8. Оптимизация циклов

Циклы в графах потоков управления

- Циклы составляют большую часть времени выполнения программы
- Оптимизация повышающая производительность циклов может оказать существенное влияние на производительность программы в целом
- Циклы влияют на время работы анализа программы.
  Например, анализ потоков данных для программы без циклов может быть выполнен за один проход узлов графа потока управления в топологическом порядке
- Для обнаружения циклов необходимы следующие концепции: упорядочение узлов CFG вглубь, доминаторы и деревья доминаторов, обратные ребра
- Для анализа сходимости итеративных алгоритмов, а также для проведения некоторых преобразований может использоваться свойство сводимости графа потока управления

Упорядочение узлов CFG в глубину

- *Поиск в графе в глубину* однократно посещает все узлы графа, начиная с входного узла и посещая, в первую очередь, максимально удаленные от входного
- Путь поиска в глубину образует *глубинное остовное дерево* (охватывающее вглубь дерево — depth-first spanning tree — DFST)
- *Упорядочение в глубину* (depth-first ordering) представляет собой обращение обратного порядка обход, иначе говоря, при упорядочении в глубину посещается узел, затем его крайний справа узел-преемник, после этого — узел, расположенный слева от него, и т.д.
- Перед тем как строить дерево для графа потока, следует выбрать, какой из преемников является крайним справа, какой — его левым соседом и т. д.

Выделение естественных циклов. Классификация дуг ГПУ

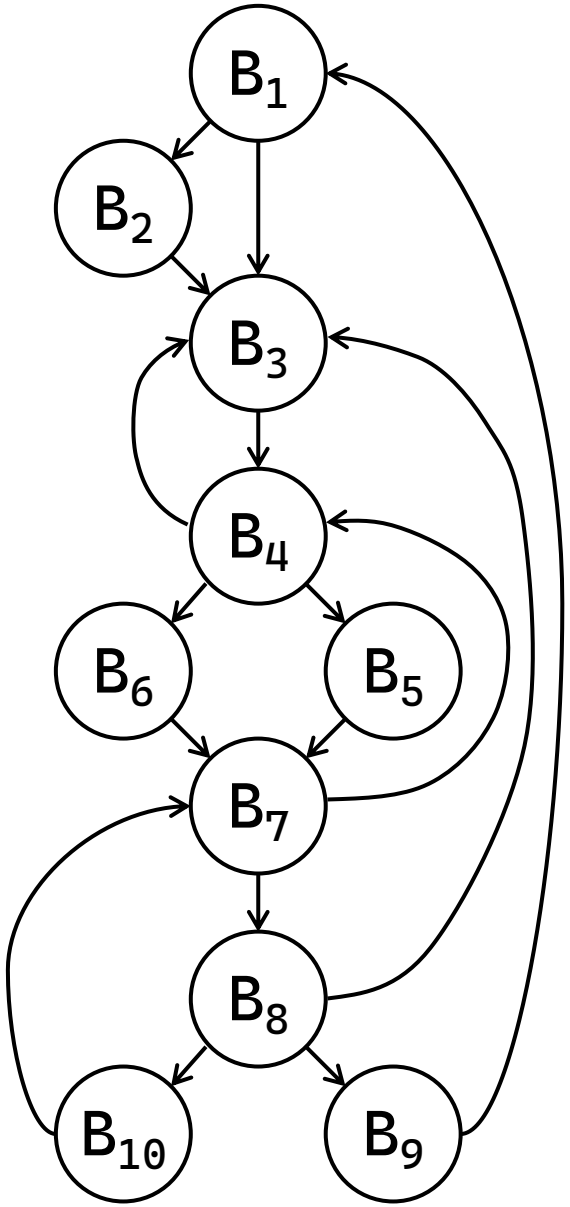
- Дуги ГПУ, являющиеся дугами и его остовного дерева, называются *наступающими* (*остовными*, advancing)
- Дуги ГПУ, не являющиеся дугами его остовного дерева, но имеющие такое же направление, что и остовные, называются *прямыми*
- Дуги ГПУ, направленные противоположно остовным, называются *отступающими* (*обратно направленными*, retreating)
  * Идут от узла m к предку m в дереве
- Отступающая дуга ГПУ $\langle B_i, B_k \rangle$ называется *обратной*, если $B_k = Dom(B_i)$
- Остальные дуги ГПУ называются *поперечными* (cross) относительно глубинного остовного дерева

* $m \to n -$ отступающее ребро, если $dfn[m] \geq dfn[n]$

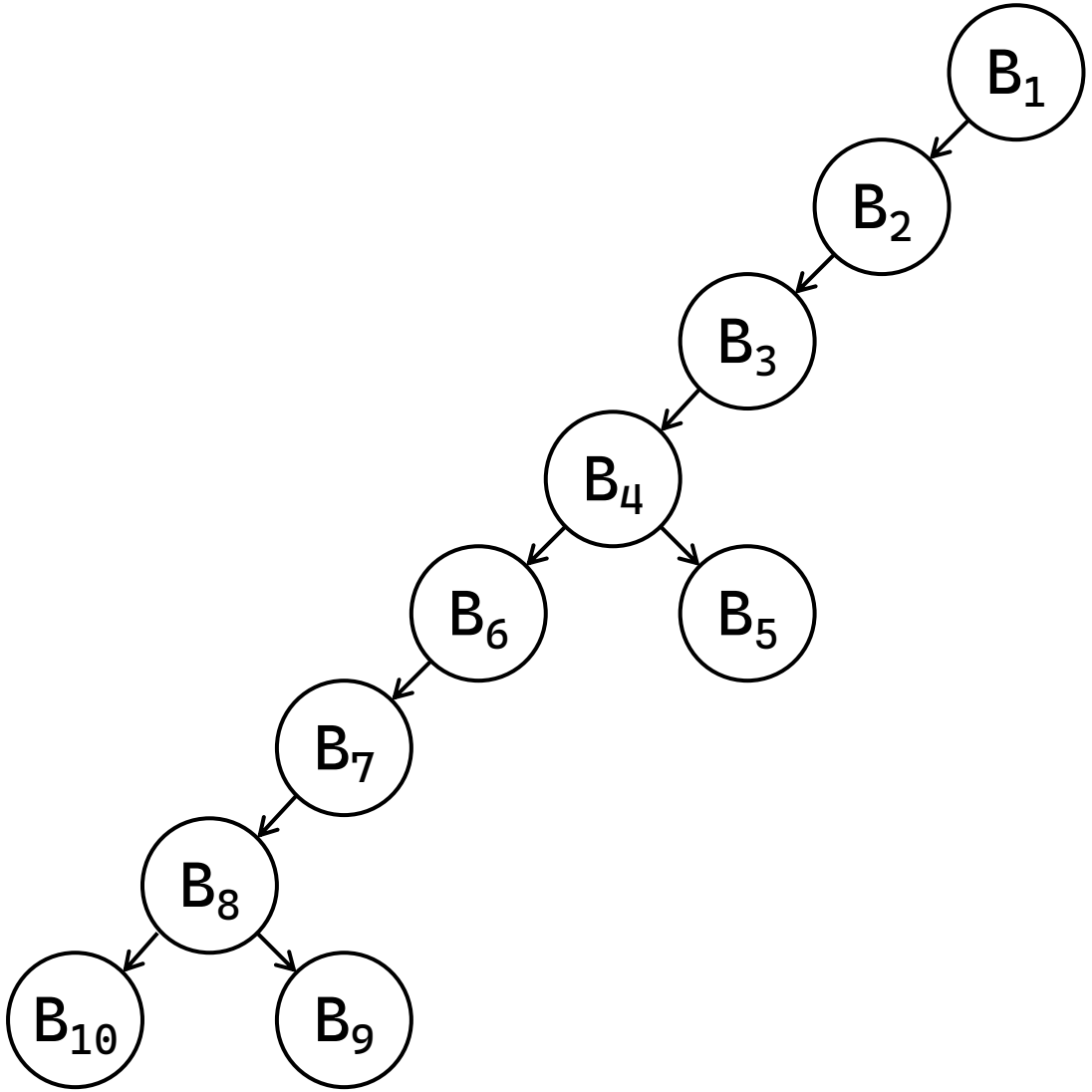
Выделение естественных циклов. Классификация дуг ГПУ

- ***Поперечные*** (*cross*) *ребра*. Ребра $m \to n$ такое, что ни m, ни n не являются предками друг друга в *глубинном остовном дереве*
- Важным свойством поперечных ребер является то, что если изобразить DFST так, чтобы дочерние узлы некоторого узла располагались слева направо в порядке, в котором они добавлялись в дерево, то все поперечные ребра будут идти справа налево

