

6. Оптимизация циклов

6.1. Выделение естественных циклов

6.1.1 Классификация дуг ГПУ

- Дуги ГПУ, являющиеся дугами и его остовного дерева, называются *наступающими (остовными)*
- Дуги ГПУ, не являющиеся дугами его остовного дерева, но имеющие такое же направление, что и остовные (при любой нумерации DFST), называются *прямыми*.
- Дуги ГПУ, направленные противоположно остовным, называются *отступающими (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 (поперечные и обратно направленные дуги любят Фортранные программисты, но и у них они постепенно выходят из моды).

6.1. Выделение естественных циклов

6.1.1 Определение естественного цикла

- **Определение.** *Естественным циклом* называется цикл со следующими свойствами:
 - Цикл имеет единственный входной узел, называемый его *заголовком*,
 - Существует обратное ребро, ведущее в заголовок цикла
- **Определение.** *Естественный цикл обратного ребра* $\langle V_i, V_k \rangle$ составляют узел V_k (*заголовок цикла*) и все узлы ГПУ, из которых можно достичь узла V_i , не проходя через узел V_k . (эти узлы составляют *тело цикла*).

6.1. Выделение естественных циклов

6.1.1 Определение естественного цикла

- **Определение.**

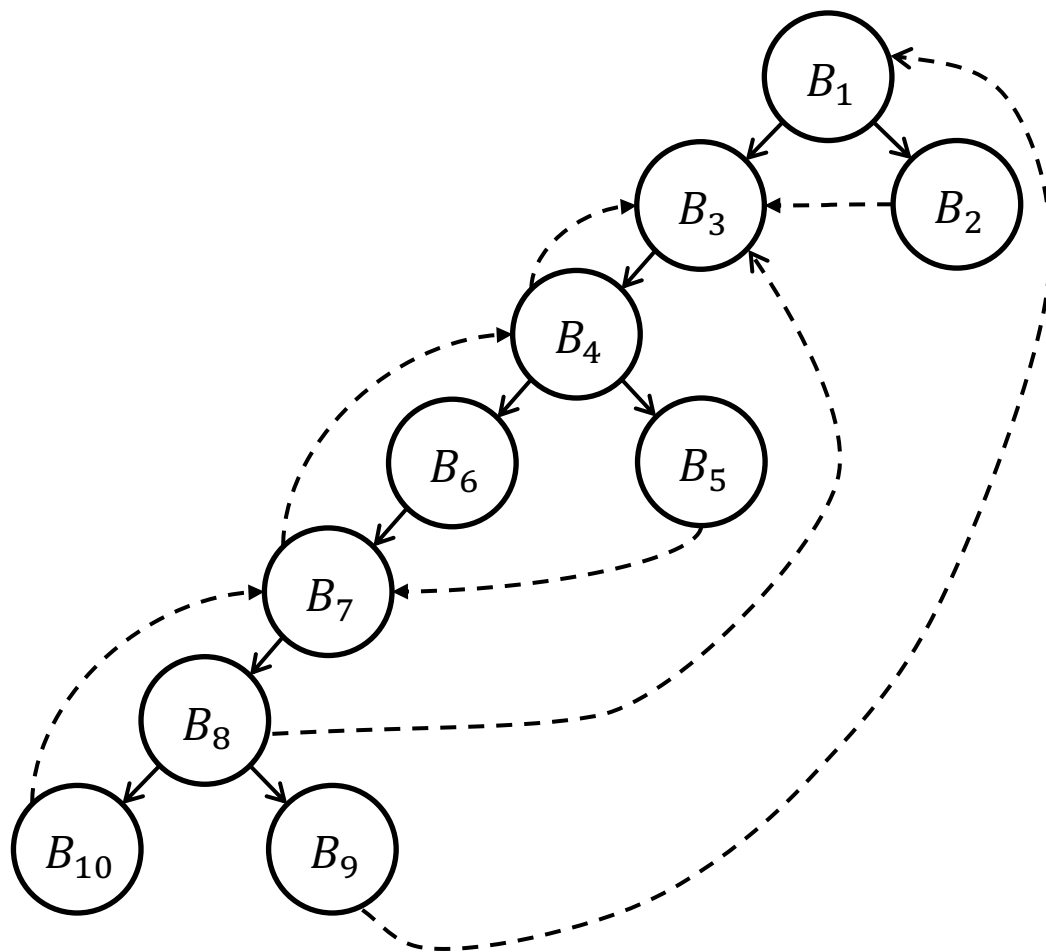
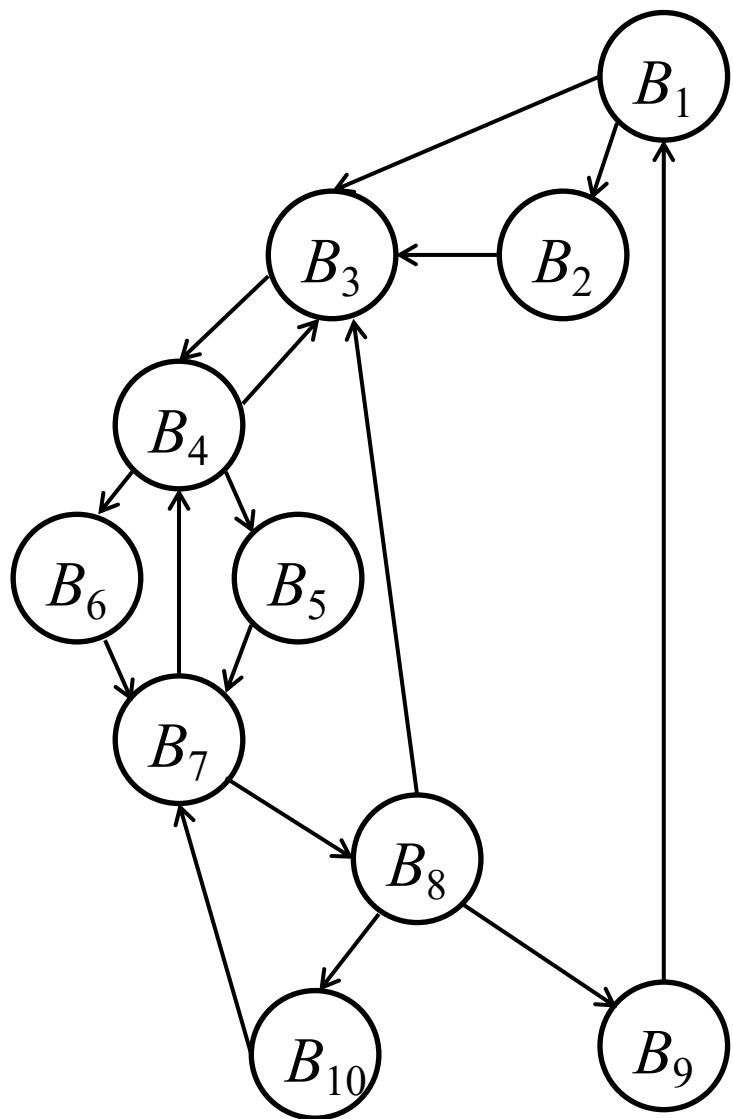
Граф потока называется приводимым (*reducible*), если все его отступающие ребра в любом DFST являются обратными

Если граф приводим, то все его DFST имеют одно и то же множество отступающих ребер, в точности совпадающее с множеством обратных ребер

Если граф не приводим, то все обратные ребра в любом DFST являются отступающими, но каждое DFST имеет дополнительные отступающие ребра, не являющиеся обратными

6.1. Выделение естественных циклов

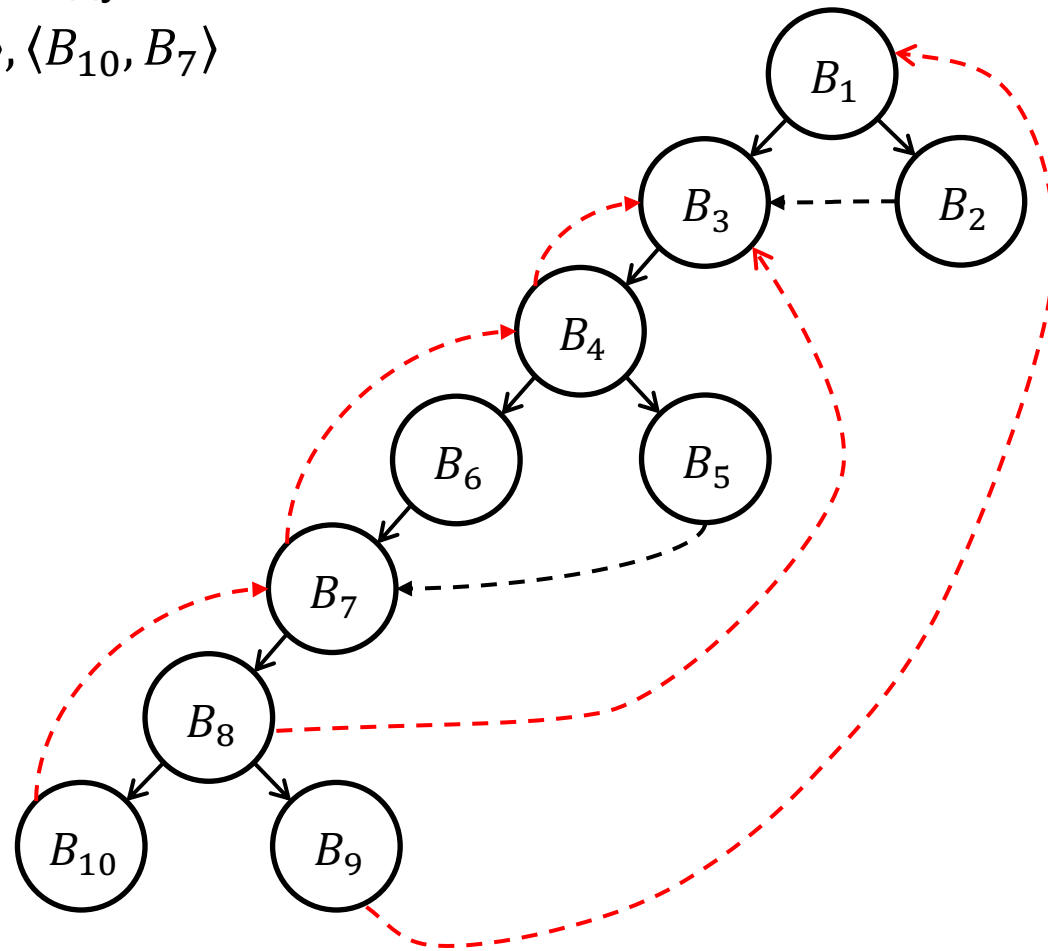
6.1.2 Естественные циклы. Пример.



6.1. Выделение естественных циклов

6.1.2 Естественные циклы. Пример.

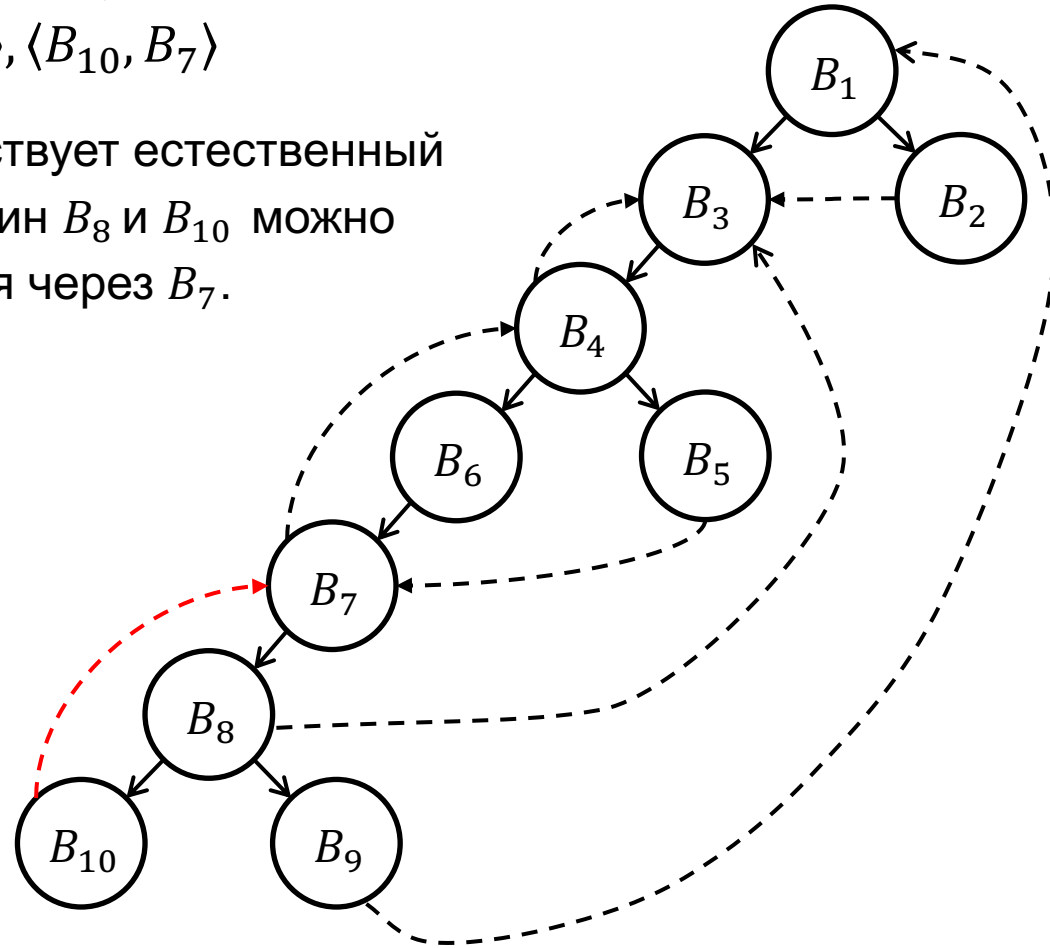
- На рисунке справа – пять обратных дуг:
 $\langle B_4, B_3 \rangle, \langle B_7, B_4 \rangle, \langle B_8, B_3 \rangle, \langle B_9, B_1 \rangle, \langle B_{10}, B_7 \rangle$



6.1. Выделение естественных циклов

6.1.2 Естественные циклы. Пример.

