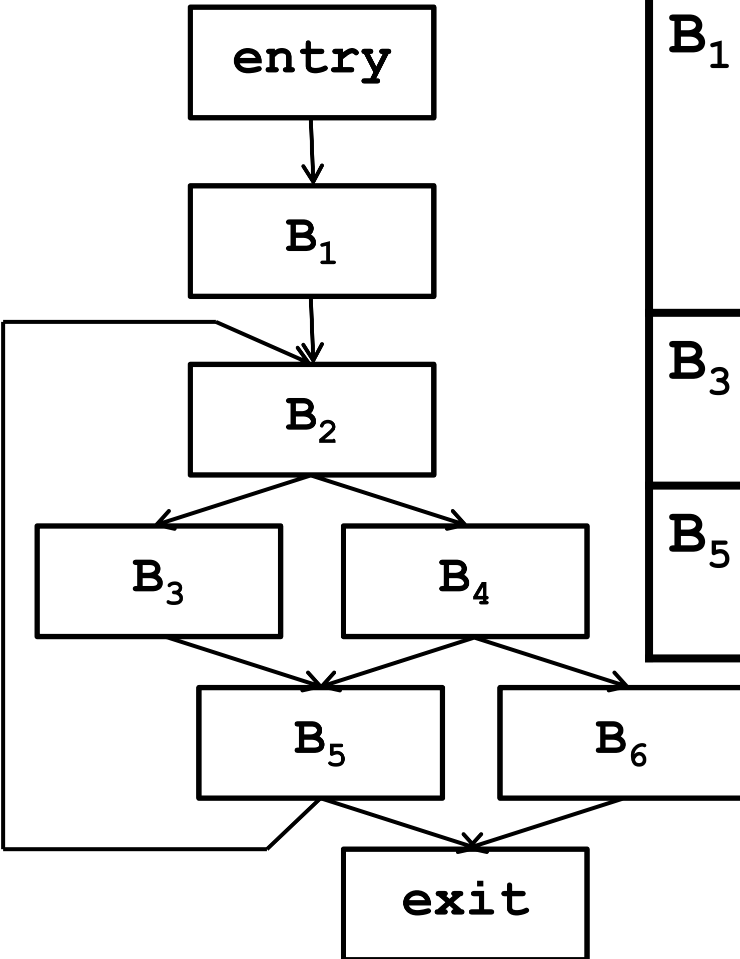
$copy(\mathbf{B_2}) = \{\langle \mathbf{g}, \mathbf{e}, \mathbf{B_2}, \mathbf{2} \rangle\}$

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

$copy(\mathbf{B_3})=\varnothing$, $copy(\mathbf{B_4})=\varnothing$, $copy(\mathbf{B_5})=\varnothing$, $copy(\mathbf{B_6})=\varnothing$