Выделение естественных циклов. Классификация дуг ГПУ

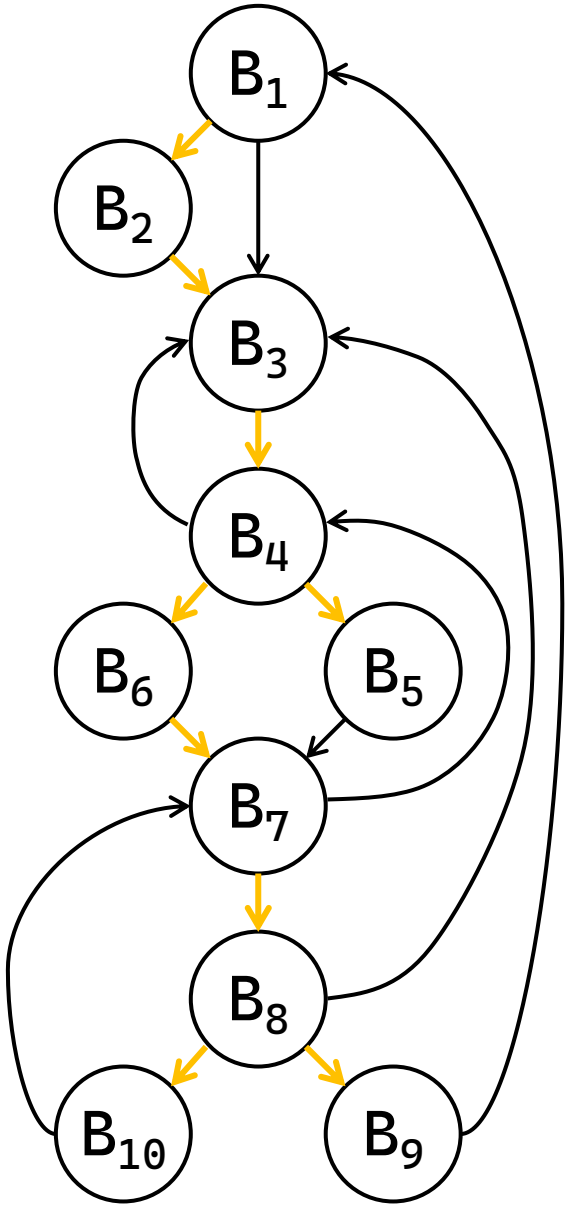


Граф потока управления



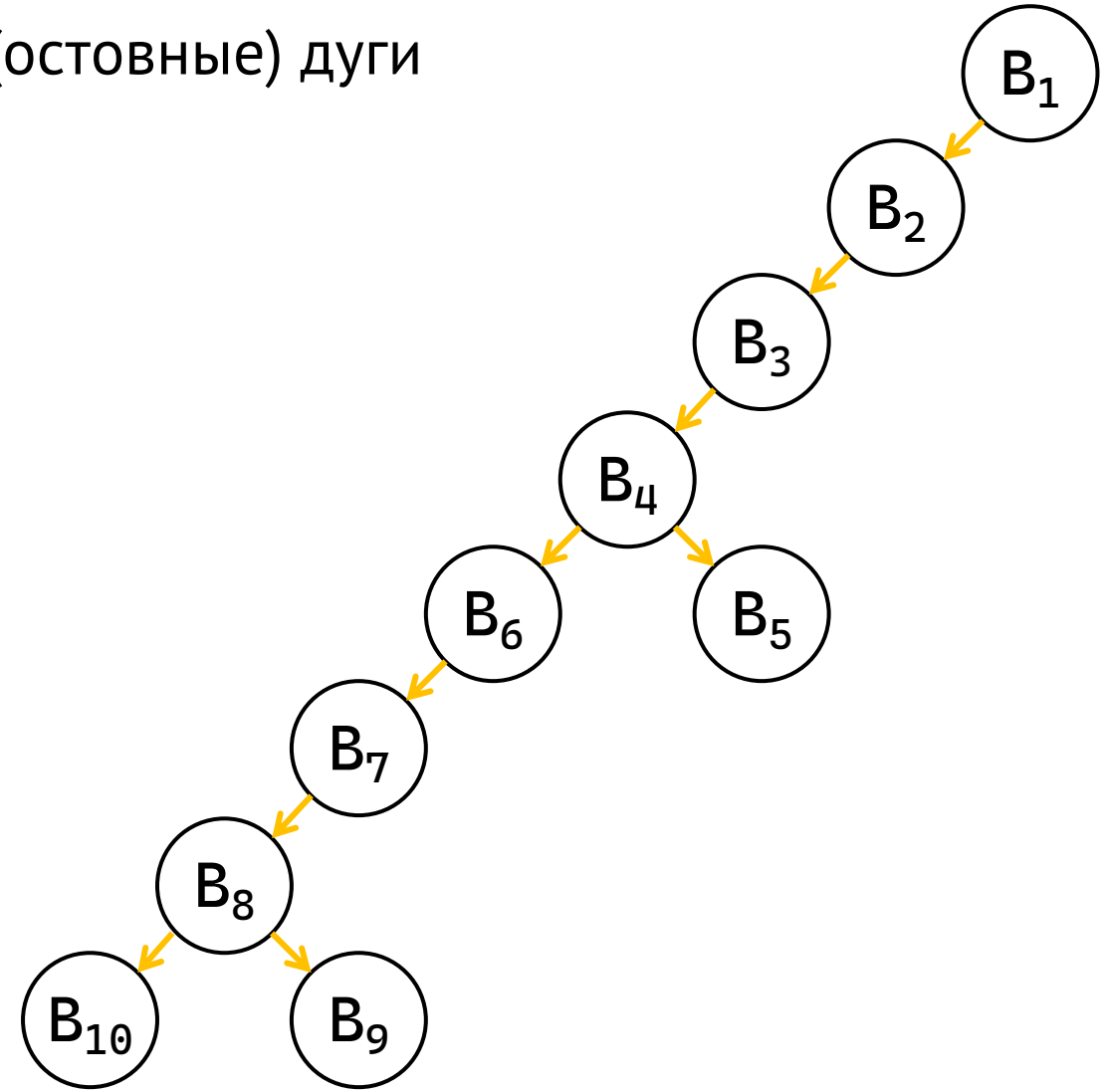
Глубинное остовное дерево

Выделение естественных циклов. Классификация дуг ГПУ



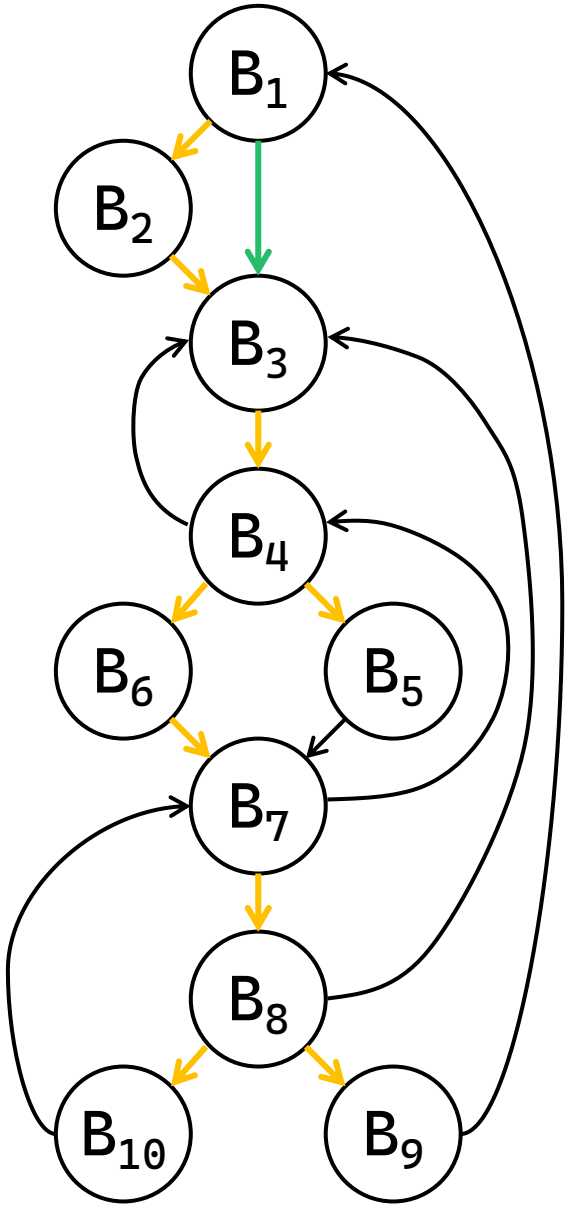
Граф потока управления

Наступающие (остовные) дуги



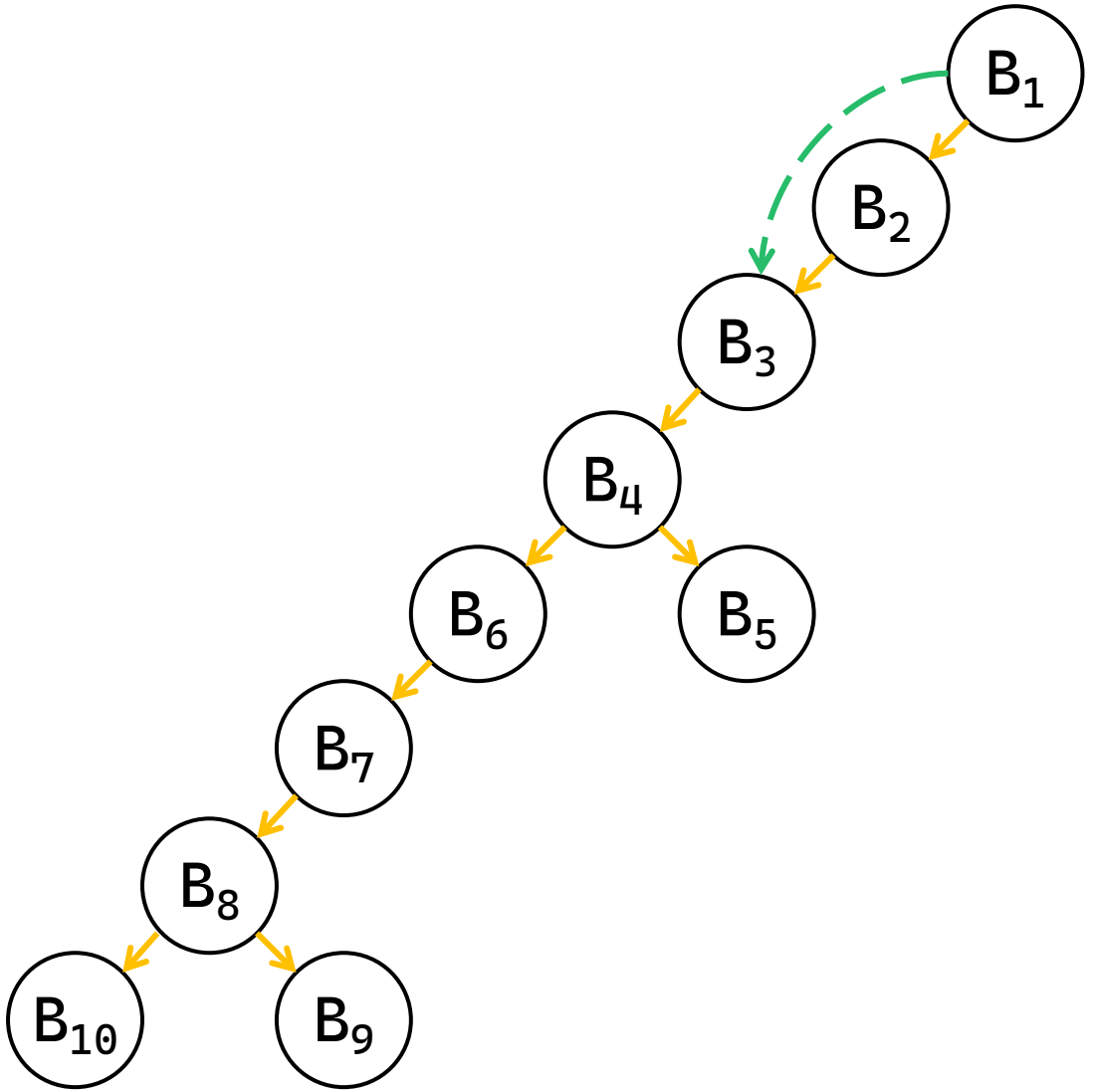
Глубинное остовное дерево

Выделение естественных циклов. Классификация дуг ГПУ



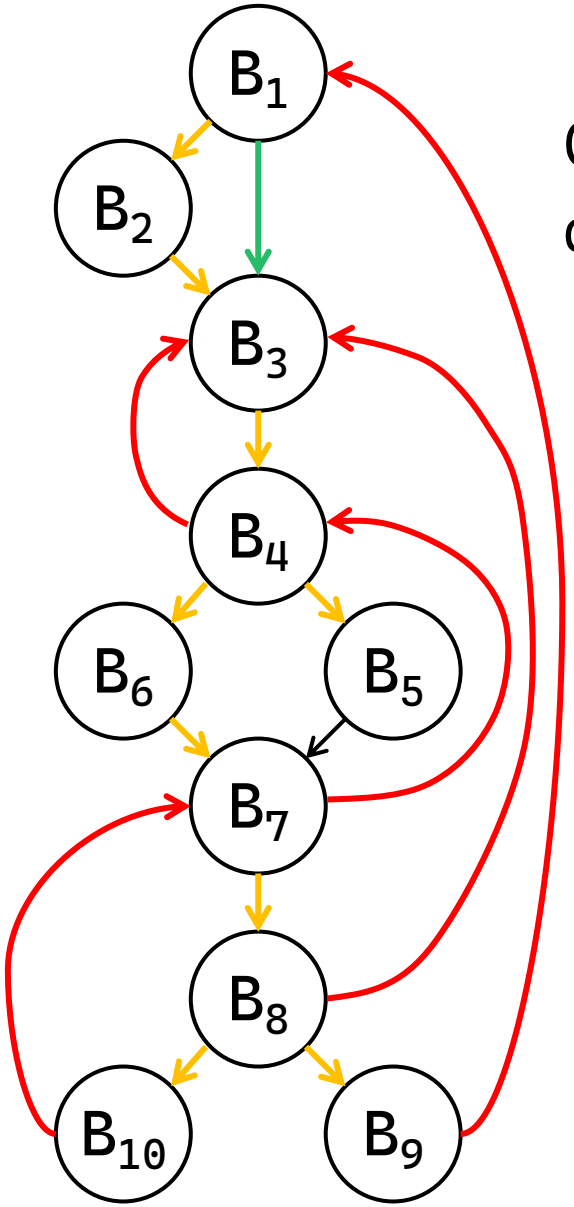
Граф потока управления

Прямые дуги



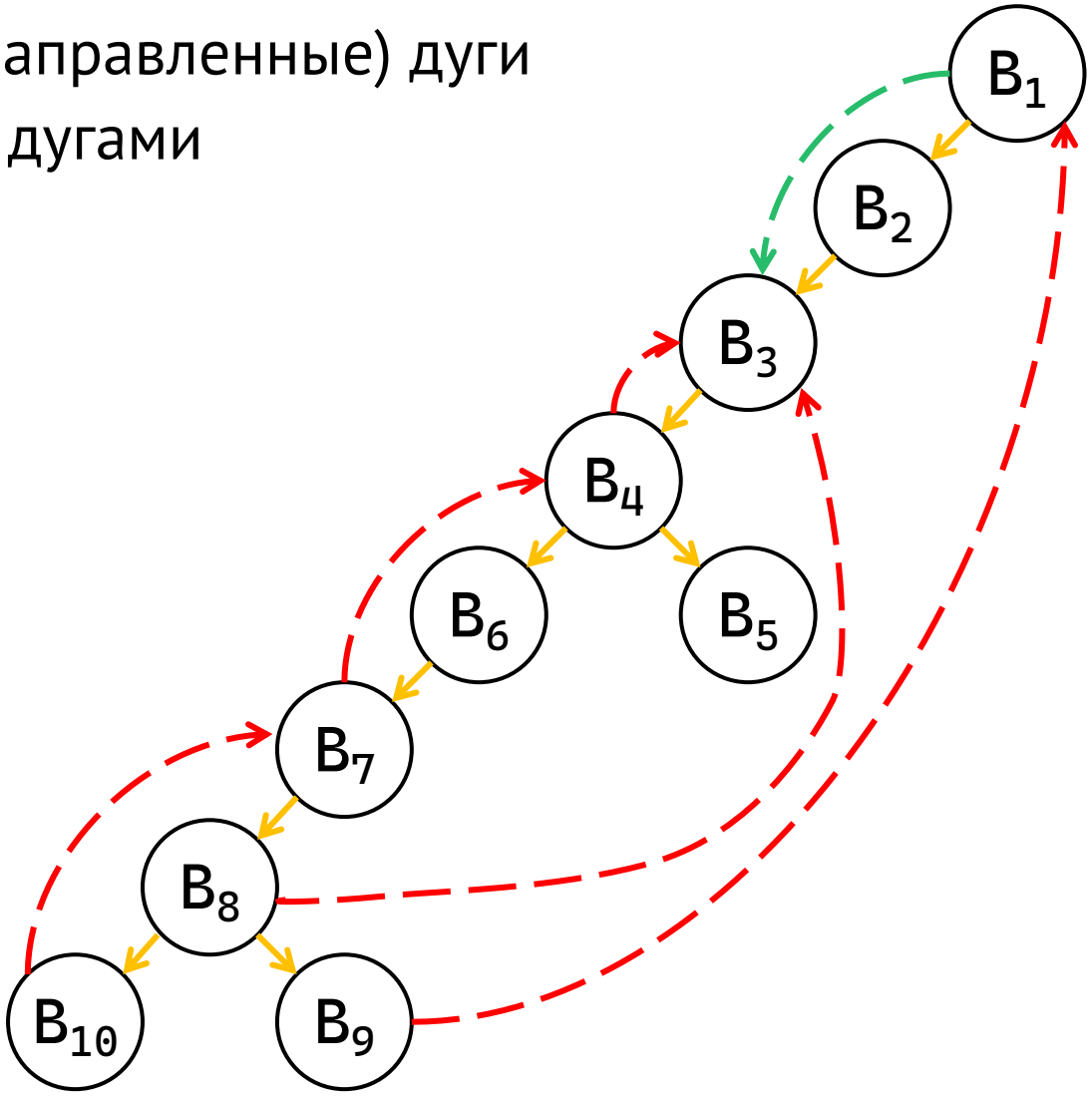
Глубинное остовное дерево

Выделение естественных циклов. Классификация дуг ГПУ



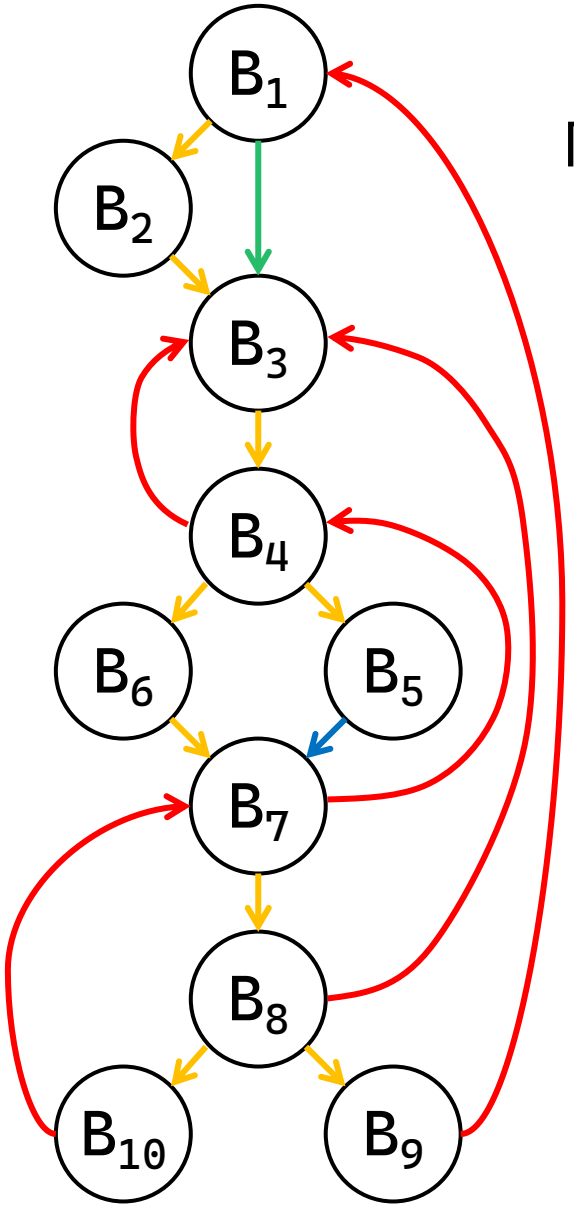