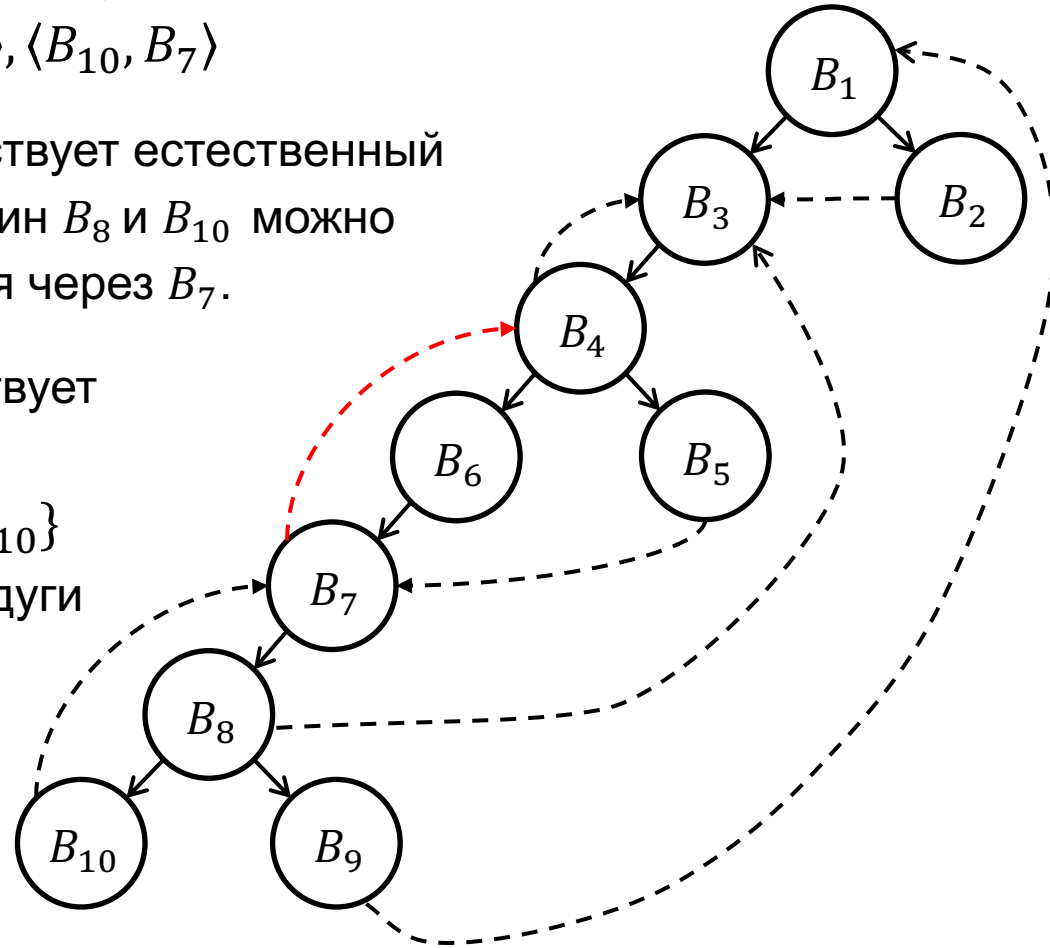
- На рисунке справа – пять обратных дуг:
 $\langle B_4, B_3 \rangle, \langle B_7, B_4 \rangle, \langle B_8, B_3 \rangle, \langle B_9, B_1 \rangle, \langle B_{10}, B_7 \rangle$
- Обратной дуге $\langle B_{10}, B_7 \rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и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



6.1. Выделение естественных циклов

6.1.2 Естественные циклы. Пример.

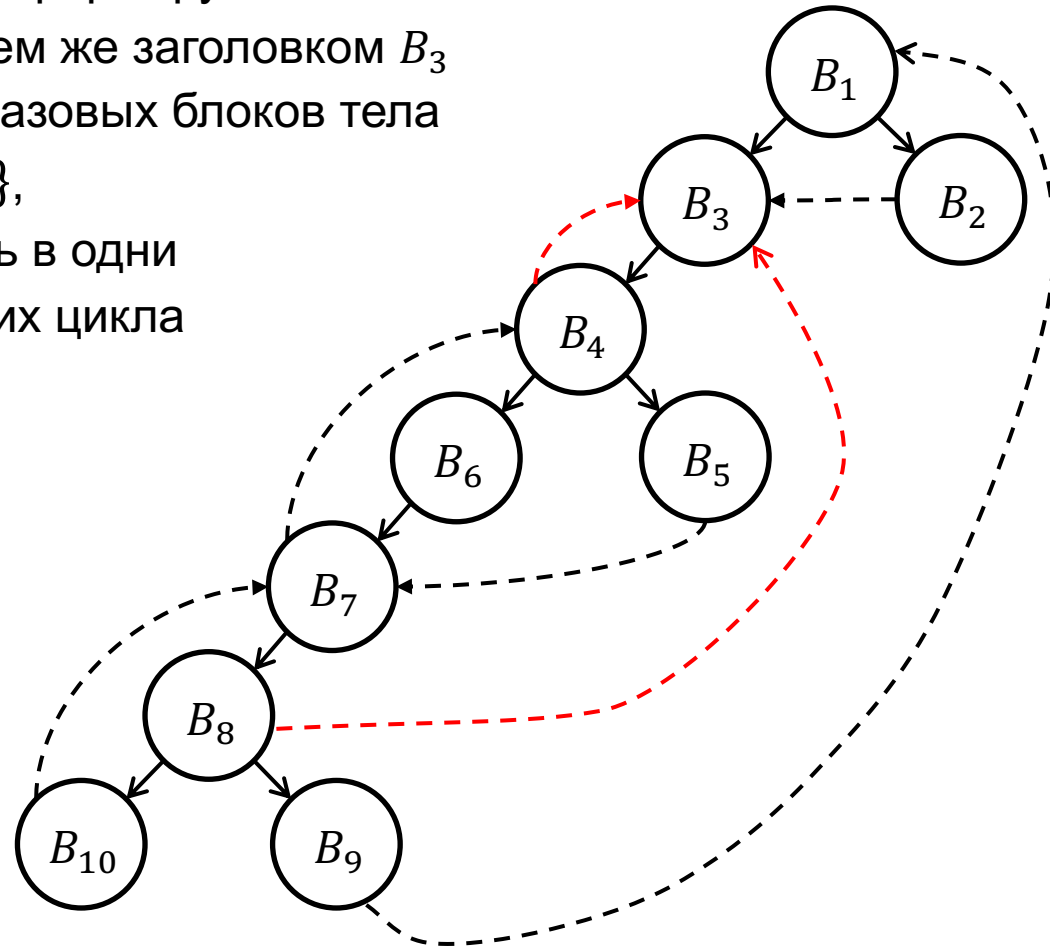
- На рисунке справа – пять обратных дуг:
 $\langle B_4, B_3 \rangle, \langle B_7, B_4 \rangle, \langle B_8, B_3 \rangle, \langle B_9, B_1 \rangle, \langle B_{10}, B_7 \rangle$
- Обратной дуге $\langle B_{10}, B_7 \rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и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



6.1. Выделение естественных циклов

6.1.2 Естественные циклы. Пример.

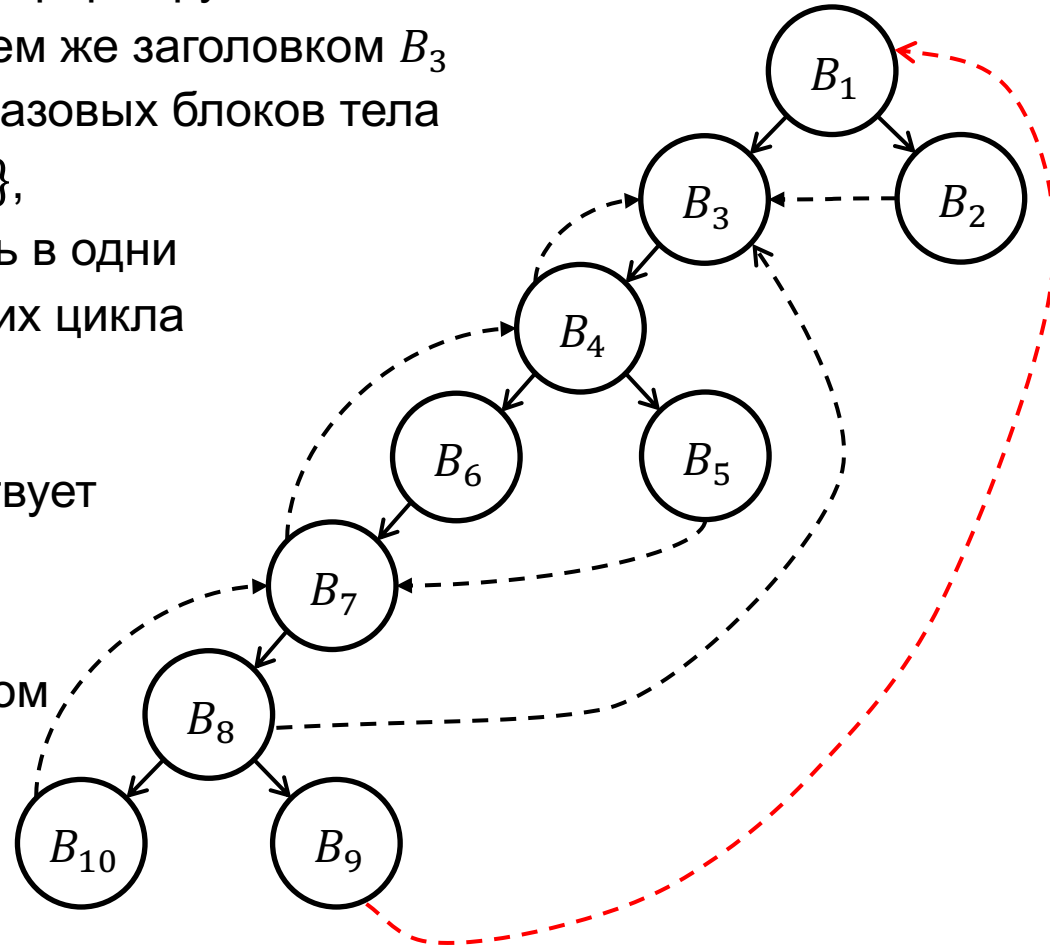
- Обратные дуги $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ формируют естественные циклы с одним и тем же заголовком B_3 и совпадающими множествами базовых блоков тела циклов $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$, поэтому циклы нужно объединить в один. Этот цикл содержит 2 предыдущих цикла ($\langle B_{10}, B_7 \rangle$, $\langle B_7, B_4 \rangle$)



6.1. Выделение естественных циклов

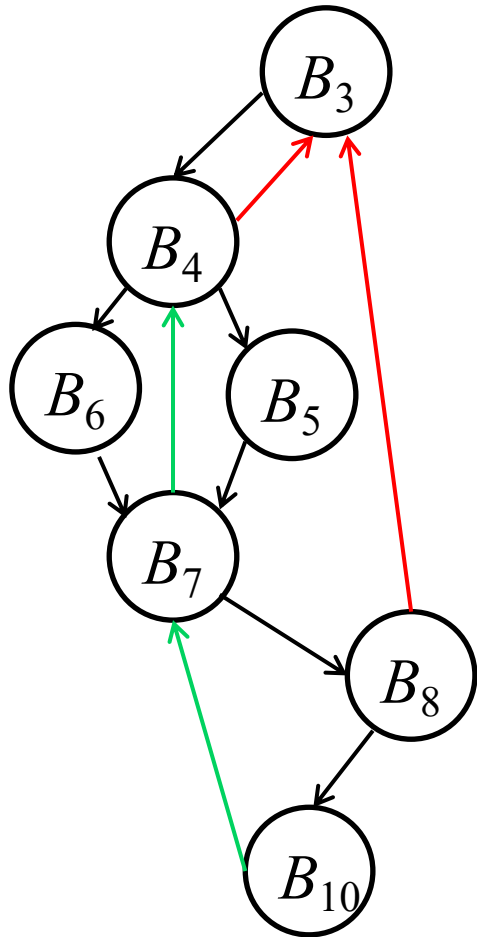
6.1.2 Естественные циклы. Пример.

- Обратные дуги $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ формируют естественные циклы с одним и тем же заголовком B_3 и совпадающими множествами базовых блоков тела циклов $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$, поэтому циклы нужно объединить в один. Этот цикл содержит 2 предыдущих цикла ($\langle B_{10}, B_7 \rangle$, $\langle B_7, B_4 \rangle$)
- Обратной дуге $\langle B_9, B_1 \rangle$ соответствует естественный цикл, содержащий весь граф потока целиком, поэтому является внешним циклом.