5.1. Простые оптимизации

5.1.3 Распространение копий

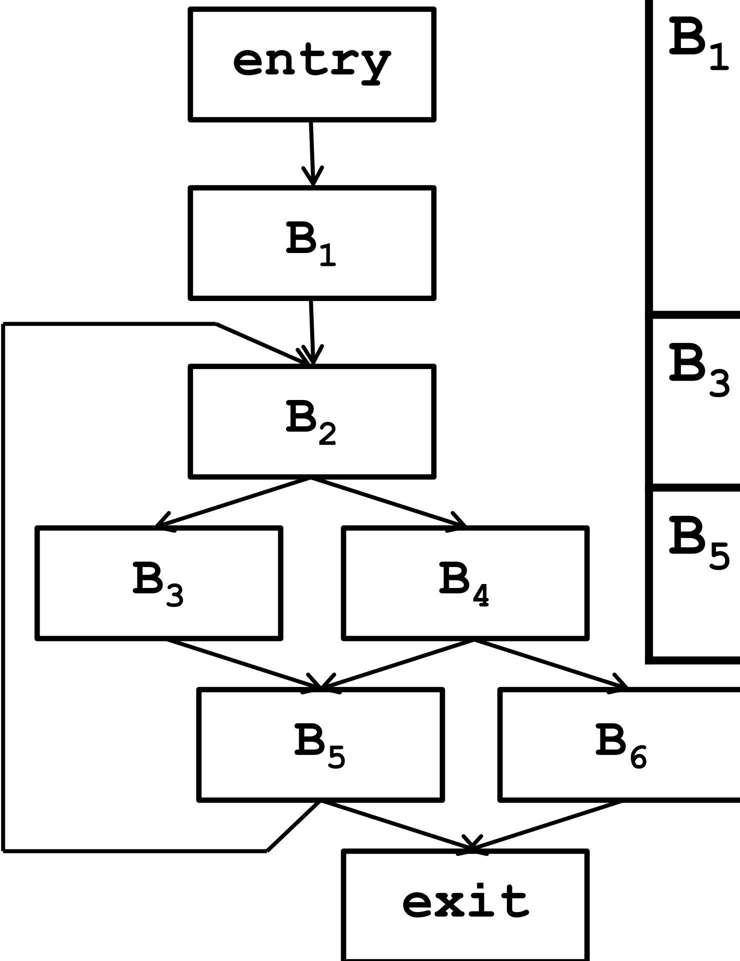


B_1 c ← +,a, b d ← c e ← *,c, c	B_2 f ← +,a, c g ← e a ← +,e, c if(a<c)goto
B_3 h ← +,e, 1	B_4 f ← -,c, e if(f>a)goto
B_5 b ← *,e, a if(f>h)goto	B_6 c ← 2

Определим множества $kill(b)$

5.1. Простые оптимизации

5.1.3 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

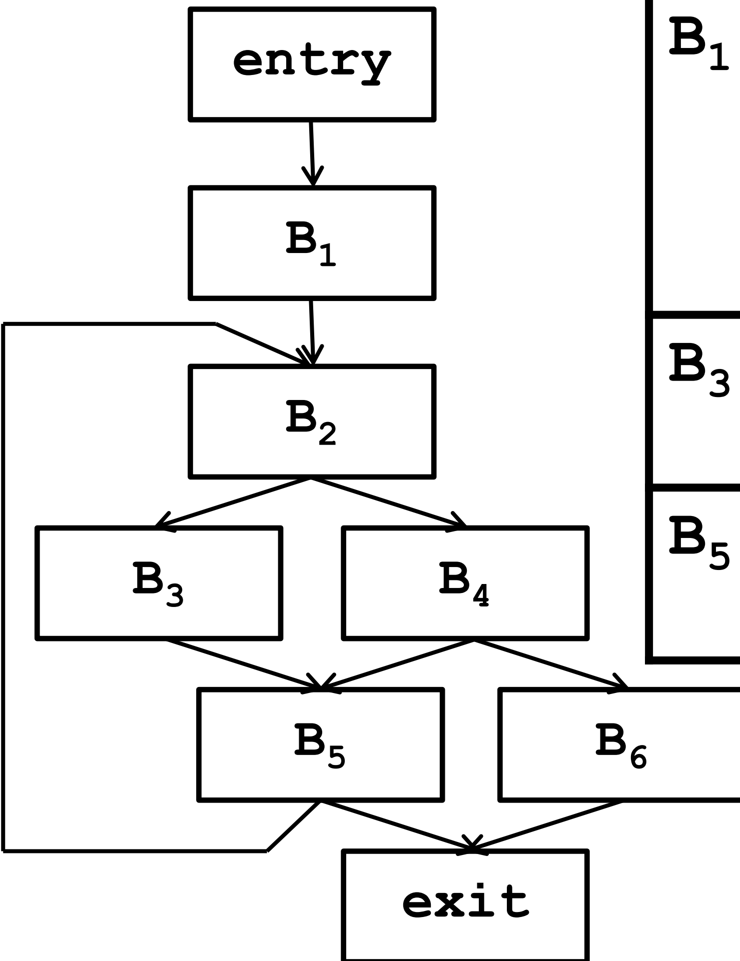
Определим множества $kill(b)$

$\langle$**g,e,B**$_2$**,2**$\rangle$ убивается в блоке **B**$_1$, так как в 3-ей строке этого блока переопределяется **e**. Следовательно $kill(\mathbf{B}_1) = \{\langle \mathbf{g,e,B}_2\mathbf{,2}\rangle\}$

$\langle$**d,c,B**$_1$**,2**$\rangle$ убивается в блоке **B**$_6$, так как единственная инструкция этого блока переопределяет **c**. Следовательно $kill(\mathbf{B}_6) = \{\langle \mathbf{d,c,B}_1\mathbf{,2}\rangle\}$