Граф потока управления

Отступающие (обратно направленные) дуги совпадают с *обратными* дугами

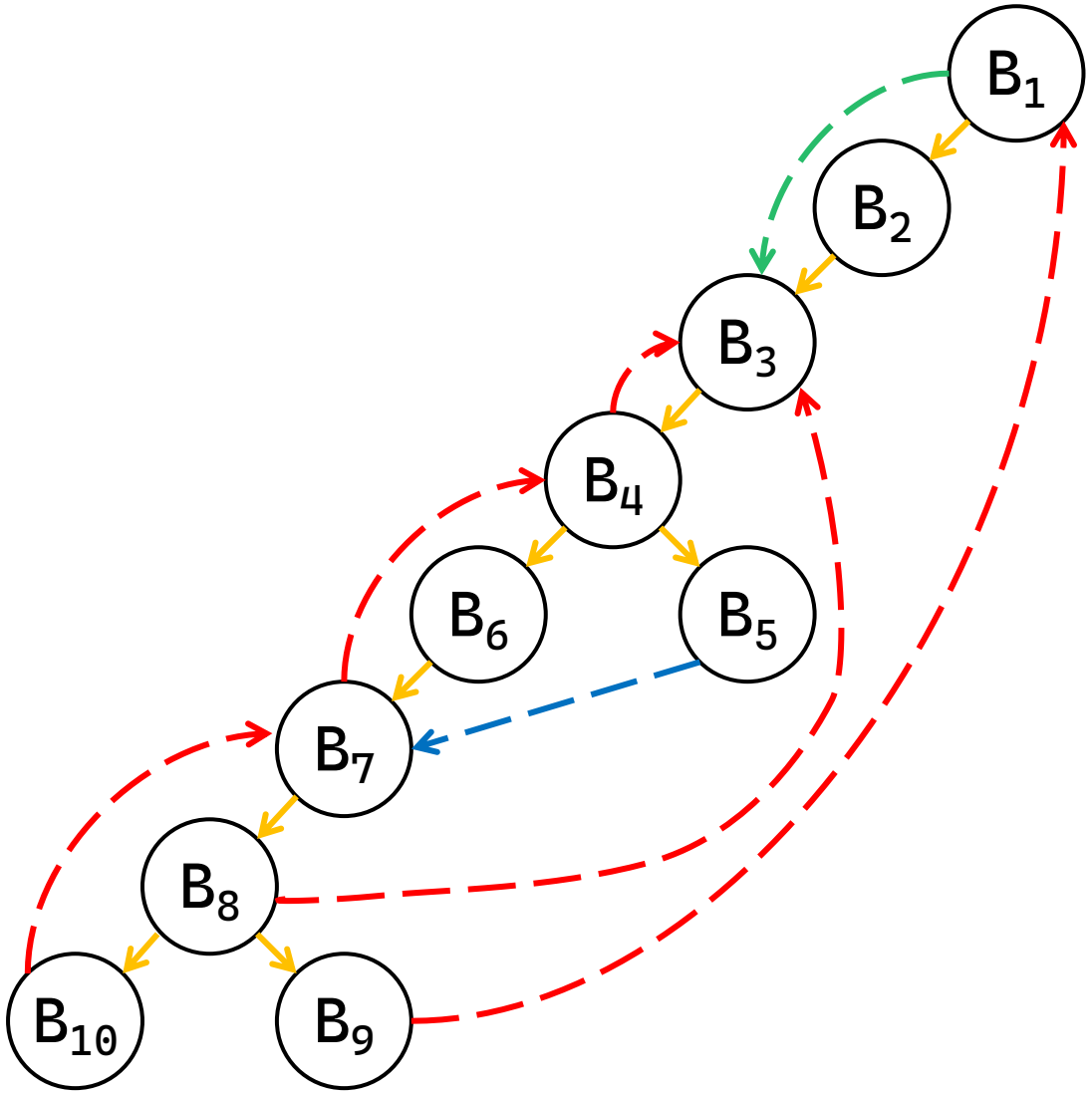


Глубинное остовное дерево

Выделение естественных циклов. Классификация дуг ГПУ



Поперечные дуги



Граф потока управления

Глубинное остовное дерево

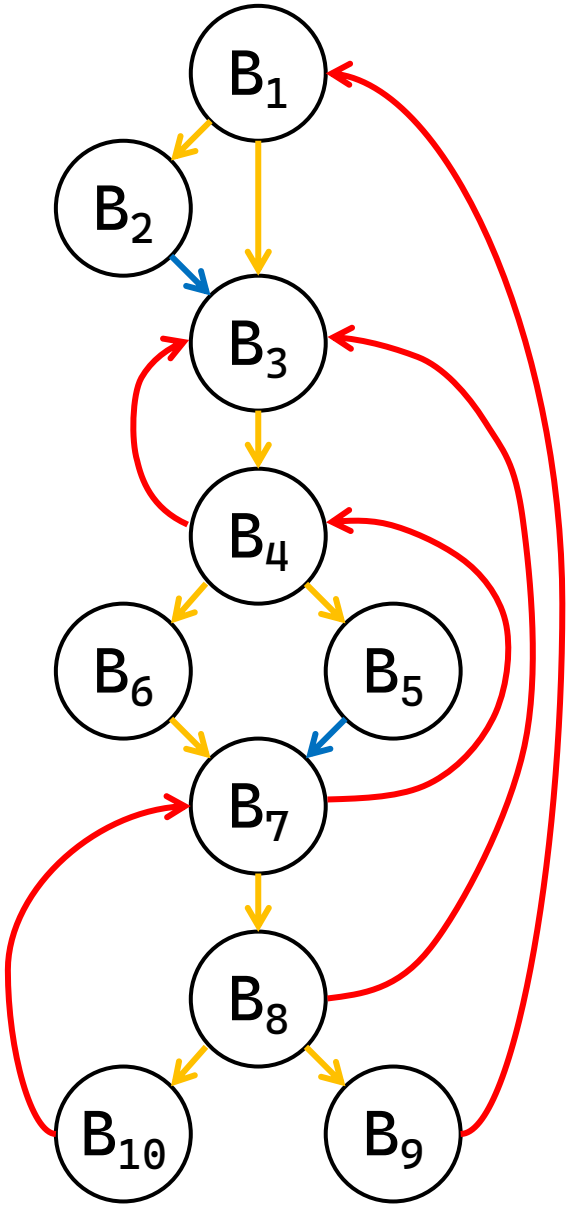
Выделение естественных циклов. Определение естественного цикла

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

Выделение естественных циклов. Определение естественного цикла

- Для любого графа потока каждое *обратное* ребро является *отступающим*, но не всякое *отступающее* ребро является *обратным*