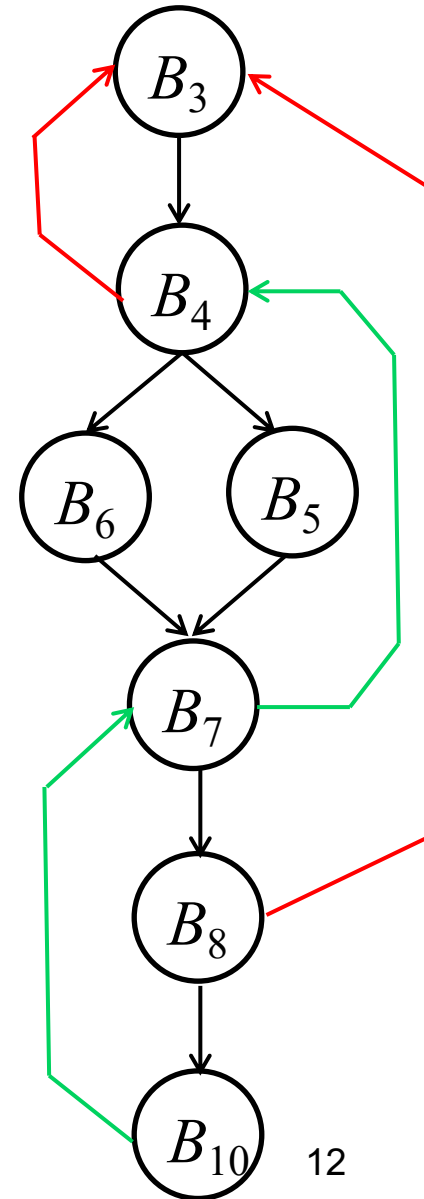


6.1. Выделение естественных циклов

6.1.2 Примеры естественных циклов

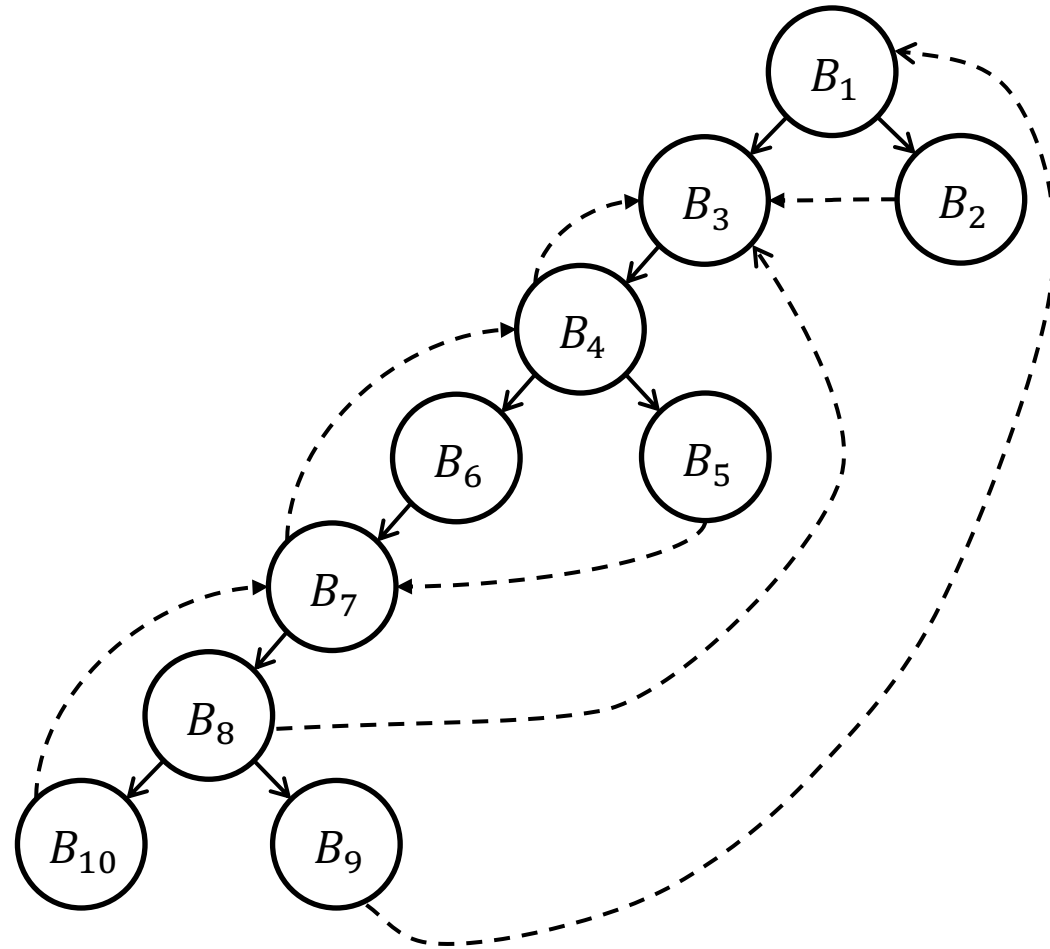
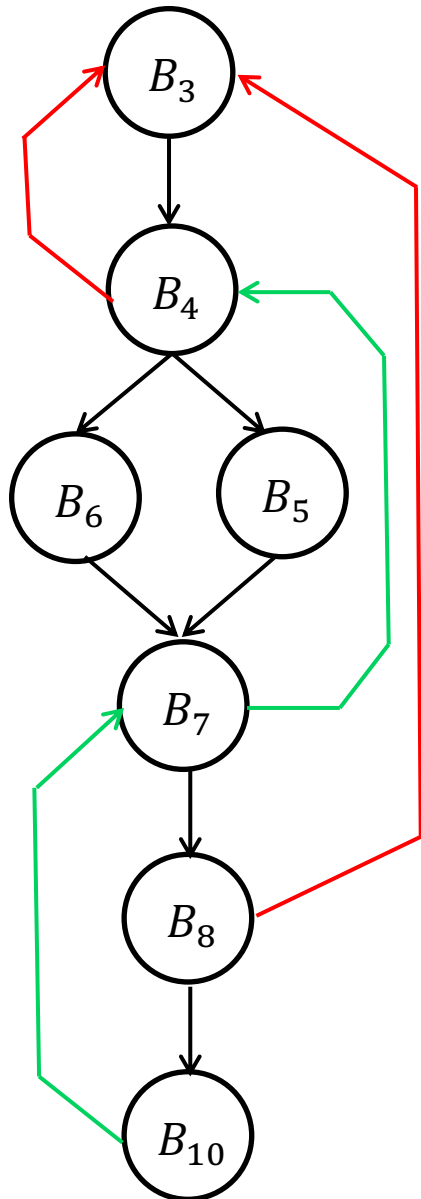


- Обратные дуги $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ формируют естественные циклы с одним и тем же заголовком B_3 и совпадающими множествами базовых блоков тела циклов $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$, поэтому циклы нужно объединить в один
- В объединенный цикл вложены циклы $\langle B_{10}, B_7 \rangle, \langle B_7, B_4 \rangle$
- Изображение ГПУ справа понятнее, чем изображение слева.



6.1. Выделение естественных циклов

6.1.2 Естественные циклы. Пример.



6.1. Выделение естественных циклов

6.1.4 Алгоритм построения естественного 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 естественным 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 .

6.1. Выделение естественных циклов

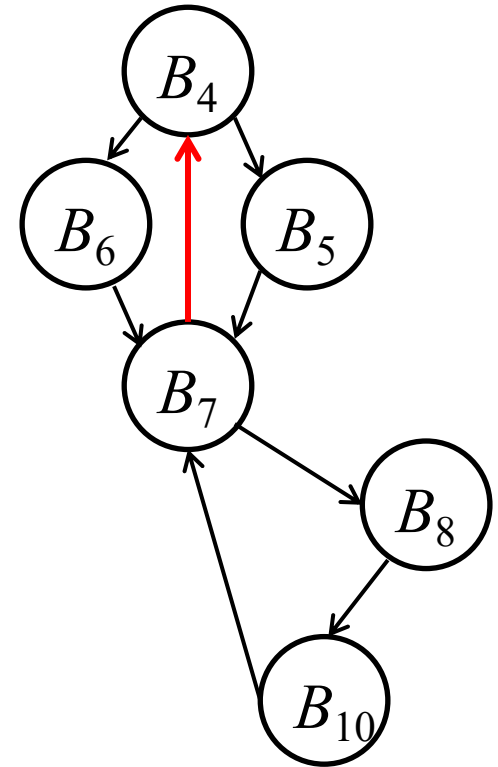
6.1.5 Пример построения естественного цикла по обратной дуге.

- Применим алгоритм 6.1.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$.

Отметим вершину B_4 как посещенную и 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 $Pred(B_7) = \{B_5, B_6, B_{10}\}$



6.1. Выделение естественных циклов

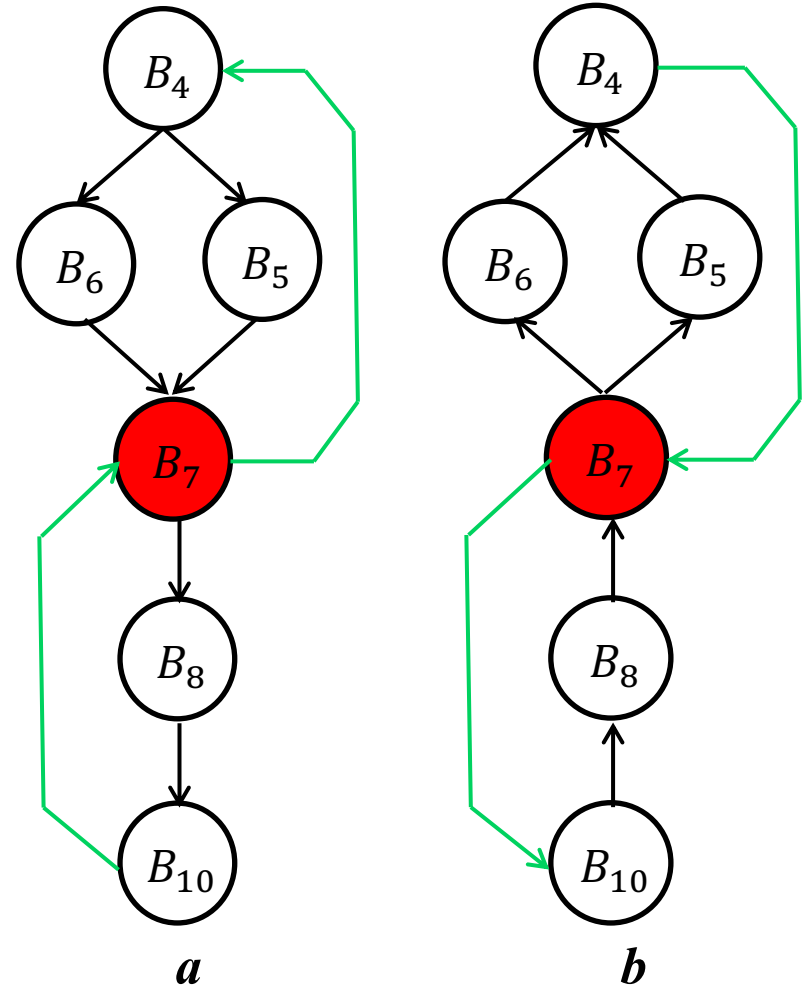
6.1.5 Пример построения естественного цикла по обратной дуге.

- Применим алгоритм 6.1.4 для построения естественного цикла, соответствующего обратной дуге

$\langle B_7, B_4 \rangle$

(рисунок *a*)

- Изменим направление дуг на противоположное (рисунок *b*) и выполним поиск в глубину, начиная с вершины B_7 , отметив вершину B_4 как посещенную.



6.1. Выделение естественных циклов

6.1.5 Пример построения естественного цикла по обратной дуге.