5.1. Простые оптимизации

5.1.3 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

Определим множества $kill(b)$

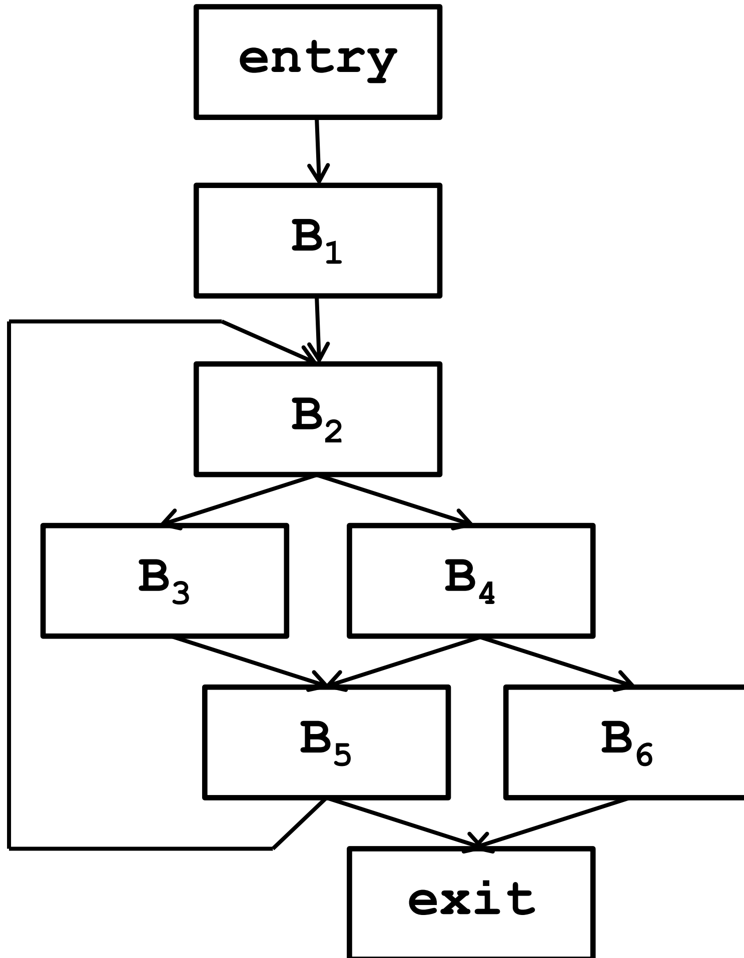
$\langle$**g,e,B$_2$,2**$\rangle$ убивается в блоке **B$_1$**, так как в 3-ей строке этого блока переопределяется **e**. Следовательно $kill(\mathbf{B_1}) = \{\langle \mathbf{g,e,B_2,2} \rangle\}$

$\langle$**d,c,B$_1$,2**$\rangle$ убивается в блоке **B$_6$**, так как единственная инструкция этого блока переопределяет **c**. Следовательно $kill(\mathbf{B_6}) = \{\langle \mathbf{d,c,B_1,2} \rangle\}$

Множества $kill(b)$ остальных блоков пустые.

5.1. Простые оптимизации

5.1.3 Распространение копий



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

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

5.2. Исключение бесполезного кода

5.2.1 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

5.2. Исключение бесполезного кода

5.2.2 Алгоритм *Mark & Sweep*.

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

5.2. Исключение бесполезного кода

5.2.3 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.

5.2. Исключение бесполезного кода

5.2.4 Постдоминаторы (напоминание)

◊ *Обратным графом* ориентированного графа $G = \langle N, E\rangle$ называется ориентированный граф $G^R = \langle N, E^R\rangle$, у которого направления всех ребер противоположны.

◊ В ГПУ вершина p является *постдоминатором* вершины n ($p = Postdom\ (n)$) , если каждый путь из вершины n в вершину $exit$ проходит через вершину p.

Постдоминаторы ГПУ – это доминаторы его *обратного графа*.

◊ *Обратная граница доминирования* ($RDF(n)$) вершины $n \in G$ это обычная граница доминирования в обратном графе G^R.

5.2. Исключение бесполезного кода

5.2.5 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.**

5.2. Исключение бесполезного кода

5.2.6. Проход $Mark$.

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

Каждая помеченная ветвь помещается в $Worklist$.

Проход завершается, когда $Worklist$ становится пустым.

5.2. Исключение бесполезного кода

5.2.6. Проход *Mark*.

◊ Базовый блок, содержащий хотя бы одну помеченную инструкцию, помечается для ускорения анализа.

◊ Ветвь состоит из одного или более блоков. Она считается помеченной, если помечен хотя бы один из ее блоков.

Каждая помеченная ветвь помещается в *Worklist*.

Проход завершается, когда *Worklist* становится пустым.