- **Определение**. Граф потока называется *сводимым* (*reducible*), если все его отступающие ребра в любом DFST являются обратными
- Если граф сводимый, то все его DFST имеют одно и то же множество *отступающих ребер*, в точности совпадающее со множеством обратных ребер
- Если граф *не сводимый*, то все обратные ребра в любом DFST являются отступающими, но каждое DFST имеет дополнительные отступающие ребра, **не являющиеся** обратными

Выделение естественных циклов. Естественные циклы. Пример



Граф потока управления

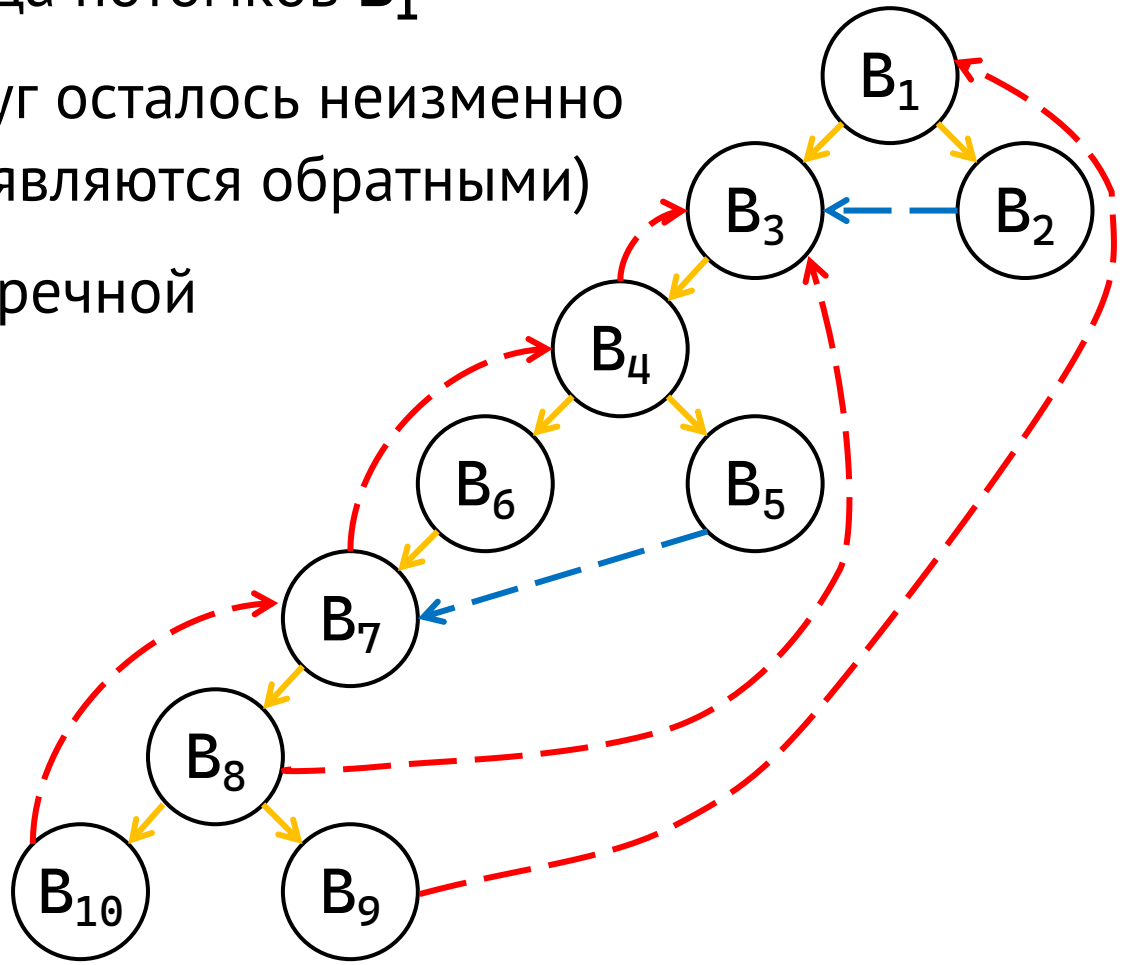
Построим другое остовное дерево

Изменим порядок обхода потомков **B_1**

Множество обратных дуг осталось неизменно (все отступающие дуги являются обратными)