- Применим алгоритм 6.1.4 для построения естественного цикла, соответствующего обратной дуге

$\langle B_7, B_4 \rangle$

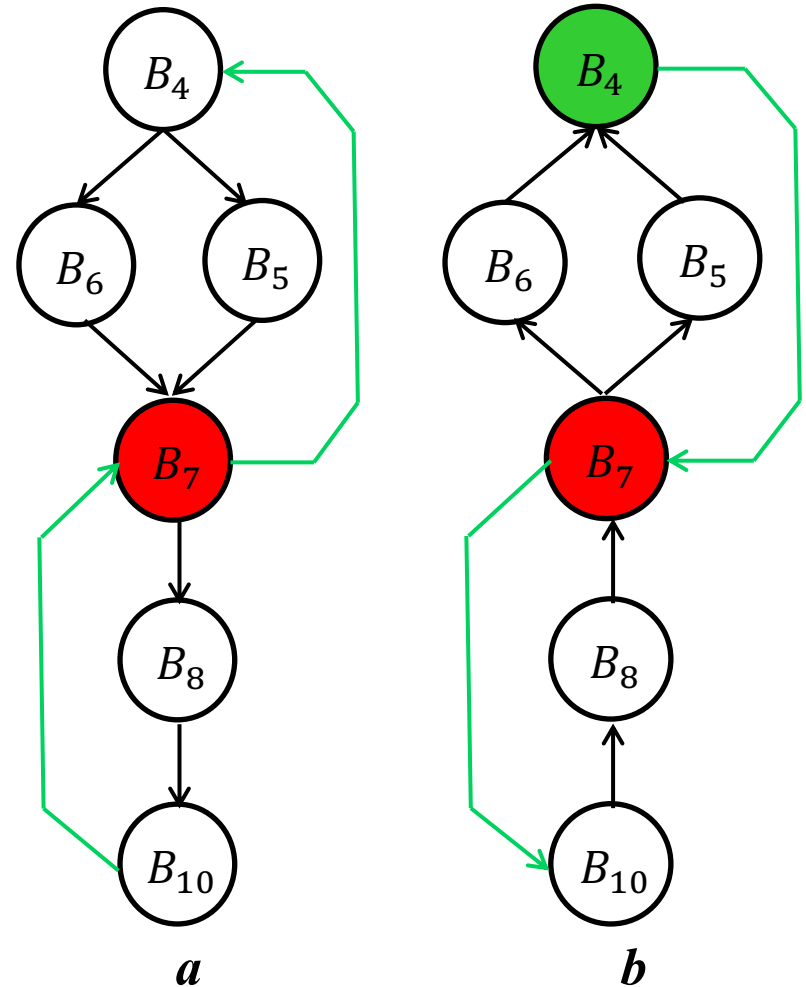
(рисунок *a*)

- На рисунке *b*

$$\text{Succ}(B_7) = \{B_5, B_6, B_{10}\}$$

$$\text{Succ}(B_{10}) = \{B_8\}$$

Следовательно, в состав цикла помимо концов обратной дуги (блоков B_7 и B_4), войдут блоки B_5, B_6, B_{10}, B_8



6.2. Перемещение кода, инвариантного относительно цикла

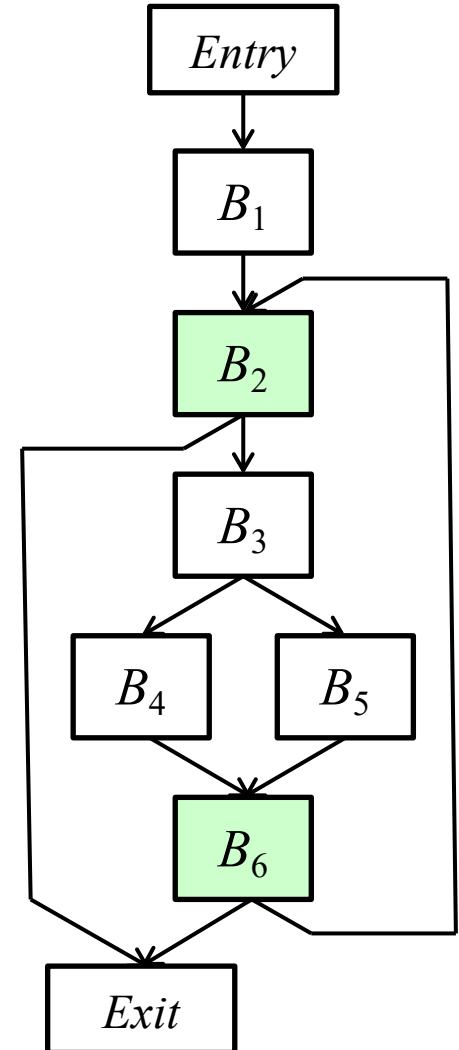
6.2.1 Код, инвариантный относительно цикла

- *Инвариантом цикла* называется:
 - константа c ;
 - переменная u , все определения которой находятся вне цикла;
 - переменная v , определяемая внутри цикла, если у v есть всего одно определение и это определение находится в базовом блоке, являющемся доминатором всех выходов из цикла
- Инструкция *инвариантна относительно цикла*, если все ее операнды являются инвариантами цикла.
- Оптимизация состоит в том, что в ГПУ добавляется еще один базовый блок – *предзаголовок* цикла, в который выносятся из цикла все инструкции, инвариантные относительно цикла.

6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

| | | |
|--|--|---|
| B_1 $b \leftarrow 2$ $i \leftarrow 1$ | B_2 $c \leftarrow 2$ $d \leftarrow b + i$ $\text{if } (i > 100)$ | B_3 $a \leftarrow b + 1$ $\text{if } (i \% 2 = 0)$ |
| B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$ | B_5 $d \leftarrow c$ $f \leftarrow d + 1$ | B_6 $i \leftarrow i + 1$ $\text{if } (i < 100)$ |



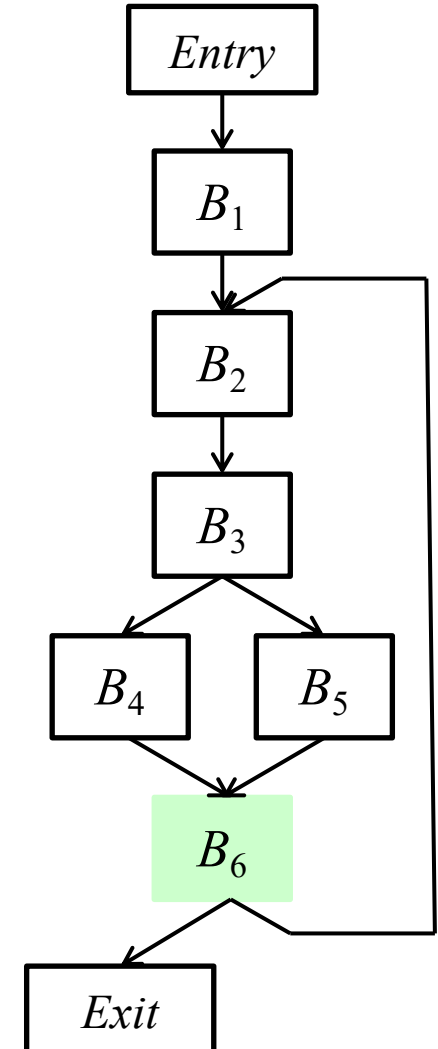
- ◇ Исходный цикл
- ◇ Блок B_1 выполняется до цикла, все остальные – в цикле.
- ◇ Выходы из цикла – блоки B_2 и B_6
- ◇ Блоки B_3 , B_4 , B_5 и B_6 не являются доминаторами блока B_2 . Следовательно, из указанных блоков нельзя выносить код, иначе это может привести к некорректной оптимизации, т.к. значение переменных, записываемых в таких «инвариантах», будет изменено (при выходе из цикла на 1-й же итерации по недоминируемой дуге)

6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

| | | |
|--|--|---|
| B_1 $b \leftarrow 2$ $i \leftarrow 1$ | B_2 $d \leftarrow b + i$ $c \leftarrow 2$ | B_3 $a \leftarrow b + 1$ $\text{if } (i \% 2 = 0)$ |
| B_4 $d \leftarrow a + d$ $e \leftarrow d + 1$ | B_5 $d \leftarrow c$ $f \leftarrow d + 1$ | B_6 $i \leftarrow i + 1$ $\text{if } (i < 100)$ |

- ◇ Изменим исходный цикл, убрав из блока ненужное сравнение. У цикла останется только один выход – блоки B_6



6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

◇ Рассмотрим цикл с заголовком B_2 .

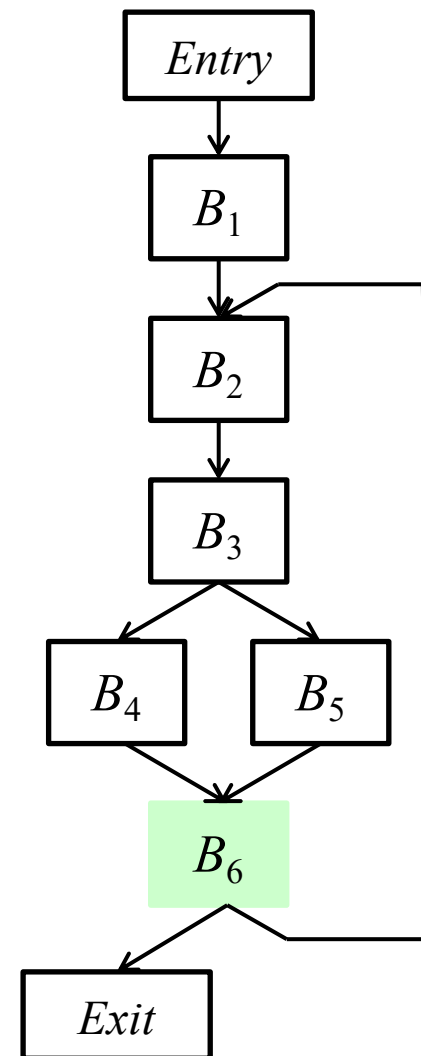
| | | |
|--|--|---|
| B_1 $b \leftarrow 2$ $i \leftarrow 1$ | B_2 $d \leftarrow b+i$ $c \leftarrow 2$ | B_3 $a \leftarrow b+1$ $\text{if } (i\%2=0)$ |
| B_4 $d \leftarrow a+d$ $e \leftarrow d+1$ | B_5 $d \leftarrow c$ $f \leftarrow d+1$ | B_6 $i \leftarrow i+1$ $\text{if } (i<100)$ |

◇ Инструкции $a \leftarrow b+1$, $c \leftarrow 2$, инвариантны относительно цикла:

$a \leftarrow b+1$: b – инвариант цикла, так как его определение находится вне цикла,

1 – инвариант цикла, как константа

$c \leftarrow 2$: 2 – инвариант цикла, как константа

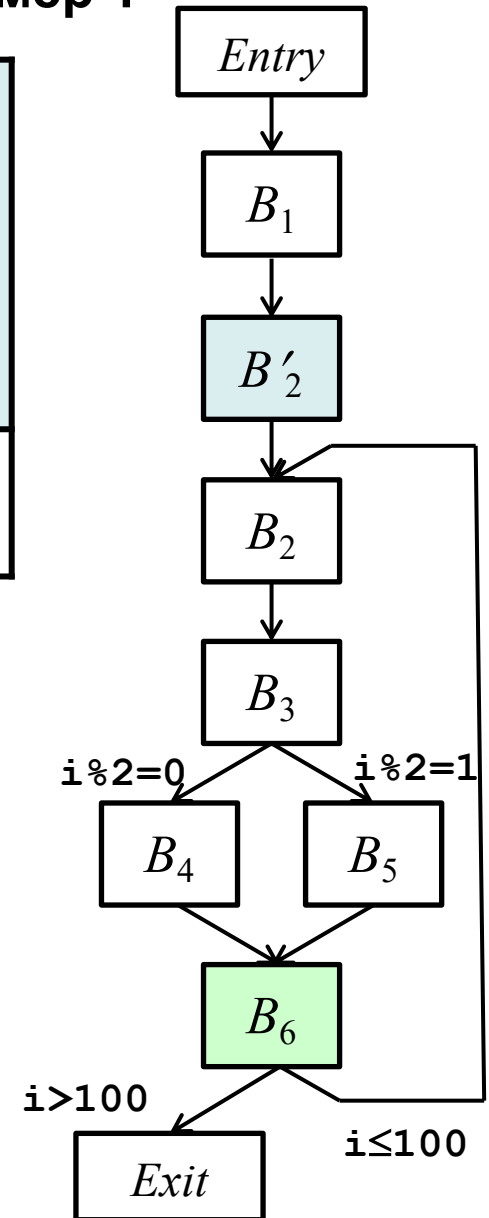


6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

| | | |
|--|---|--|
| B_1 $b \leftarrow 2$ $i \leftarrow 1$ | B_2 $d \leftarrow b+i$ $c \leftarrow 2$ | B'_2 |
| | B_3 $a \leftarrow b+1$ $\text{if}(i\%2==0)$) | |
| B_4 $d \leftarrow a+d$ $e \leftarrow d+1$ | B_5 $d \leftarrow c$ $f \leftarrow d+1$ | B_6 $i \leftarrow i+1$ $\text{if}(i>100)$ |

◇ После добавления предзаголовка (B'_2)

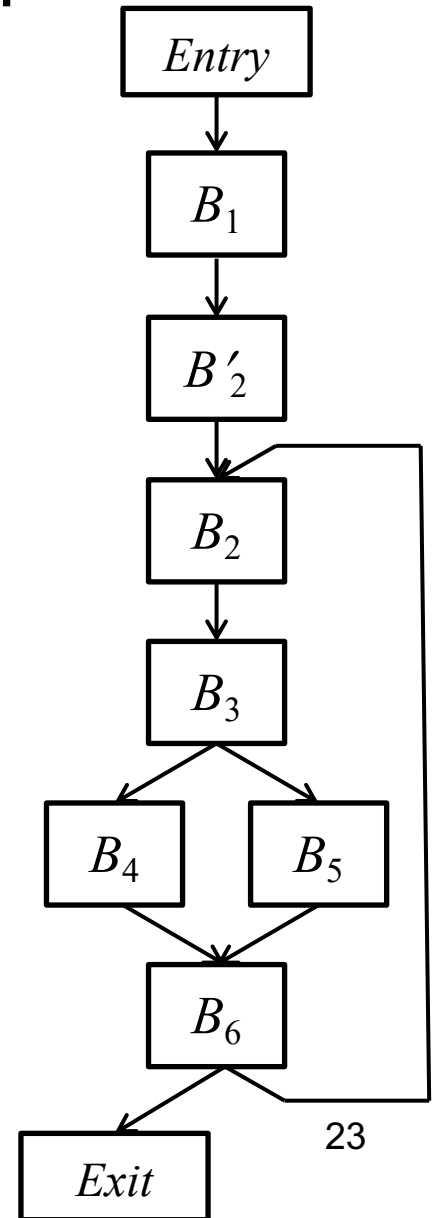


6.2. Перемещение кода, инвариантного относительно цикла

6.2.1 Код, инвариантный относительно цикла. Пример 1

| | | |
|--|--|--|
| B_1 $b \leftarrow 2$ $i \leftarrow 1$ | B_2 | B'_2 $a \leftarrow b+1$ $c \leftarrow 2$ |
| | B_3 $\text{if}(i\%2=0)$ | |
| B_4 $d \leftarrow a+d$ $e \leftarrow d+1$ | B_5 $d \leftarrow c$ $f \leftarrow d+1$ | B_6 $i \leftarrow i+1$ $\text{if}(i>100)$ |