5.2. Исключение бесполезного кода

5.2.6. 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}
```

5.2. Исключение бесполезного кода

5.2.6 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.

5.2. Исключение бесполезного кода

5.2.6 Проход *Sweep*

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

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

5.3. Исключение недостижимого кода

5.3.1 Постановка задачи

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

5.3. Исключение недостижимого кода

5.3.2 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.

5.4. Оптимизация потока управления

5.4.1. 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.

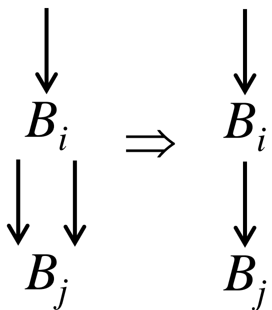
5.4. Оптимизация потока управления

5.4.2. 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).



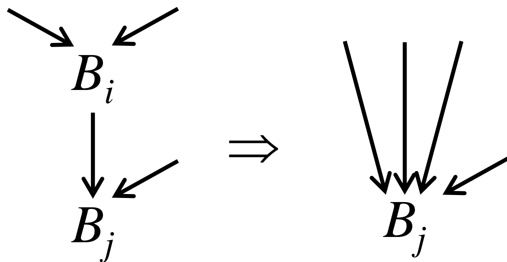
$$B_i \to B_j \;(\text{два перехода}) \quad \Rightarrow \quad B_i \to B_j$$

5.4. Оптимизация потока управления

5.4.2. 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.

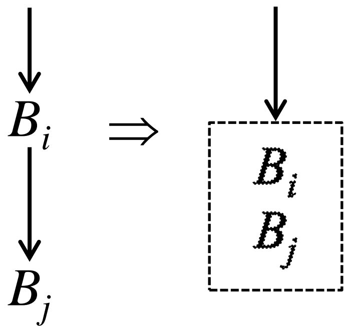


Так как у B_i всего один последователь, B_j, преобразование перенаправляет дуги, входящие в B_i, к B_j и исключает B_i из $Pred(B_j)$, что упрощает ГПУ и ускоряет выполнение.

5.4. Оптимизация потока управления

5.4.2. Основные преобразования. Преобразование 3.

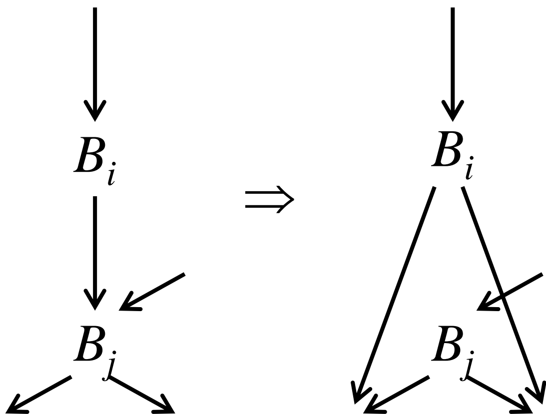
◊ 3. *Объединение блоков:* Если имеется блок B_i, который оканчивается переходом на B_j, у которого всего один предшественник, B_i, он может объединить эти блоки как показано на рисунке внизу, что позволяет исключить переход из B_i в B_j.



5.4. Оптимизация потока управления

5.4.2. 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).



$\Rightarrow$

B_i

B_j

5.4. Оптимизация потока управления

5.4.3. 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 */
```

5.4. Оптимизация потока управления

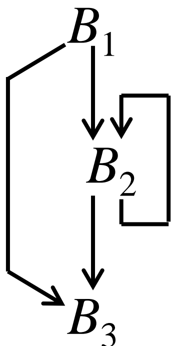
5.4.3. Функция `Clean()`

- 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.

5.4. Оптимизация потока управления

5.4.4. 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.



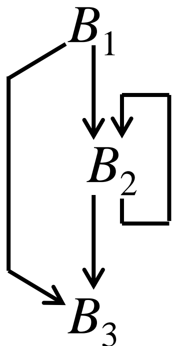
Исходный ГПУ

5.4. Оптимизация потока управления

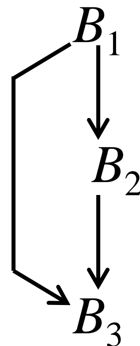
5.4.4. Пример

◊ 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

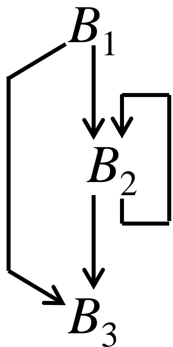


ГПУ после *Mark & Sweep*

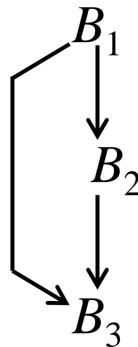
5.4. Оптимизация потока управления

5.4.4. Пример

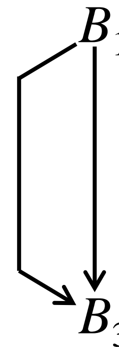
- ◊ $Clean$ загоняет B_2 в B_1, в результате получается ГПУ, изображенный на третьем рисунке слева.
- ◊ Теперь ветвление в конце B_1 становится избыточным, и $Clean$ заменяет его переходом, в результате получается ГПУ, изображенный на четвертом рисунке слева.
- ◊ Наконец, если окажется, что B_1 – единственный 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.