Дуга **B_2–>B_3** стала поперечной

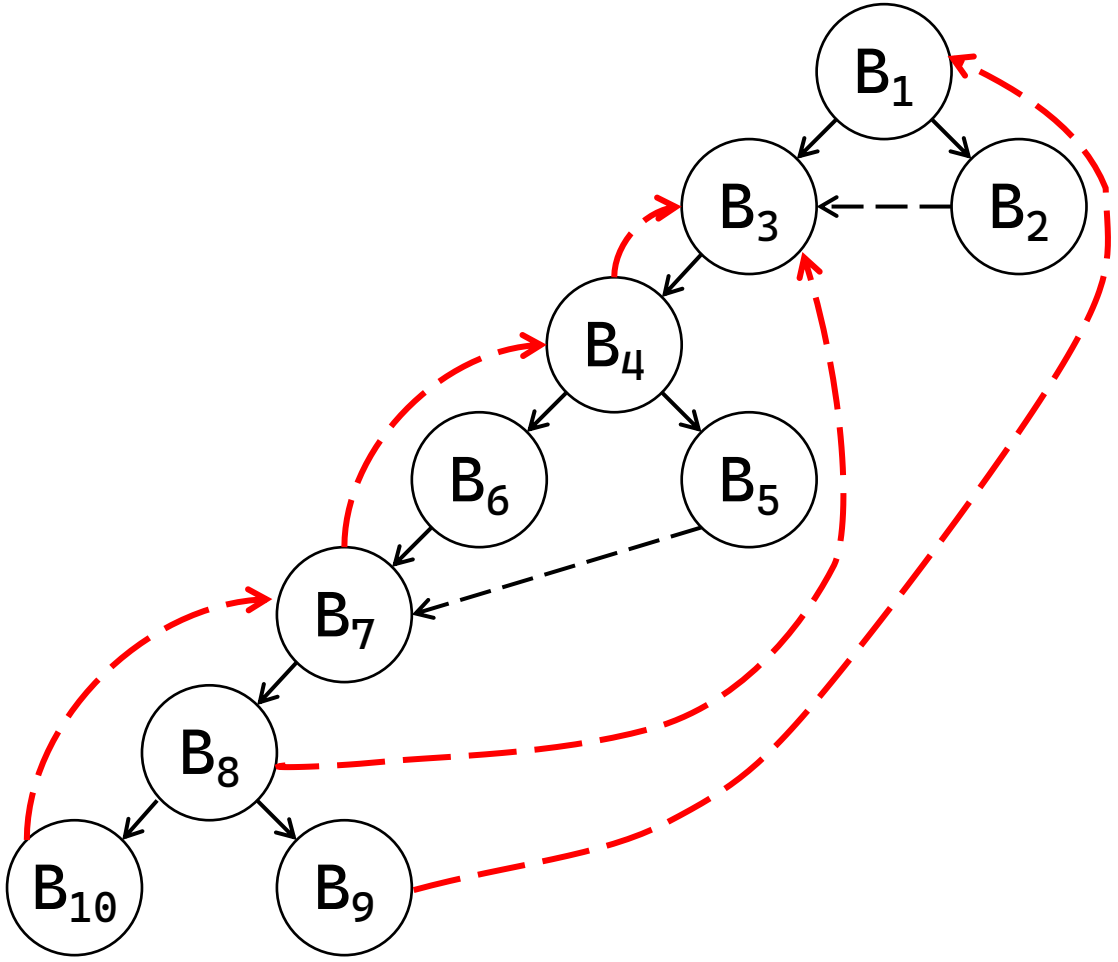
Дуга **B_1–>B_3** стала прямой наступающей



Глубинное остовное дерево

Выделение естественных циклов. Естественные циклы. Пример

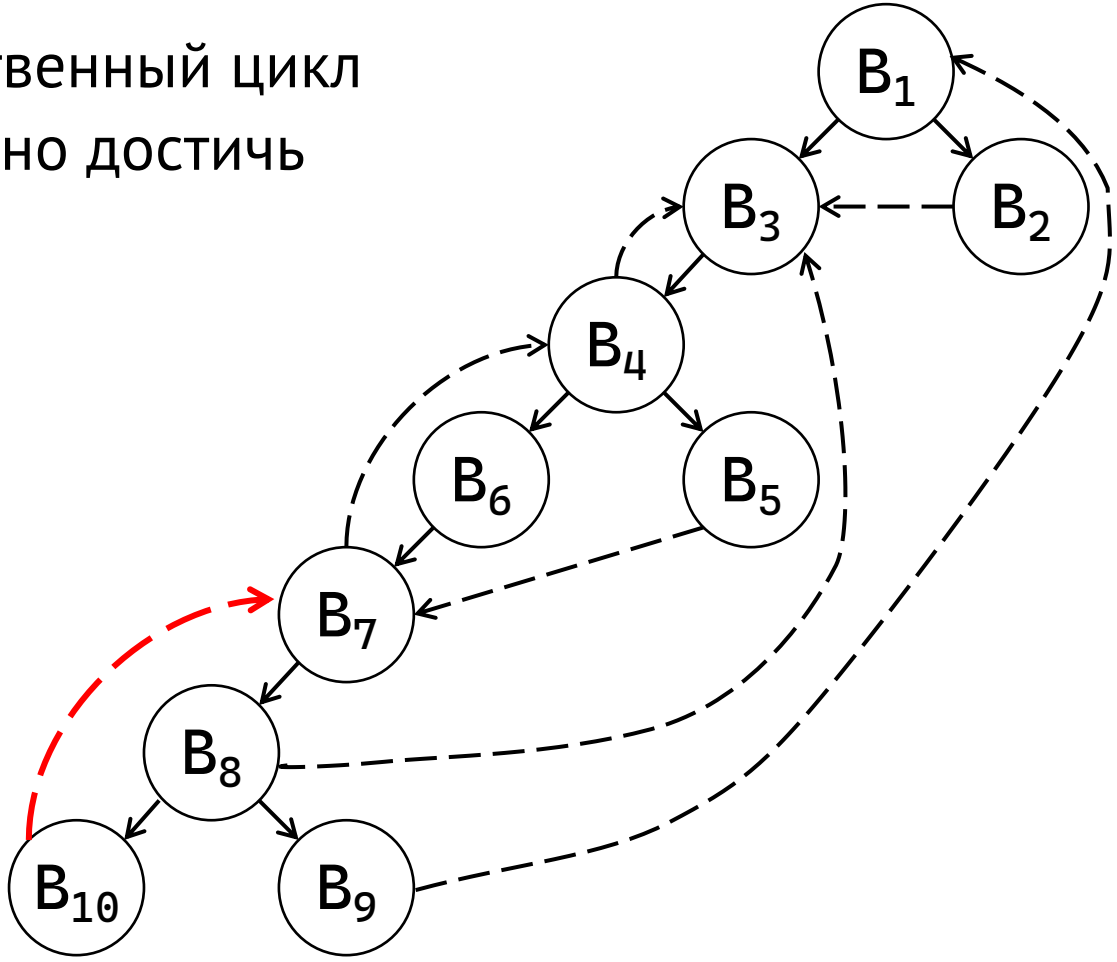
- На рисунке справа – пять обратных дуг:
  $\langle B_4, B_3 \rangle, \langle B_7, B_4 \rangle, \langle B_8, B_3 \rangle, \langle B_9, B_1 \rangle, \langle B_{10}, B_7 \rangle$



Глубинное остовное дерево

Выделение естественных циклов. Естественные циклы. Пример

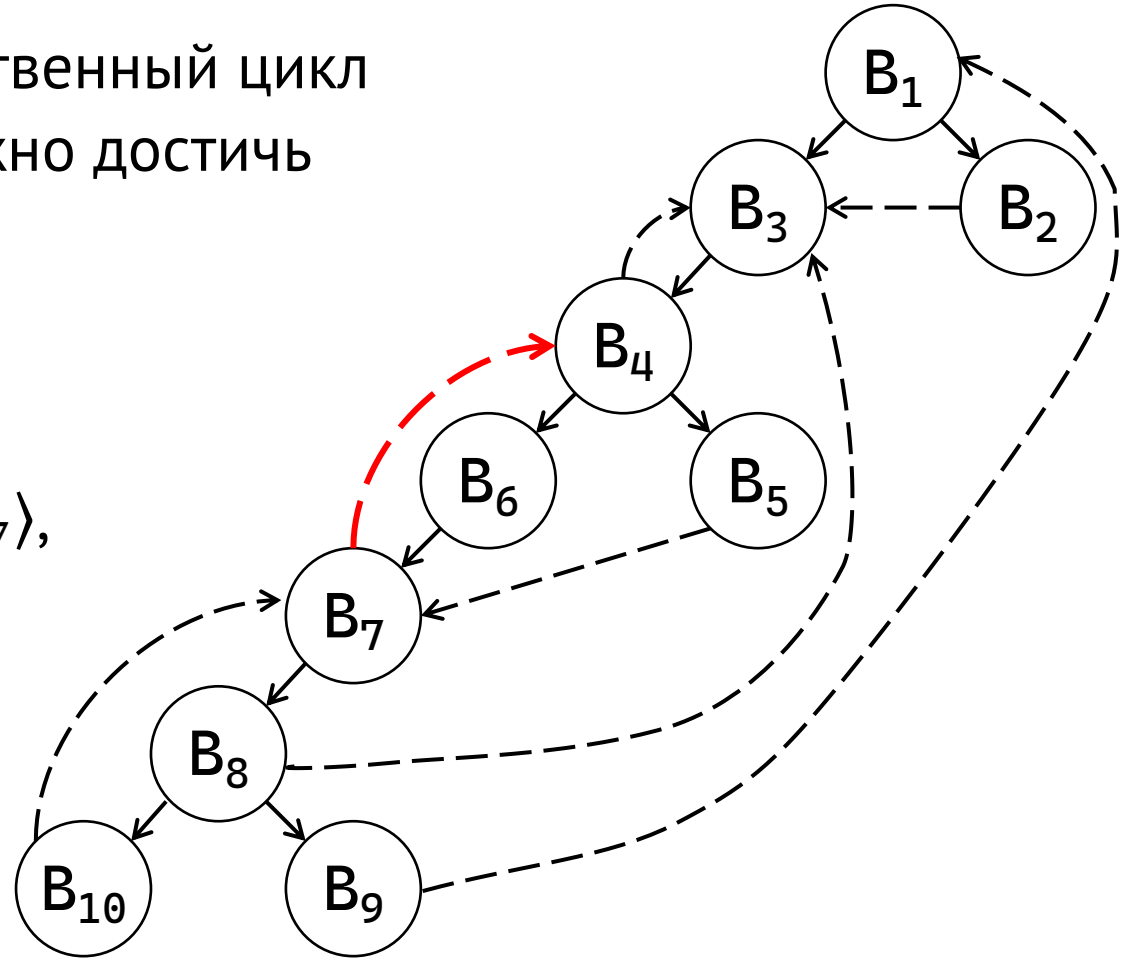
- На рисунке справа – пять обратных дуг:
  $\langle B_4, B_3\rangle, \langle B_7, B_4\rangle, \langle B_8, B_3\rangle, \langle B_9, B_1\rangle, \langle B_{10}, B_7\rangle$
- Обратной дуге $\langle B_{10}, B_7\rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и B_{10}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