◇ После вынесения инвариантного кода в предзаголовок

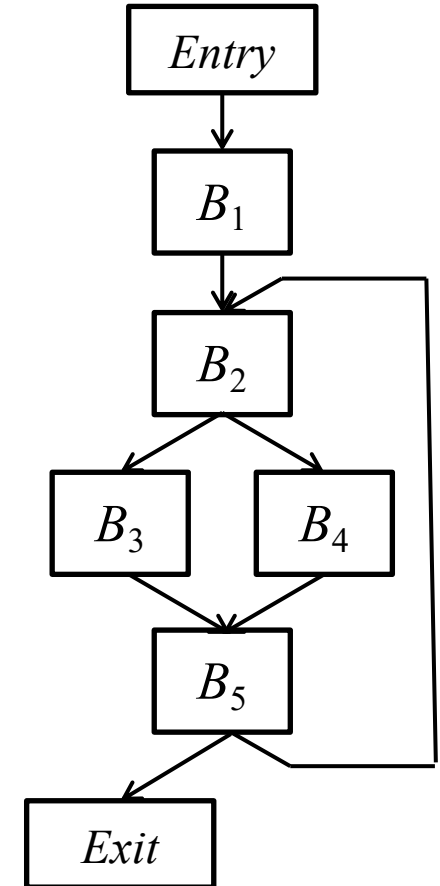


6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Код, инвариантный относительно цикла. Пример 2

| | | | | | |
|-------|--|-------|---|-------|---|
| B_1 | $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 | $a \leftarrow b+c$ $\text{if } (i \% 2 = 0)$ | B_5 | $d \leftarrow a+1$ $f \leftarrow e+2$ $i \leftarrow i+1$ $\text{if } (i \leq 100)$ |
| B_3 | $e \leftarrow 2$ | | | | |
| B_4 | $e \leftarrow 3$ | | | | |

- ◇ Исходный цикл
- ◇ Блок B_1 выполняется до цикла, все остальные – в цикле

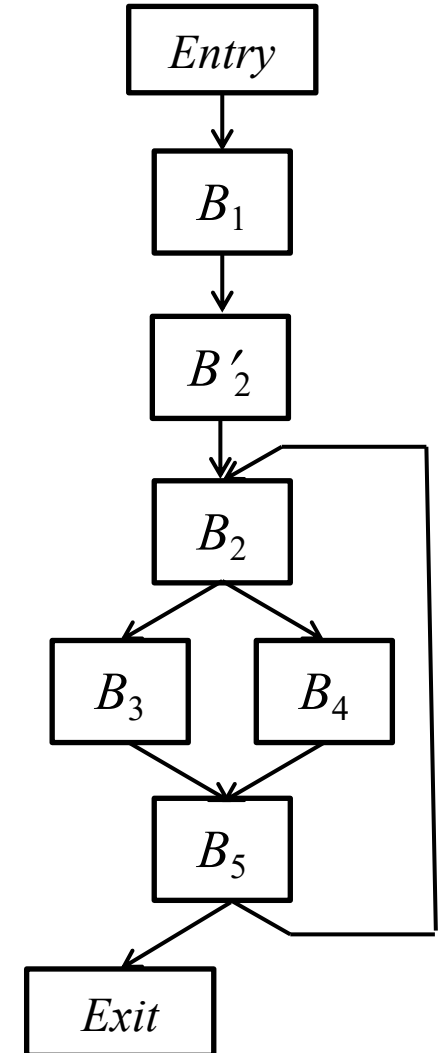


6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $a \leftarrow b+c$ $\text{if}(i\%2=0)$ | B_5 $d \leftarrow a+1$ $f \leftarrow e+2$ $i \leftarrow i+1$ $\text{if}(i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 | |
| B_4 $e \leftarrow 3$ | | |

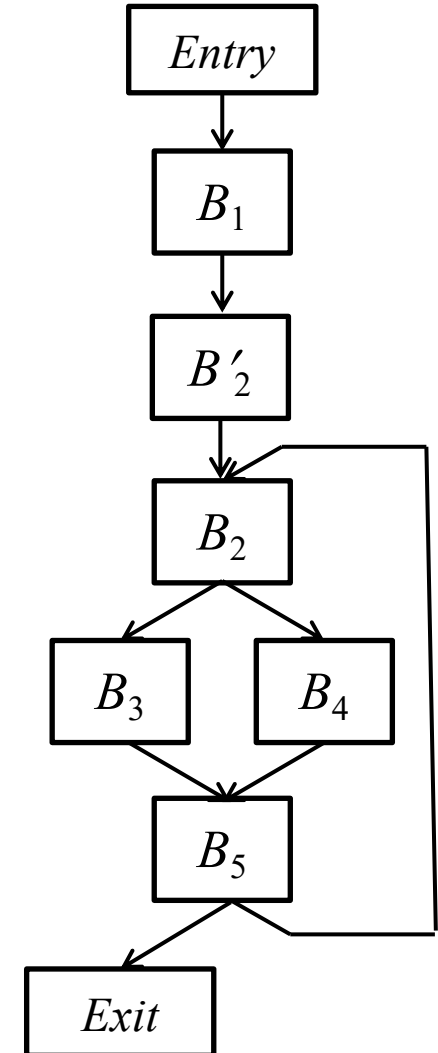
◇ После добавления предзаголовка (B'_2)



6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$ | B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 | |
| B_4 $e \leftarrow 3$ | | |

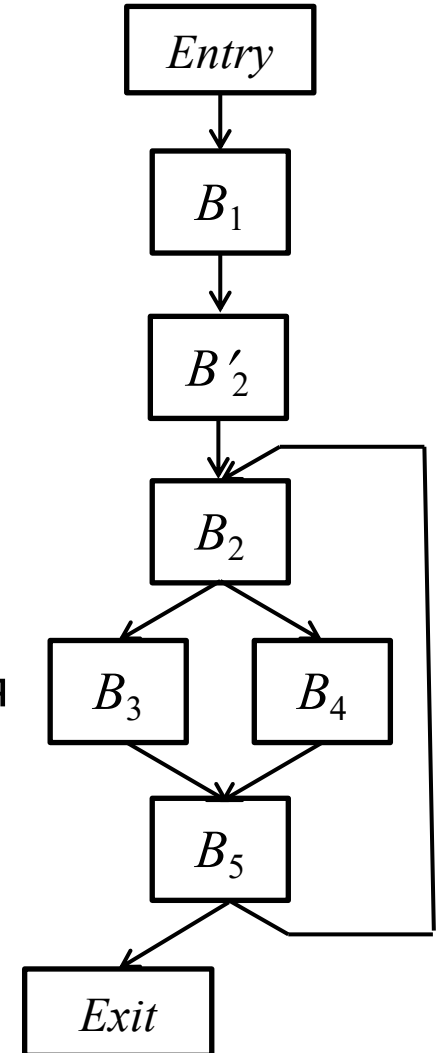


- ◇ Инструкции $a \leftarrow b + c$, $d \leftarrow a + 1$ инвариантны относительно цикла:
 - $a \leftarrow b + c$: определения b и c находятся вне цикла,
 - $d \leftarrow a + 1$: 1 константа
 a определено только один раз в блоке, который является доминатором выхода из цикла
- ◇ Инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла; почему?

6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $a \leftarrow b+c$ $\text{if } (i \% 2 = 0)$ | B_5 $d \leftarrow a+1$ $f \leftarrow e+2$ $i \leftarrow i+1$ $\text{if } (i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b+c$ $d \leftarrow a+1$ | |
| B_4 $e \leftarrow 3$ | | |

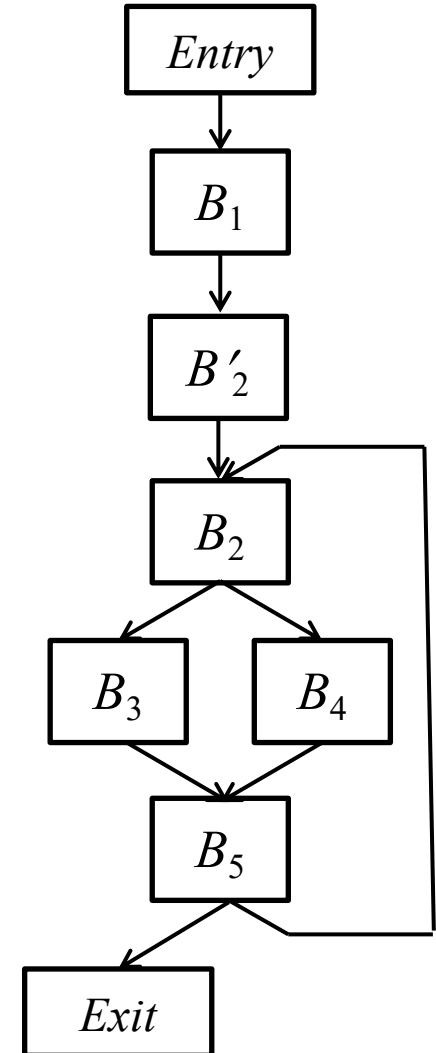


- ◇ Инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла; почему?
- ◇ Во-первых, e определяется внутри цикла не один, а два раза; следовательно, значение e изменяется внутри цикла; при вынесении инструкций в предзаголовков e будет определяться только один раз и в зависимости от порядка инструкций в предзаголовке будет всегда иметь только одно значение (2 или 3)

6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $a \leftarrow b + c$ $\text{if } (i \% 2 = 0)$ | B_5 $d \leftarrow a + 1$ $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$ | |
| B_4 $e \leftarrow 3$ | | |



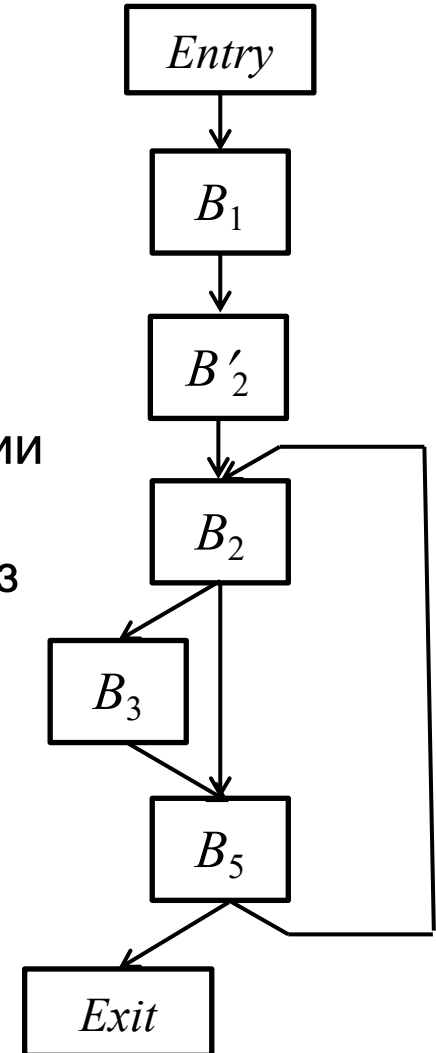
- ◇ Инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла; почему?
- ◇ Во-вторых, предзаголовок цикла является доминатором каждого базового блока, входящего в цикл, включая его заголовок (по построению); следовательно, предзаголовок является доминатором всех выходов из цикла (у рассматриваемого цикла всего один выход); значит в предзаголовке можно только инструкции из базовых блоков являющихся доминаторами всех выходов из цикла.

6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $a \leftarrow b+c$ $\text{if } (i \% 2 = 0)$ | B_5 $d \leftarrow a+1$ $f \leftarrow e+2$ $i \leftarrow i+1$ $\text{if } (i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b+c$ $d \leftarrow a+1$ | |

- ◇ В предзаголовок можно выносить только инструкции из базовых блоков являющихся доминаторами всех выходов из цикла. Если, например, удалить из цикла блок B_4 , то e будет присваиваться только одно значение. Но даже в этом случае нельзя выносить инструкцию $e \leftarrow 2$ в предзаголовок, так как блок B_3 все равно не будет доминатором выхода и результат выполнения программы может измениться.

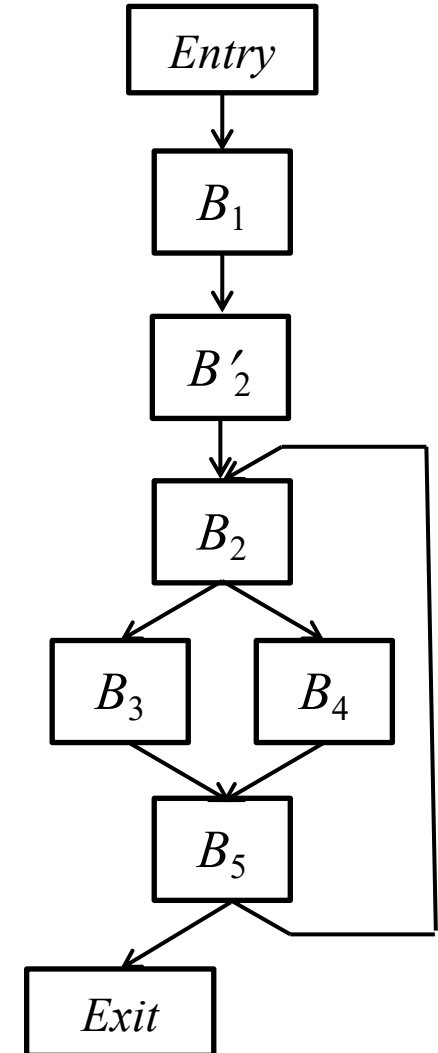


6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $\text{if}(i \% 2 = 0)$ | B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if}(i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$ | |
| B_4 $e \leftarrow 3$ | | |

- ◇ Таким образом, выносить в предзаголовок можно только те инструкции, которые выполняются в блоках, доминирующих над всеми выходами из цикла (в рассматриваемом примере у цикла всего один выход) и которые выполняют единственное присваивание соответствующей переменной.