Глубинное остовное дерево

Выделение естественных циклов. Естественные циклы. Пример

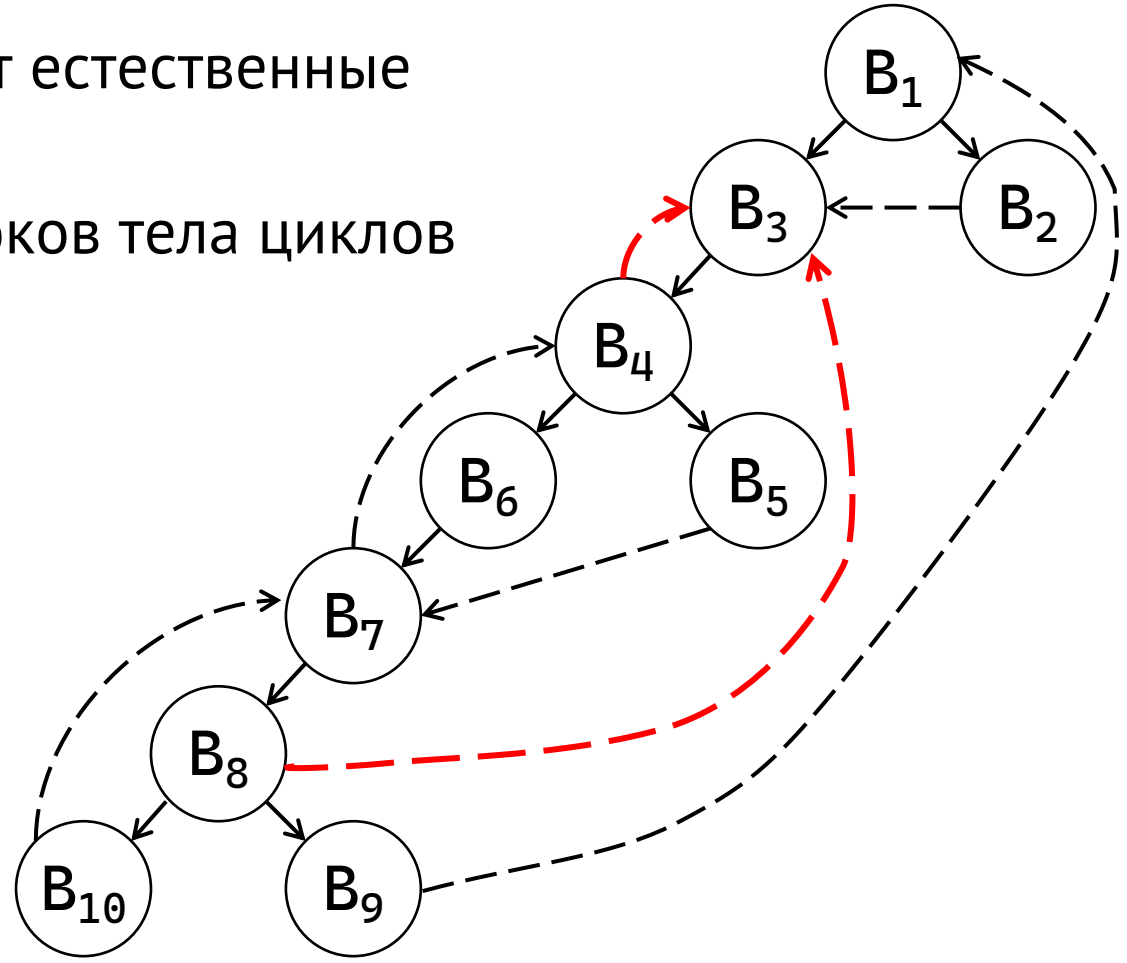
- На рисунке справа – пять обратных дуг:
  $\langle B_4, B_3\rangle, \langle B_7, B_4\rangle, \langle B_8, B_3\rangle, \langle B_9, B_1\rangle, \langle B_{10}, B_7\rangle$
- Обратной дуге $\langle B_{10}, B_7\rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и B_{10}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



Глубинное остовное дерево

Выделение естественных циклов. Естественные циклы. Пример

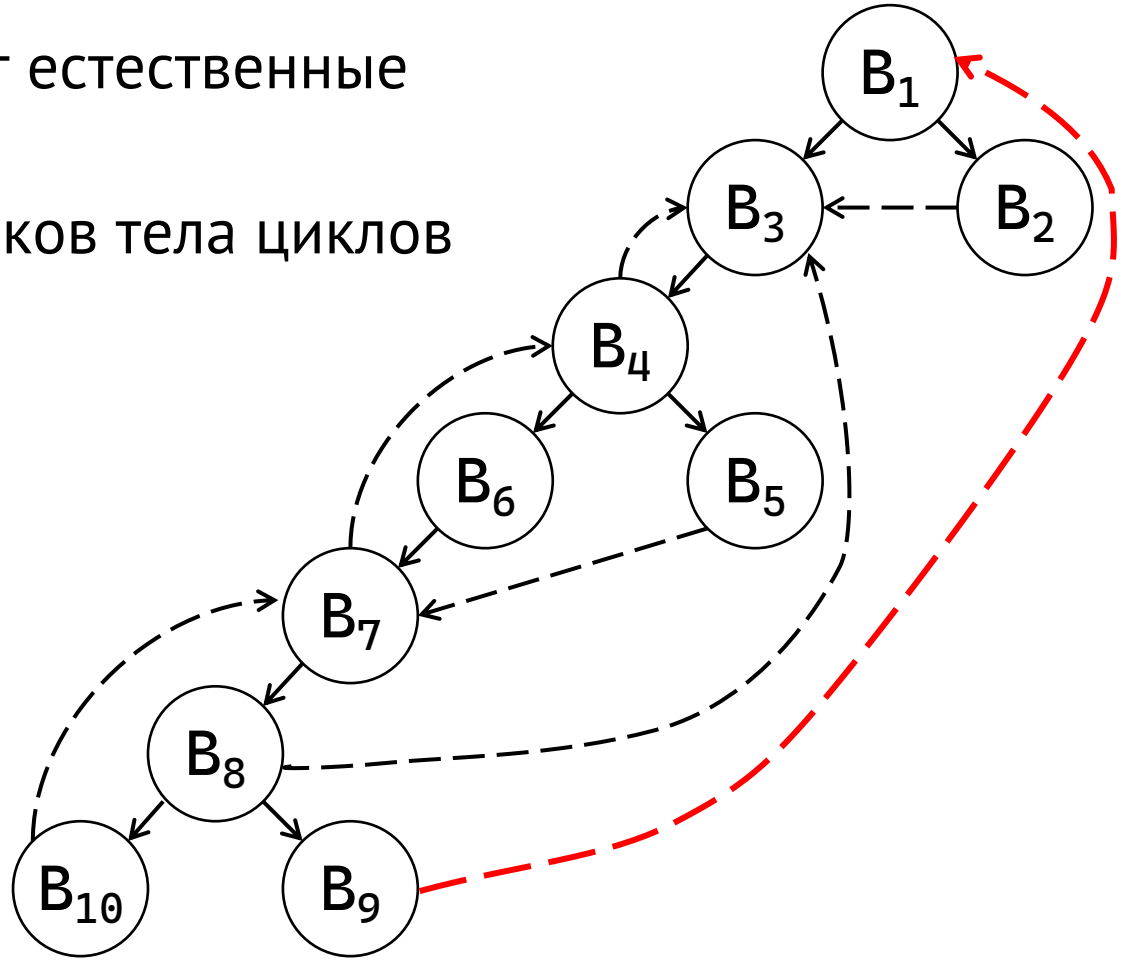
- На рисунке справа – пять обратных дуг:
  $\langle B_4, B_3\rangle, \langle B_7, B_4\rangle, \langle B_8, B_3\rangle, \langle B_9, B_1\rangle, \langle B_{10}, B_7\rangle$
- Обратные дуги $\langle B_4, B_3\rangle$ и $\langle B_8, B_3\rangle$ формируют естественные циклы с одним и тем же заголовком B_3
  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
  $(\langle B_{10}, B_7\rangle, \langle B_7, B_4\rangle)$



Глубинное остовное дерево

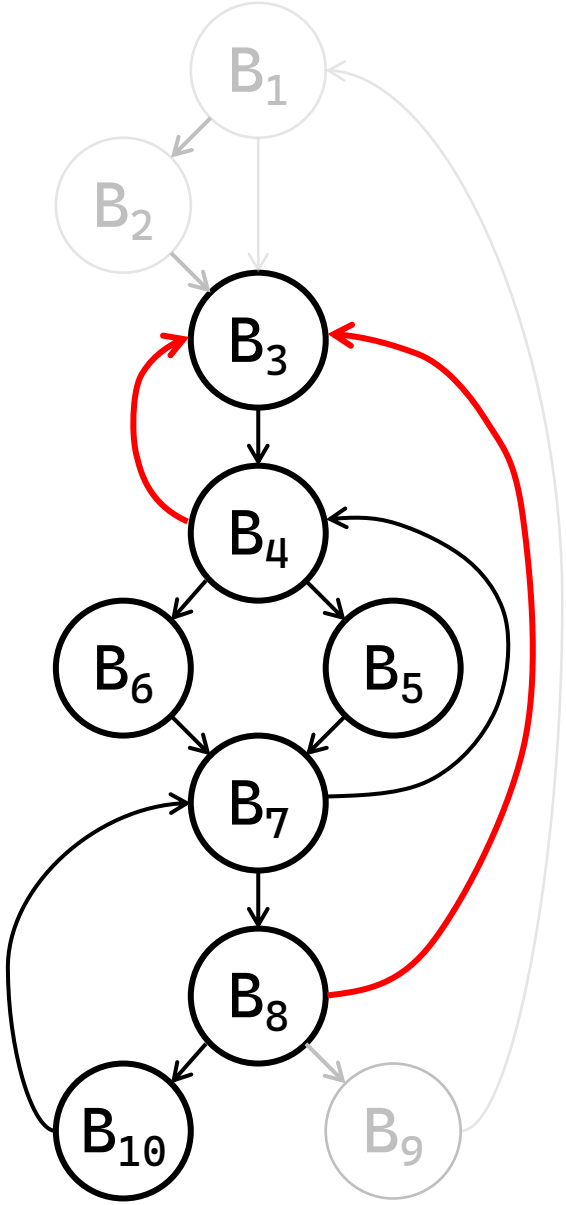
Выделение естественных циклов. Естественные циклы. Пример