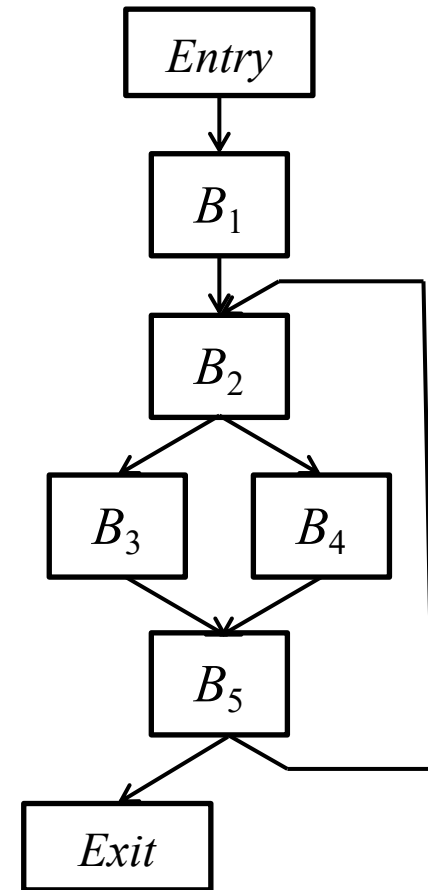
6.2. Перемещение кода, инвариантного относительно цикла

6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $\text{if}(i \% 2 = 0)$ | B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if}(i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$ | |
| B_4 $e \leftarrow 3$ | | |

Итак:

- ◇ $a \leftarrow b + c$ инвариантный код, и его можно вынести в предзаголовок B'_2 (b и c определяются вне цикла, a определяется в блоке B_2 , который доминирует над единственным выходом из цикла – блоком B_5)
- ◇ $d \leftarrow a + 1$ инвариантный код, и его можно вынести в предзаголовок B'_2 (a определяется внутри цикла только в блоке B_2 , который доминирует над единственным выходом из цикла – блоком B_5)



6.2. Перемещение кода, инвариантного относительно цикла

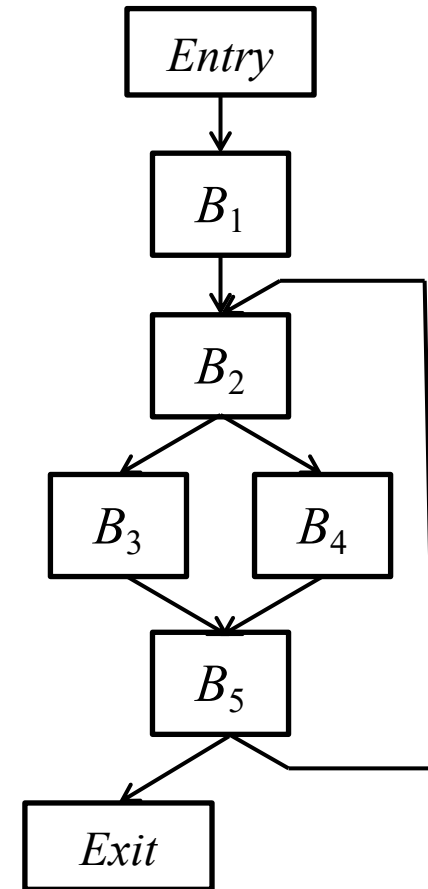
6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $\text{if } (i \% 2 = 0)$ | B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if } (i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$ | |
| B_4 $e \leftarrow 3$ | | |

Итак:

- ◇ инструкции $e \leftarrow 2$ и $e \leftarrow 3$ не инвариантны относительно цикла, так как выполняются в блоках, не являющихся доминаторами выхода из цикла – блока B_5
- ◇ инструкция $f \leftarrow e + 2$ не инвариантна относительно цикла, так как e может изменяться во время выполнения цикла

Эти инструкции нельзя выносить в предзаголовки B'_2 .



6.2. Перемещение кода, инвариантного относительно цикла

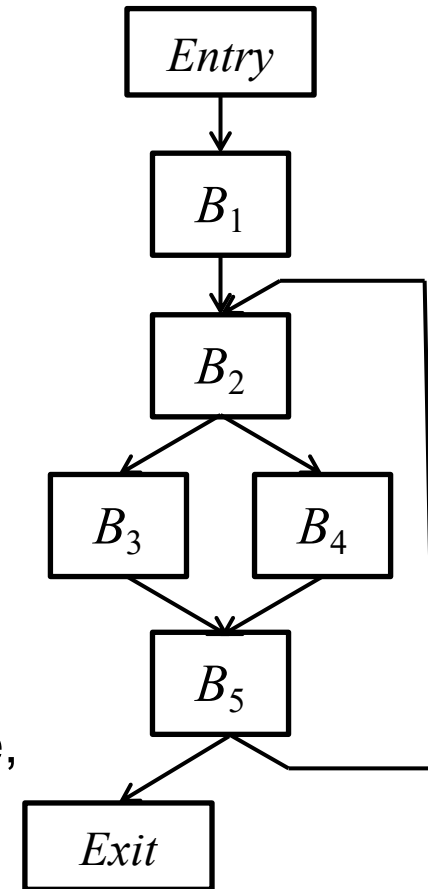
6.2.2 Алгоритм перемещения инвариантного кода. Пример 2

| | | |
|--|---|---|
| B_1 $b \leftarrow 2$ $c \leftarrow 3$ $i \leftarrow 1$ | B_2 $\text{if}(i \% 2 = 0)$ | B_5 $f \leftarrow e + 2$ $i \leftarrow i + 1$ $\text{if}(i \leq 100)$ |
| B_3 $e \leftarrow 2$ | B'_2 $a \leftarrow b + c$ $d \leftarrow a + 1$ | |
| B_4 $e \leftarrow 3$ | | |



Таким образом выражение присваивания $x = e$, где e – инвариант цикла, можно выносить в предзаголовки цикла, если выполняются следующие три условия:

1. это единственное определение x в цикле,
2. оно является доминатором всех выходов из цикла, на которых x живо,
3. это единственное определение x , достигающее использований x внутри цикла: x не является живым на входе в заголовок цикла.



6.2. Перемещение кода, инвариантного относительно цикла

6.2.3 Алгоритм перемещения инвариантного кода

◇ **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, и которые находятся в блоках, доминирующих все выходы из цикла
5. Повторять шаги 2 – 4, пока инвариантные инструкции не перестанут выделяться
6. Переместить все выделенные инструкции в предзаголовок.

6.3. Другие оптимизации циклов

6.3.1 Обзор оптимизирующих преобразований циклов

- ◇ Вынесение инвариантного кода в предзаголовок (уже рассмотрели)
- ◇ Исключение умножений при вычислении индуктивных переменных (замена $t \leftarrow a*i + c$, где c и a – инварианты цикла, в частности – константы, на $t \leftarrow c + a$)
- ◇ Исключение проверок границ массивов
- ◇ Раскрутка циклов (чтобы сократить число проверок на окончание)
- ◇ Перестановка циклов в гнезде и другие преобразования, повышающие локальность данных, обрабатываемых в цикле (оптимизация работы с КЭШем)

6.4. Индуктивные переменные

6.4.1 Определение индуктивной переменной

- ◇ **Определение.** Переменная v называется *индуктивной переменной* цикла L , если на каждой итерации цикла значение v увеличивается на значение переменной (или константы) c , являющейся инвариантом цикла, то есть на каждом витке цикла выполняется инструкция:
$$v \leftarrow v + c$$
- ◇ Тривиальным примером индуктивной переменной является счетчик цикла, то есть переменная i , которой в начале цикла присваивается значение 0 (или 1) и значение которой на каждом витке цикла увеличивается на 1.
- ◇ Если индуктивная переменная v на каждой итерации цикла принимает значение $c * i + d$, где c и d – инварианты цикла, то v является линейной индуктивной переменной.
- ◇ Когда в цикле удастся обнаружить индуктивные переменные, становится возможным выполнить различные оптимизации этого цикла.

6.4. Индуктивные переменные

6.4.2 Семейства индуктивных переменных

- ◇ Нетрудно доказать, что линейные функции от индуктивных переменных тоже индуктивные переменные.
Следовательно, в циклах, могут встречаться семейства индуктивных переменных. Обычно в циклах бывает несколько таких семейств.
- ◇ *Основной* индуктивной переменной по определению является индуктивная переменная \dot{i} , все определения которой эквивалентны определению вида $\dot{i} \leftarrow +, \dot{i}, c$, где c – инвариант цикла (как правило, константа). Нет необходимости, чтобы значение c было одинаковым в каждом таком определении.
Основная индуктивная переменная является *линейной*, если в цикле имеется всего одно ее определение, причем это определение является доминатором (или постдоминатором) всех остальных вершин цикла.
- ◇ *Производная* индуктивная переменная \dot{j} выражается через одну из основных индуктивных переменных \dot{i} как $c * \dot{i} + d$, где c и d – инварианты цикла.
- ◇ Основная индуктивная переменная \dot{i} и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*.

6.4. Индуктивные переменные

6.4.2 Семейства индуктивных переменных

◇ Основная индуктивная переменная i и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*.

Для производной индуктивной переменной j удобно использовать обозначение $j = \langle i, c, d \rangle$.

Применяя это обозначение к i , получим $i = \langle i, 1, 0 \rangle$

◇ В примере справа:

◇ i – основная индуктивная переменная;

◇ j – линейная основная индуктивная переменная;

◇ k и l – линейные производные индуктивные переменные семейства j ;

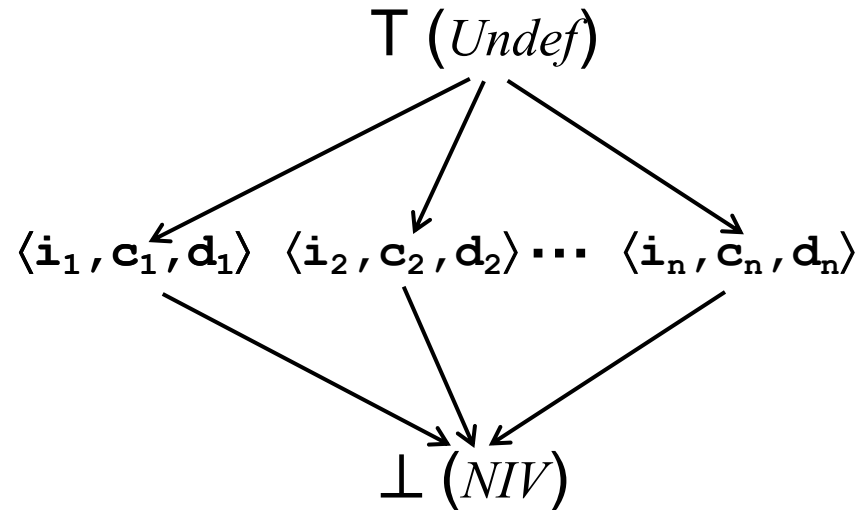
◇ m – производная индуктивная переменная семейства i .

```
while (i < 10) {  
    j = j + 2;  
    if (j > 4)  
        i = i + 1;  
    i = i - 1;  
    k = j + 10;  
    l = k * 4;  
    m = i * 8;  
}
```

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Для обнаружения индуктивных переменных используется анализ потока данных в пределах цикла.
- ◇ Элементами полурешетки, на которой проводится анализ, являются «тройки» $\langle \mathbf{i}_k, \mathbf{c}_k, \mathbf{d}_k \rangle$ (см. рисунок).
- ◇ По определению элементы полурешетки, отличные от \top и \perp , несравнимы один с другим.
- ◇ На элемент \perp отображаются переменные, не являющиеся индуктивными (он так и называется *NIV* – *Not Induction Variable*)
- ◇ На элемент \top отображаются переменные, которым не было присвоено никаких значений, и по этой причине невозможно установить, является ли переменная индуктивной. Поэтому на элемент \top отображаются переменные, имеющие значение «неопределенный (неинициализированный) элемент» – *Undef*.
В начале анализа всем переменным присваивается значение *Undef*.



6.4. Индуктивные переменные

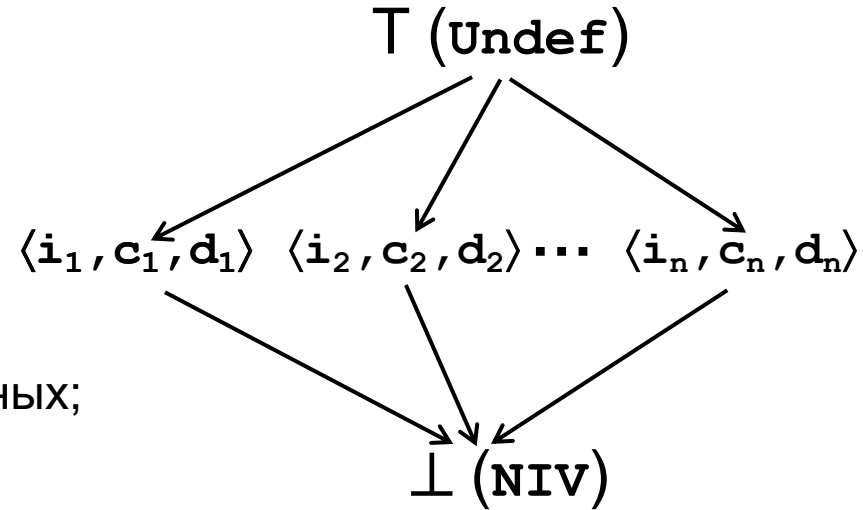
6.4.3 Обнаружение индуктивных переменных

◇ Рассмотрим переменную x .