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

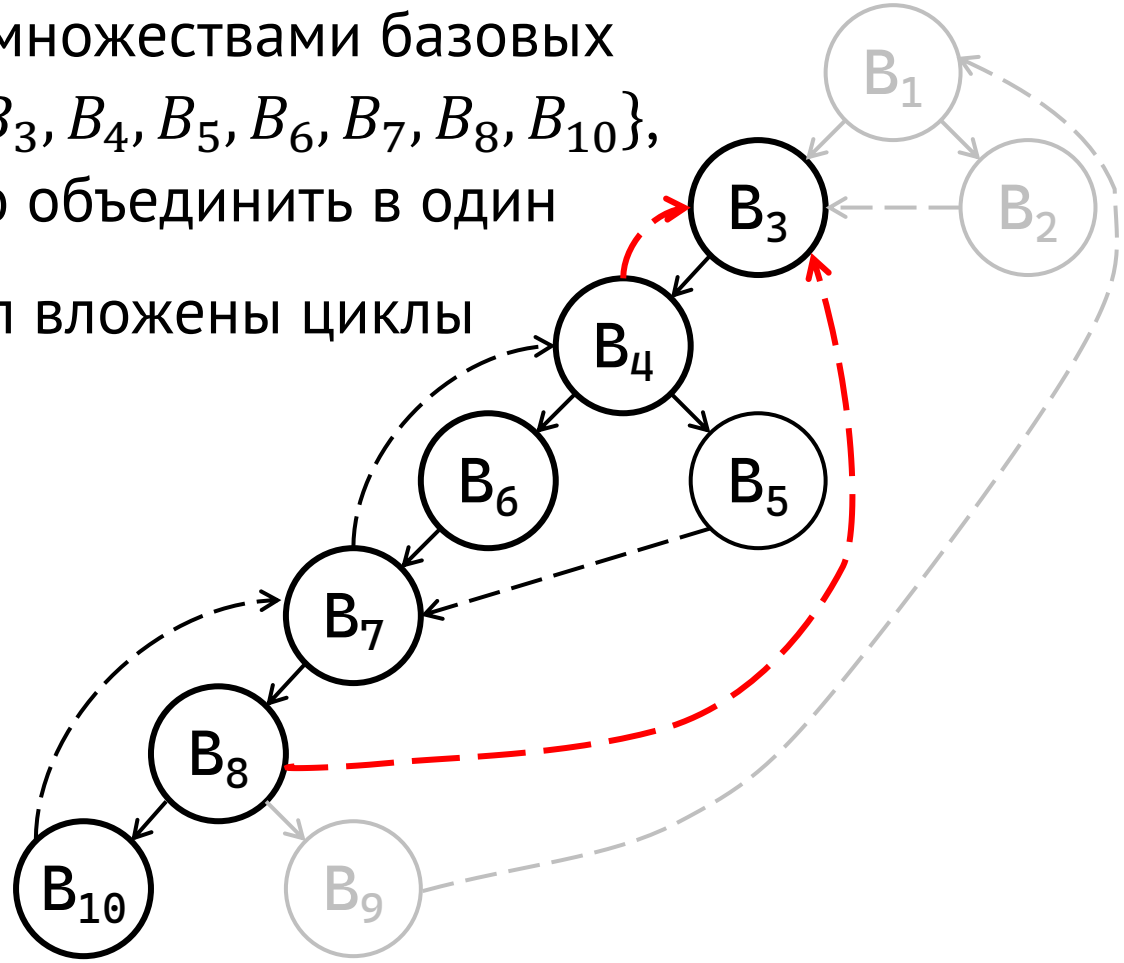


Глубинное остовное дерево

Выделение естественных циклов. 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
- В объединенный цикл вложены циклы $(\langle B_{10}, B_7 \rangle, \langle B_7, B_4 \rangle)$



Граф потока управления

Глубинное остовное дерево

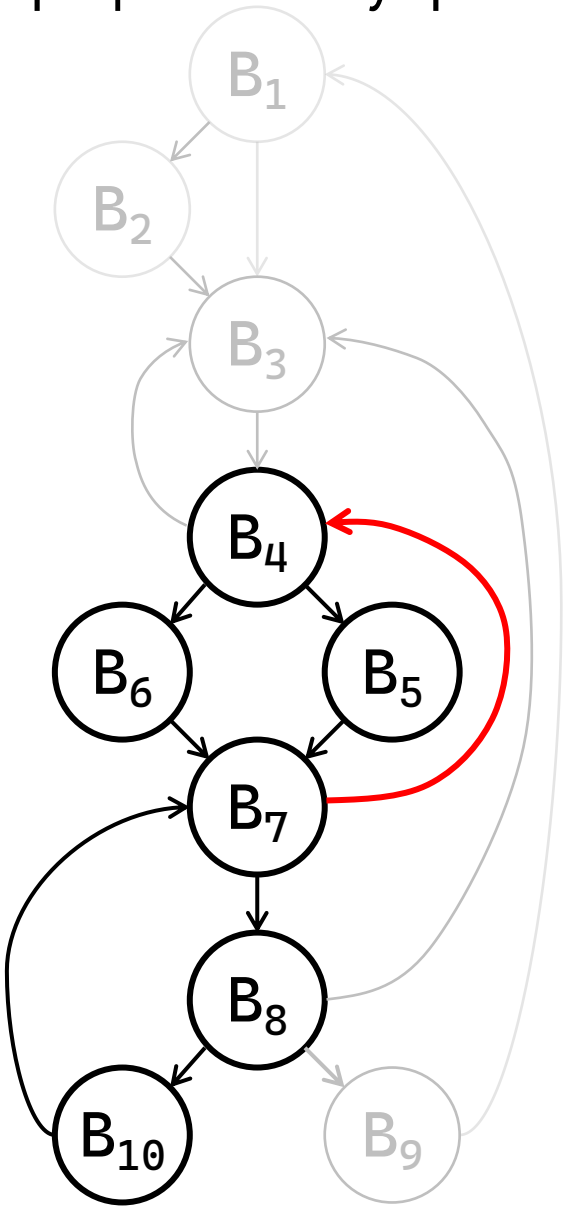
Выделение естественных циклов. Алгоритм построения естественных циклов по обратной дуге

- **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.

Выделение естественных циклов. Пример построения естественного цикла по обратной дуге.

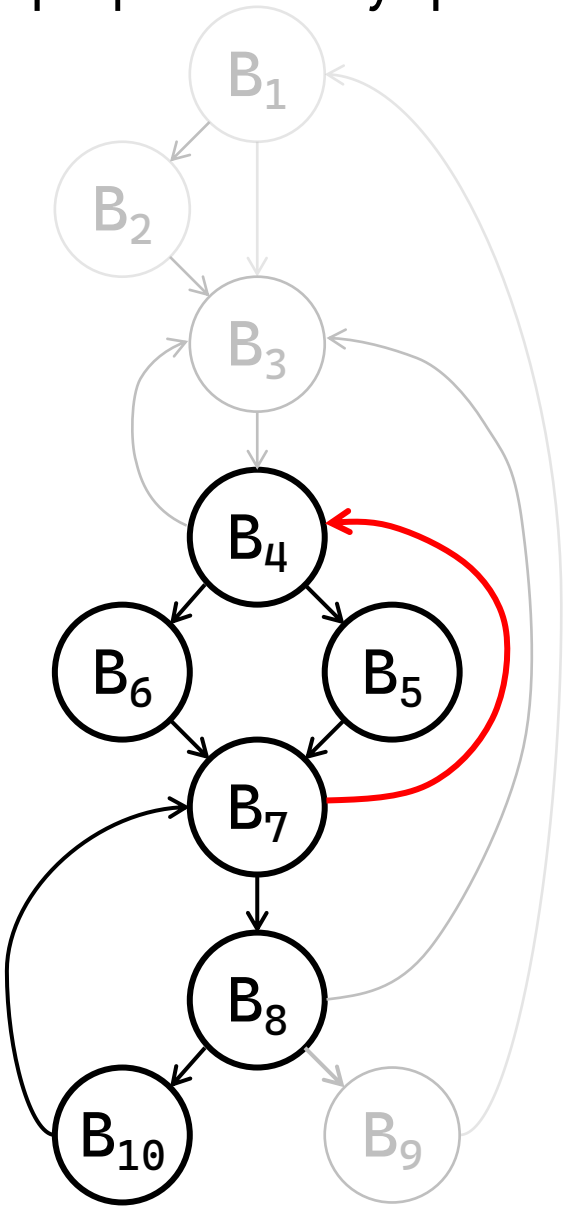
Граф потока управления:



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

Выделение естественных циклов. Пример построения естественного цикла по обратной дуге.