Возможны следующие варианты:

- ◇ x в рассматриваемом цикле ничего не присваивается; тогда над x надписывается **Undef**;
- ◇ x присваивается значение $c_k * i_k + d_k$, где i_k – одна из основных индуктивных переменных; тогда x ставится в соответствие $\langle i_k, c_k, d_k \rangle$; самой индуктивной переменной i_k ставится в соответствие $\langle i_k, 1, 0 \rangle$.

- ◇ Если в точку сбора нескольких путей, для x , которому уже соответствует $\langle i_k, c_k, d_k \rangle$, на другом пути соответствует значение $c_m * i_m + d_m$, где i_m – другая основная индуктивная переменная; тогда старое соответствие для x стирается, и для него записывается **NIV**.



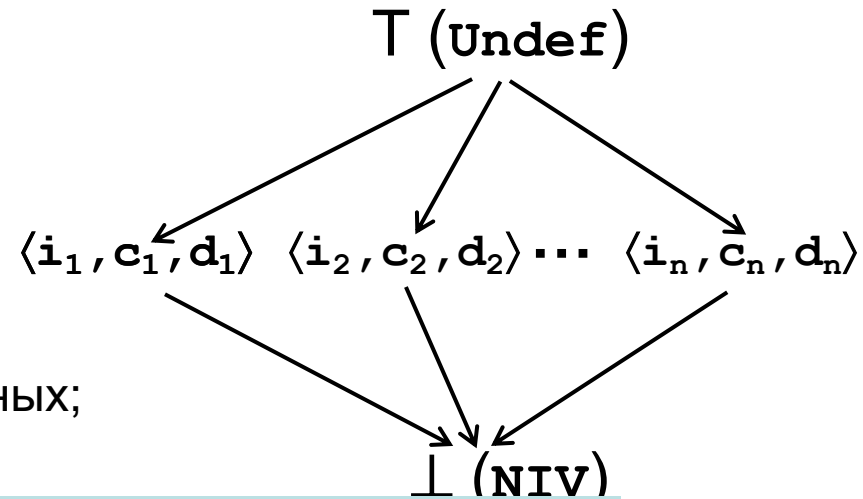
6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

◇ Рассмотрим переменную x .

Возможны следующие варианты:

- ◇ x в рассматриваемом цикле ничего не присваивается; тогда над x надписывается **Undef**;
- ◇ x присваивается значение $c_k * i_k + d_k$, где i_k – одна из основных индуктивных переменных; тогда x ставится в соответствие



«Соответствие» означает, что существует отображение (функция), которое каждой переменной, встречающейся в исследуемом цикле, ставит в соответствие необходимый элемент рассматриваемой полурешетки в текущей точке программы. На значениях этой функции будет задана передаточная функция анализа потока данных (т.е. передаточная функция отображает состояние программы до начала базового блока в состояние после него, где «состояние программы», в свою очередь, является отображением переменных на элементы полурешетки.

ение
; тогда
IV.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Чтобы начать анализ потока данных необходимо построить начальное отображение (**var**, **val**), где **var** – переменная, а **val** – ее значение. Эти пары помещаются в очередь **Worklist**.
- ◇ Сначала находят **основные** индуктивные переменные. Для этого просматриваются все инструкции цикла и находятся все переменные **i**, которым сначала присваивается целое значение **c** ($i \leftarrow c$), а потом ее значения изменяются только инструкциями вида $i \leftarrow +, i, d$, или $i \leftarrow +, d, i$, где **d** – инвариант цикла.
- ◇ Значением основной индуктивной переменной **i**, т. е. **val(i)** является «тройка» $\langle i, 1, 0 \rangle$, так как очевидно, что $i = 1 * i + 0$. Каждой основной индуктивной переменной **i** соответствует пара $(i, \langle i, 1, 0 \rangle)$, которая помещается в **Worklist**.
- ◇ Отметим, что присваивание $i \leftarrow c$ должно быть **только одно**. (Вообще говоря, если таких присваиваний несколько, т. е. есть **n** инструкций $i \leftarrow c_k$ с разными c_k , ($k = 1, \dots, n$), то переменная **i** может расщепляться на **k** переменных i_k , но без применения SSA формы это требует усложнения алгоритма с тем, чтобы отслеживать, какое определение достигает использования, и поэтому мы такие случаи сейчас не рассматриваем).

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ Чтобы начать анализ потока данных необходимо построить начальное отображение (**var**, **val**), где **var** – переменная, а **val** – ее значение. Эти пары помещаются в очередь **Worklist**.
- ◇ Если обнаруживаются инструкции вида
$$p \leftarrow +, j, c \text{ или } q \leftarrow *, j, c,$$
где j – индуктивная переменная семейства i : $\text{val}(j) = \langle i, a, b \rangle$, а c – инвариант цикла, то p и q – тоже индуктивные переменные семейства i , причем
$$\text{val}(p) = \langle i, a, b+c \rangle \text{ и } \text{val}(q) = \langle i, a*c, b*c \rangle.$$
- ◇ Если обнаруживаются инструкции вида $p \leftarrow e$, где e – выражение, не содержащее индуктивных переменных, то $\text{val}(p) = \text{NIV}$
- ◇ Всем остальным переменным сопоставляется значение \top (**Undef**)
- ◇ Все пары (**var**, **val**), найденные на первом этапе, кроме пар, у которых $\text{val} = \text{NIV}$ помещаются в том порядке, в котором они были найдены, в очередь **Worklist**.
- ◇ На этом первый этап алгоритма обнаружения индуктивных переменных заканчивается.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

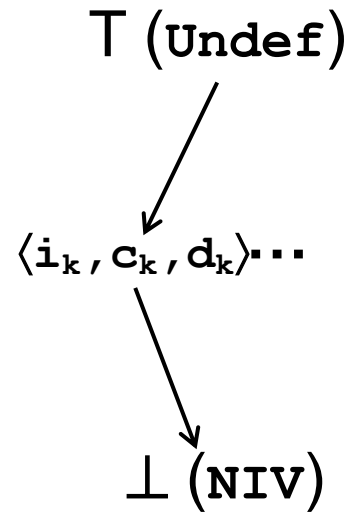
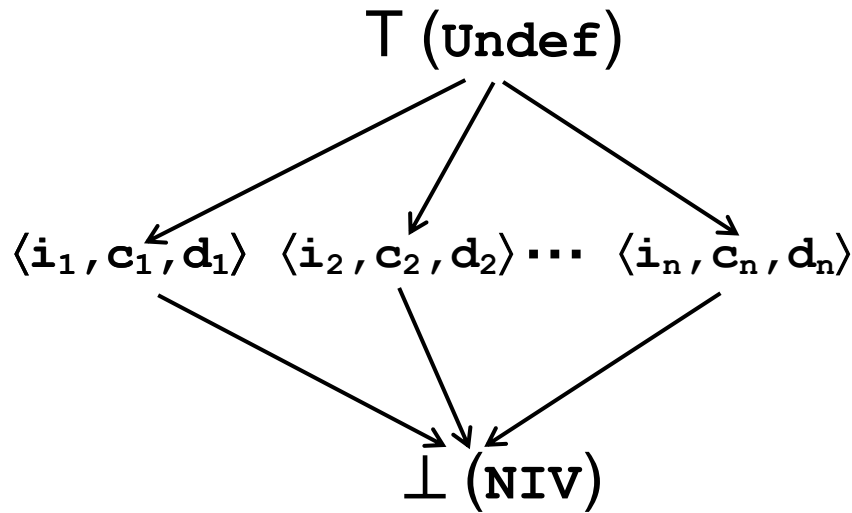
- ◇ После того, как начальное отображение построено и **Worklist** заполнен, начинается второй этап алгоритма обнаружения индуктивных переменных.
 - ◇ Из очереди **Worklist** выбирается очередная пара (**var**, **val**)
 - ◇ Если эта пара имеет вид (**i**, $\langle \mathbf{i}, \mathbf{1}, \mathbf{0} \rangle$), она помещается в список найденных индуктивных, начиная семейство **i**.

Теперь все индуктивные переменные, значение которых имеет первым членом «тройки» **i**, будут помещаться в это семейство.
 - ◇ Если эта пара имеет вид (**j**, $\langle \mathbf{i}, \mathbf{a}, \mathbf{b} \rangle$), выполняется обход ГПУ, чтобы выяснить, не превратится ли **val(j)** в **NIV**. Каждая пара анализируется независимо от других пар. Это связано со структурой полурешетки. В основном уточняются кандидаты в индуктивные переменные, имеющие значение **Undef**.
 - ◇ Анализ состоит в том, чтобы просмотреть все базовые блоки, в которых встречается **j** и убедиться, что во всех блоках **val(j)** остается в своем семействе индуктивных переменных. Основную роль при таком анализе играет операция сбора для функции **val**.

6.4. Индуктивные переменные

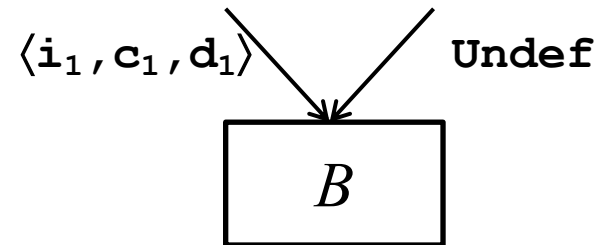
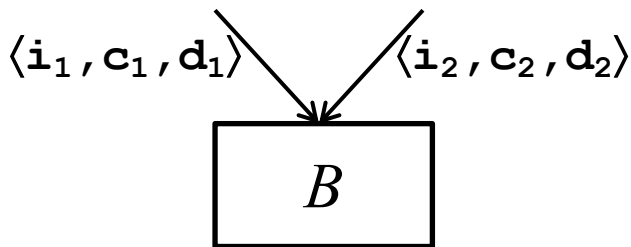
6.4.3 Обнаружение индуктивных переменных

- ◇ Каждое значение анализируется самостоятельно на полурешетке, изображенной на правом рисунке.



Анализ всех значений выполняется на декартовом произведении таких полурешеток

В результате анализа часть ИП станет **NIV**, а часть **Undef** станет ИП



$$\langle i_1, c_1, d_1 \rangle \wedge \langle i_2, c_2, d_2 \rangle = \begin{cases} \langle i_1, c_1, d_1 \rangle & \text{если } i_1 = i_2 \\ NIV & \text{иначе} \end{cases}$$

$$\langle i_1, c_1, d_1 \rangle \wedge Undef = \langle i_1, c_1, d_1 \rangle$$

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◇ В результате анализа будет получено несколько семейств индуктивных переменных и станут возможными оптимизации, связанные с индуктивными переменными – снижение сложности операций и исключение индуктивных переменных.

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций

- ◇ Будет показано, что для индуктивной переменной можно заменить умножение сложением (т. е. арифметическую операцию, выполняемую более сложным (и долгим) алгоритмом заменить на более простую и быструю). Причем при указанной замене количество выполняемых операций (инструкций) не изменится.
- ◇ Рассмотрим простейший пример:

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

Значения переменной j вычисляются с использованием умножения, но j – производная индуктивная переменная $\langle i, 3, a \rangle$, принадлежащая семейству основной индуктивной переменной i .

- ◇ Как вычислять j , используя сложение вместо умножения?

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, продолжение примера

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ Как вычислять j , используя сложение вместо умножения?
- ◇ Сделать это очень просто:
Для производной индуктивной переменной j из семейства основной индуктивной переменной i : $\text{val}(j) = \langle i, c, d \rangle$ необходимо выполнить следующий простой алгоритм:
 1. Создать новые переменные s и k , и в предзаголовке цикла выполнить присваивания $s = c*i + d$; и $k = c*h$;
 2. В теле цикла заменить определение $j = e$ на $j = s$;
 3. В теле цикла после присваивания $i = i + h$ вставить присваивание $j = j + k$;

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, продолжение примера

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

◇ Применив алгоритм к рассматриваемому циклу, получим

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    j = s;
    a[j] = a[j] + 1;
    i = i + 2;
    s = s + 6;
}
```

◇ И, что интересно, в преобразованном цикле нет явной зависимости между j и i : счетчик цикла и индекс элемента массива изменяются как бы независимо.

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, окончание примера

- ◇ Таким образом, алгоритм снижения стоимости позволил заменить в теле цикла

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

умножение ($j = p + 3*i$) на сложение ($s = s + 6$),

в результате чего получился цикл

```
i = 1;
s = p + 3*i;          //предзаголовок цикла
while (i < a.length) {
    j = s;
    a[j] = a[j] + 1;
    i = i + 2;
    s = s + 6;
}
```

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

◇ Рассмотрим оптимизированный цикл

```
i = 1;  
s = p + 3*i;  
while (i < a.length) {  
    j = s;  
    a[j] = a[j] + 1;  
    i = i + 2;  
    s = s + 6;  
}
```

◇ В нем три индуктивных переменных i , j и s .

Анализ покажет, что две из них лишние, и их можно исключить.

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    j = s;
    a[j] = a[j] + 1;
    i = i + 2;
    s = s + 6;
}
```



Во-первых, переменные j и s – это, по существу, одна переменная, так как s используется только для приращения индекса j . Эти переменные (j и s) можно объединить в одну, например s и исключить лишнюю индуктивную переменную j . Цикл после этого будет иметь вид (лишняя индуктивная переменная j исключена):

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    a[s] = a[s] + 1;
    i = i + 2;
    s = s + 6;
}
```

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

```
i = 1;
s = p + 3*i;
while (i < a.length) {
    a[s] = a[s] + 1;
    i = i + 2;
    s = s + 6;
}
```

◇ **Во-вторых**, переменная i используется только в качестве счетчика цикла. Такие переменные называются *почти бесполезными*. Для того, чтобы исключить i , нужно вычислить в предзаголовке верхнюю границу t для s .

Вспомнив, что s – производная индуктивная переменная $\langle i, 3, p \rangle$ семейства i , получим $t = a + 3*a.length$

◇ После исключения i цикл примет окончательный вид:

```
i = 1;
s = p + 3*i;
t = 3*a.length
while (s < t) {
    a[s] = a[s] + 1;
    s = s + 6;
}
```

◇ **Предзаголовков** можно не оптимизировать: он выполняется всего один раз.

6.4. Индуктивные переменные

6.4.5 Исключение индуктивных переменных

- ◇ Таким образом, снижение стоимости и исключение (лишних) индуктивных переменных позволили оптимизировать исходный цикл

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ Получив следующий оптимизированный цикл:

```
i = 1;
s = p + 3*i;
t = 3*a.length
while (s < t) {
    a[s] = a[s] + 1;
    s = s + 6;
}
```

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ В языках *со строгой типизацией* (например, в языке *Java*), доступ к элементу массива по индексу возможен только тогда, когда индекс элемента не выводит за границы массива. Следовательно, транслятор *Java*-программы на байт-код, а значит и *JIT* должны вставлять в циклы соответствующие проверки: прежде, чем вычислить адрес элемента `a[j]`, компилятор должен убедиться, что `j` – безопасный индекс, не выходящий за пределы массива `a[]`.
- ◇ В частности, в цикл из примера 6.4.4 будет добавлена проверка, которая гарантирует, что индекс `j`, по которому производится доступ, удовлетворяет условиям:

$$0 \leq j < a.length.$$

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

```
i = 1;
while (i < a.length) {
    j = p + 3*i;
    a[j] = a[j] + 1;
    i = i + 2;
}
```

- ◇ В цикл, рассматриваемый в примере 6.4.4, добавится строка (она выделена **ЦВЕТОМ**):

```
    i = 1;
    while (i < a.length) {
        j = p + 3*i;
        if (i < 0 | i ≥ a.length) goto L_err;
        a[j] = a[j] + 1;
        i = i + 2;
    }
```

- ◇ Наличие указанной строки внутри цикла существенно замедлит его выполнение (ведь на каждом витке цикла добавится два сравнения).

6.5. Исключение проверок границ массивов

6.5.2 Решение проблемы.

- ◇ Лучше всего, конечно, вообще исключить проверку, или, по крайней мере, вынести ее в предзаголовок.
 - ◇ Идея в том, что выполнение условия выхода из цикла (в рассматриваемом примере $i < a.length$) часто гарантирует, что обращения к массиву не выведут за его пределы.
 - ◇ Если в этом можно убедиться во время статического анализа, проверка может быть удалена.
 - ◇ Если это выясняется во время динамического анализа (или во время выполнения программы), цикл может быть представлен в двух версиях:
 - ◇ Быстрой, в которой нет проверок границ и
 - ◇ Медленной, в которой такие проверки есть.
- А пользователь пусть сам разбирается, где и когда какую версию использовать.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◇ Рассмотрим проблему исключения проверок границ массивов, когда выполняются следующие условия:

1. Индуктивная переменная j семейства i сравнивается с инвариантом цикла u (проверка условия выхода из цикла $j < u$).
2. Индуктивная переменная k того же семейства i сравнивается с инвариантом цикла n (проверка условия выхода из цикла $k < n$), причем из $j < u$ следует $k < n$.
3. k и j изменяются в одном и том же направлении.
4. Проверка условия $j < u$ находится в блоке, являющемся доминатором блока, в котором находится проверка условия $k < n$.

Утверждение. При выполнении условий 1, 2, 3 и 4 проверка условия $k < n$ избыточна и может быть удалена.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◇ Для обоснования утверждения, выясним, когда из $j < u$ следует $k < n$?

Пусть $j = \langle i, c_j, d_j \rangle$, $k = \langle i, c_k, d_k \rangle$.

Если условие $j < u$ выполняется, то

$$c_j * i + d_j < u$$

Без потери общности можно считать, что $c_j > 0$. Тогда

$$i < \frac{(u - d_j)}{c_j}$$

Следовательно,

$$k = c_k * i + d_k < c_k * \left(\frac{(u - d_j)}{c_j} \right) + d_k$$

Если удастся показать в результате статического или динамического анализа, что правая часть последнего неравенства $\leq n$, то условие $k < n$ будет выполнено.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◇ Итак, необходимо выяснить, верно ли что

$$c_k * \left(\frac{(u-d_j)}{c_j} \right) + d_k \leq n \quad (*)$$

Это можно либо установить во время компиляции, либо (если статический анализ недостаточен) попытаться поместить соответствующую проверку в предзаголовок цикла.

```
while (i < a.length) {  
    a[i] = a[i] + 1;  
    i = i + 1;  
}
```

◇ Для обращения к $a[i]$ в примере выше условие $i < a.length$ оказывается достаточным для одновременной проверки и верхней границы массива $a[]$. В этом нетрудно убедиться во время компиляции, подставив $c_i = 1$, $d_i = 0$, вместо (c_j, d_j) и (c_k, d_k) в формулу (*). Получим верное неравенство:

$$1 * (a.length - 0) / 1 + 0 \leq a.length, \text{ или } a.length \leq a.length.$$

Что касается 0, левой границы для i , то проверку условия $i > 0$ можно вынести в предзаголовок (так как 0 – инвариант цикла).

6.6. Раскрытие циклов

6.6.1 Что это такое

◇ **Простой пример.** Выполним раскрытие внутреннего цикла в гнезде.

◇ Исходное гнездо циклов (на языке C):

```
for(j = 1; j <= nj; j++){
    for(i = 1; i <= ni; i++){
        y[i] += x[j] * m[i][j];
    }
}
```

◇ После раскрытия внутреннего цикла на 4:

```
for(j = 1; j <= nj; j++){
    mod_i = ni % 4;
    if(mod_i >= 1){
        for(i = 1; i <= mod_i; i++){
            y[i] += x[j] * m[i][j];
        }
    }
    for(i = mod_i + 1; i <= ni; i += 4){
        y[i] += x[j] * m[i][j];
        y[i+1] += x[j] * m[i+1][j];
        y[i+2] += x[j] * m[i+2][j];
        y[i+3] += x[j] * m[i+3][j];
    }
}
```

6.6. Раскрутка циклов

6.6.1 Что это такое

◇ **Простой пример.** Выполним раскрутку внутреннего цикла в гнезде.

◇ Исходное гнездо циклов (на языке C):

```
for(i = 1; i <= ni; i++) {
```

Раскрутка цикла

копирует тело цикла для нескольких итераций и корректирует вычисление индексов соответствующим образом.

◇ После раскрутки внутреннего цикла на 4:

```
for(j = 1; j <= nj; j++) {  
    mod_i = ni % 4;  
    if(mod_i >= 1) {
```

Если ni не делится на 4, то остается кусок цикла, который выполняется в коротком *цикле-прологе*.

Назначение цикла-пролога – гарантировать, что количество итераций цикла раскручиваемого на m , кратно m .

Такое преобразование цикла (расщепление цикла для обеспечения удобных границ) называется *выравниванием цикла*.

6.6. Раскрутка циклов

6.6.1 Что это такое.

◇ Простой пример. Но можно выполнить и раскрутку внешнего цикла.

◇ После раскрутки внешнего цикла на 4:

```
mod_j = nj % 4;
if(mod_j >= 1){
    for(j = 1; j <= mod_j; j++){
        for(i = 1; i < ni; i++){
            y[i] += x[j] * m[i][j];
        }
    }
}
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+1] * m[i][j+1];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+2] * m[i][j+2];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

6.6. Раскрутка циклов

6.6.1 Что это такое.

◇ Простой пример. Но можно

◇ После раскрутки вне

```
mod_j = nj % 4;
if(mod_j >= 1){
    for(j = 1; j <= mod_j; j++){
        for(i = 1; i < ni; i++){
            y[i] += x[j] * m[i][j];
        }
    }
}
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+1] * m[i][j+1];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+2] * m[i][j+2];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

```
for(j = 1; j < nj; j++){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    }
}
```

6.6. Раскрытие циклов

6.6.2 Слияние циклов.

- ◇ Слияние циклов – объединение двух циклов с одинаковыми границами изменения индекса и шагом в один.
Слияние корректно, когда каждое определение и каждое использование в результирующем (слитом) цикле имеют такие же значения, что и в исходных (сливаемых) циклах.
- ◇ В рассматриваемом простом примере можно выполнить три слияния циклов.

◇ До слияния внутренних циклов:

```
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+1] * m[i][j+1];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+2] * m[i][j+2];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+3] * m[i][j+3];
    };
}
```


6.6. Раскрытие циклов

6.6.2 Слияние циклов.

- ◇ Слияние циклов – объединение двух циклов с одинаковыми границами изменения индекса и шагом в один.
- ◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ После первого слияния:

```
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
        y[i] += x[j+1] * m[i][j+1];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+2] * m[i][j+2];
    };
    for(i = 1; i < ni; i++){
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

6.6. Раскрытие циклов

6.6.2 Слияние циклов.

◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ После третьего слияния:

```
for(j = mod_j + 1; j <= nj; j += 4) {
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
        y[i] += x[j+1] * m[i][j+1];
        y[i] += x[j+2] * m[i][j+2];
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

◇ Последний цикл можно преобразовать к виду:

```
for(j = mod_j + 1; j <= nj; j += 4) {
    for(i = 1; i < ni; i++){
        y[i] = y[i] + x[j] * m[i][j] + x[j+1] * m[i][j+1]
            + x[j+2] * m[i][j+2] + x[j+3] * m[i][j+3];
    };
}
```

6.6. Раскрытие циклов

6.6.2 Слияние циклов.

◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ Таким образом, оптимизируя цикл

```
mod_j = nj % 4;
if(mod_j >= 1){
    for(j = 1; j <= mod_j; j++){
        for(i = 1; i < ni; i++){
            y[i] += x[j] * m[i][j];
        }
    }
},
```

получаем окончательно следующий цикл:

```
for(j = mod_j + 1; j <= nj; j += 4){
    for(i = 1; i < ni; i++){
        y[i] += x[j] * m[i][j];
        y[i] += x[j+1] * m[i][j+1];
        y[i] += x[j+2] * m[i][j+2];
        y[i] += x[j+3] * m[i][j+3];
    };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

◇ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◇ Таким образом, оптимизируя цикл

```
mod_j = nj % 4;  
if(mod_j >= 1){  
    for(i = 1; i <= mod_j - i; i++) {
```

Рассмотренное оптимизирующее преобразование называется *раскруткой с последующим сжатием*.

```
    }  
}  
},
```

Если у сливаемых циклов границы изменения переменной цикла i не совпадают, можно применить выравнивание циклов.

```
for(i = 1; i < ni; i++){  
    y[i] += x[j] * m[i][j];  
    y[i] += x[j+1] * m[i][j+1];  
    y[i] += x[j+2] * m[i][j+2];  
    y[i] += x[j+3] * m[i][j+3];  
};  
}
```

6.6. Раскрутка циклов

6.6.3 Сравнение двух видов раскрутки циклов.

- ◇ Раскрутка внутреннего цикла производит код, **выполняющий намного меньше проверок на выход из цикла**. Доступ к двумерному массиву $m[i][j]$ последователен, так как С-массив располагается по строкам.
- ◇ Раскрутка внешнего цикла не только сокращает количество проверок на выход из цикла, но и (особенно в случае раскрутки с последующим сжатием) обеспечивает повторное использование $y[i]$, а также последовательный доступ как к элементам x , так и к элементам m . Увеличение повторного использования данных существенно изменяет соотношение между арифметическими операциями и операциями доступа к памяти в цикле, повышая *локальность* данных.
- ◇ Кроме того, при каждом подходе могут проявляться и другие прямые и косвенные улучшения кода. Окончательная производительность цикла зависит от всех улучшений, как прямых, так и косвенных.
- ◇ Важным косвенным улучшением помимо увеличения локальности данных является увеличение количества операций в теле цикла.
- ◇ Раскрутка цикла позволяет реализовать его параллельное выполнение (этот вопрос будет рассмотрен при планировании кода).

6.7. Заключительные замечания

6.7.1 Другие преобразования циклов.

- ◇ В компиляторах применяются и другие преобразования циклов. Но эти преобразования применяются во время планирования кода для обеспечения параллельного выполнения инструкций программы на каждом ядре процессора (эти преобразования будут рассмотрены в последующих лекциях).
- ◇ Есть несколько преобразований циклов, используемые для повышения степени локальности данных, что позволяет увеличить производительность кэша данных при выполнении программы. Обеспечение локальности данных необходимо и для распараллеливания программы.