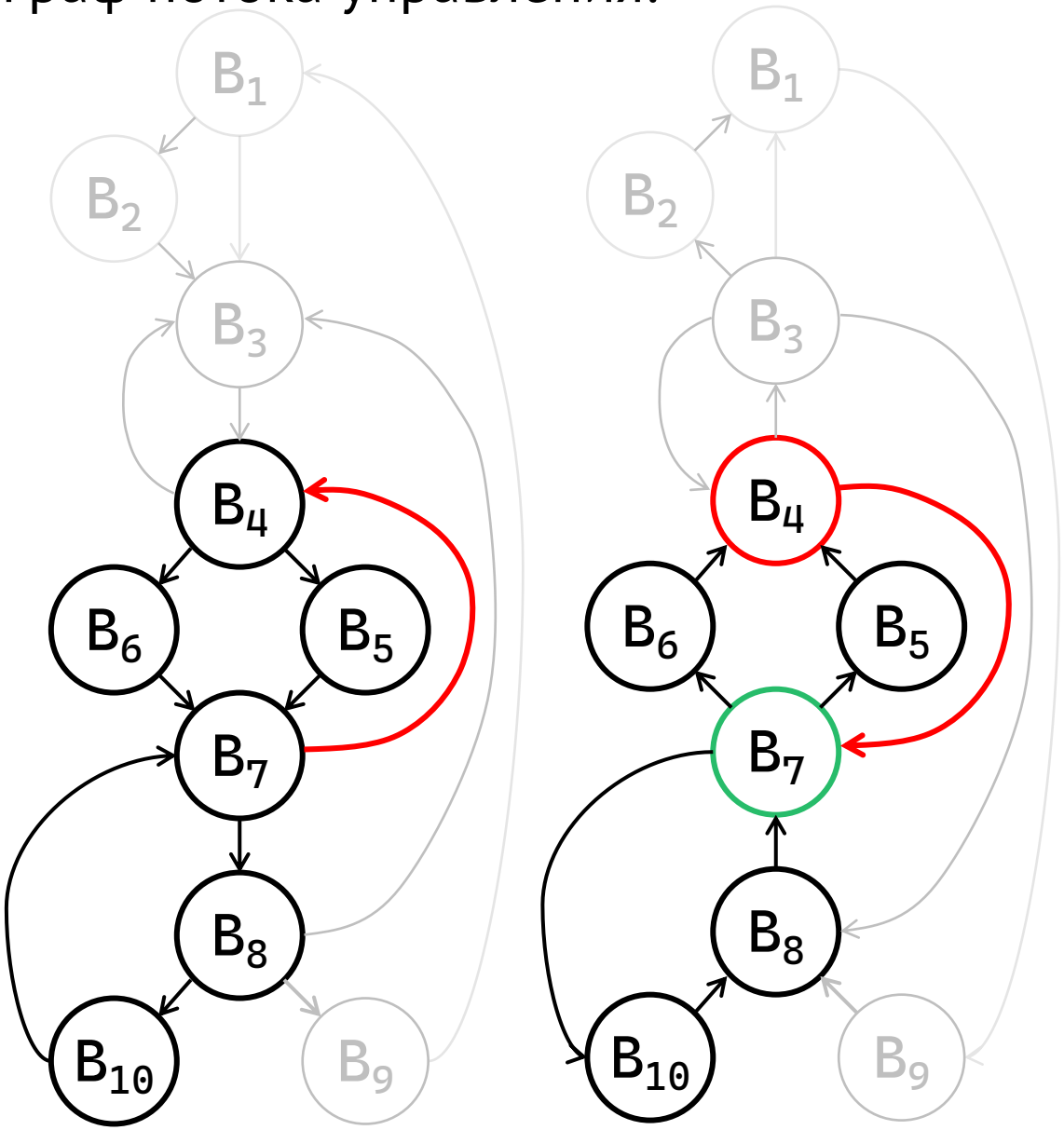
Граф потока управления:



- Применим алгоритм 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 $Pred(B_7) = \{B_5, B_6, B_{10}\}$

Выделение естественных циклов. Пример построения естественного цикла по обратной дуге.

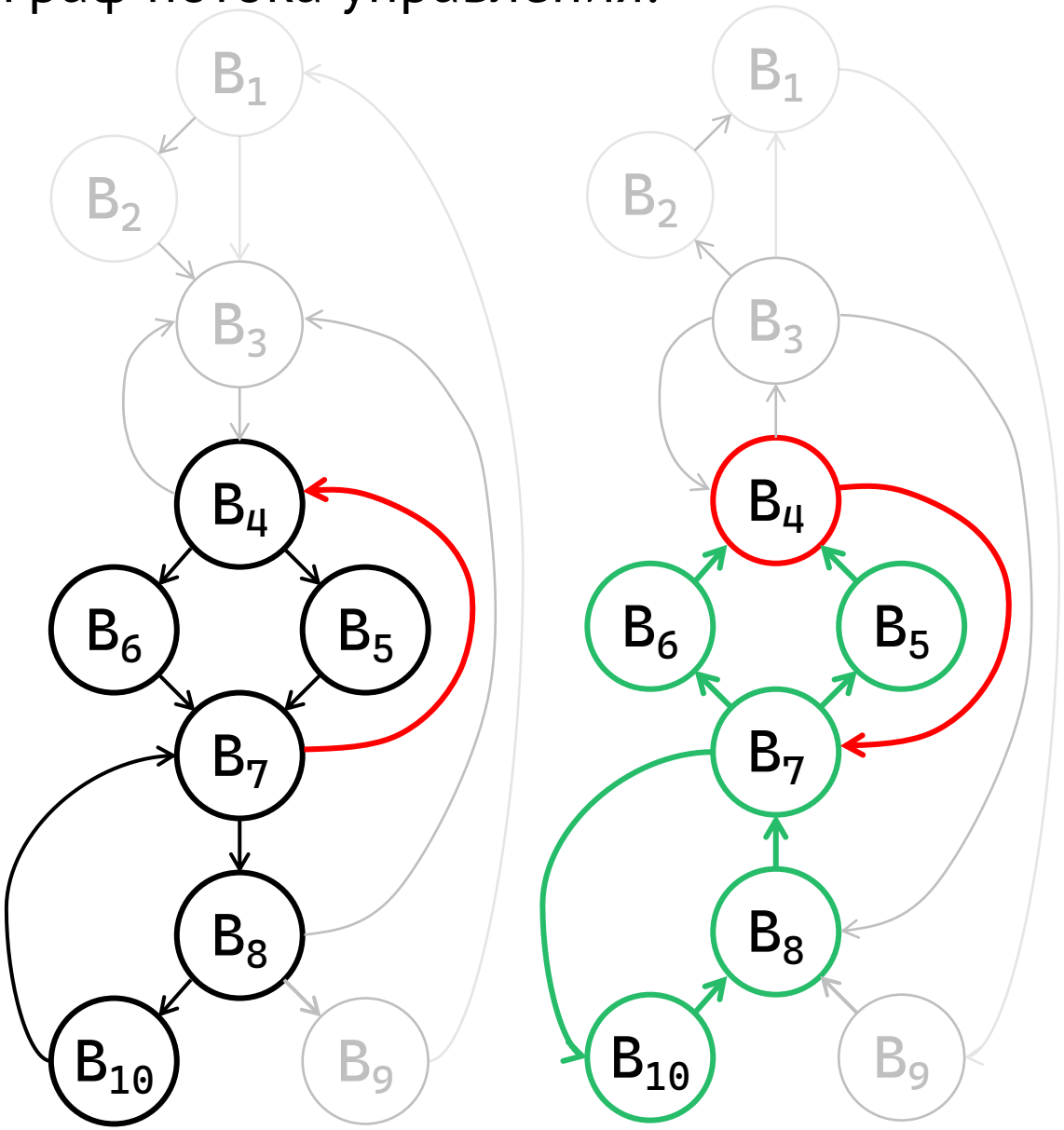
Граф потока управления:



- Применим алгоритм построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$
- Изменим направление дуг на противоположное и выполним поиск в глубину, начиная с вершины B_7, отметив вершину B_4 как посещенную.

Выделение естественных циклов. Пример построения естественного цикла по обратной дуге.

Граф потока управления:

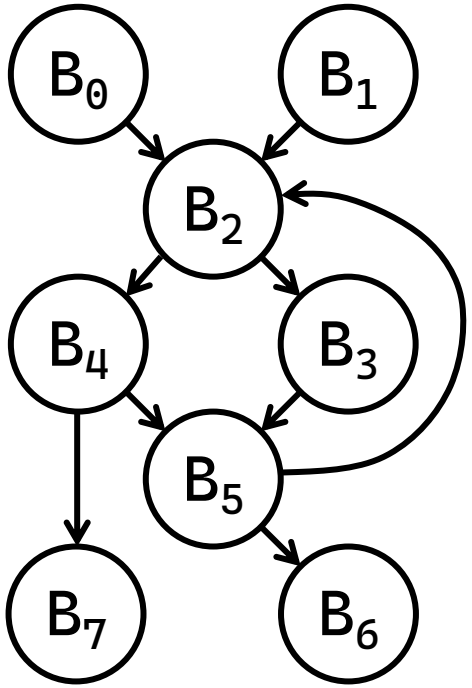


- Применим алгоритм построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$
- Изменим направление дуг на противоположное и выполним поиск в глубину, начиная с вершины B_7, отметив вершину B_4 как посещенную.
- На рисунке справа

$$Succ(B_7) = \{B_5, B_6, B_{10}\}$$
$$Succ(B_{10}) = \{B_8\}$$

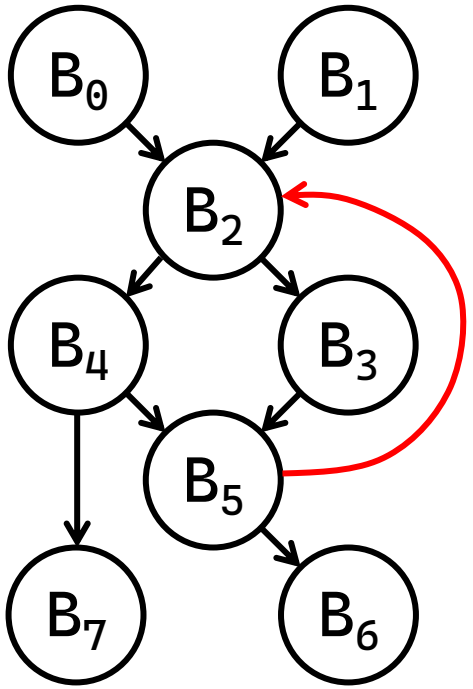
Следовательно, в состав цикла помимо концов обратной дуги (блоков B_7 и B_4), войдут блоки B_5, B_6, B_{10}, B_8

Естественные циклы. Терминология



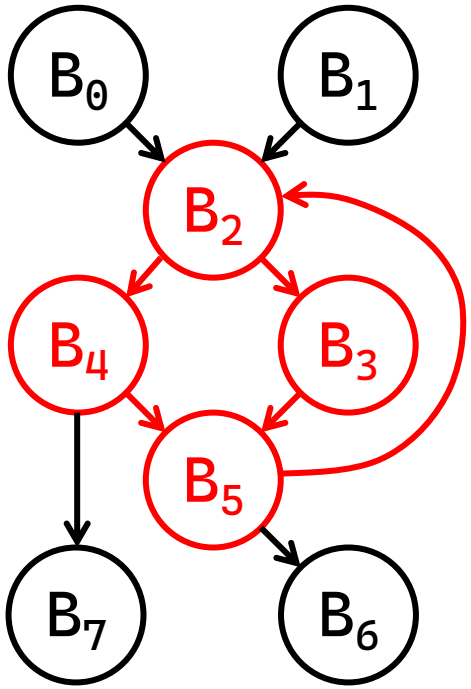
Естественные циклы. Терминология

- Обратное ребро



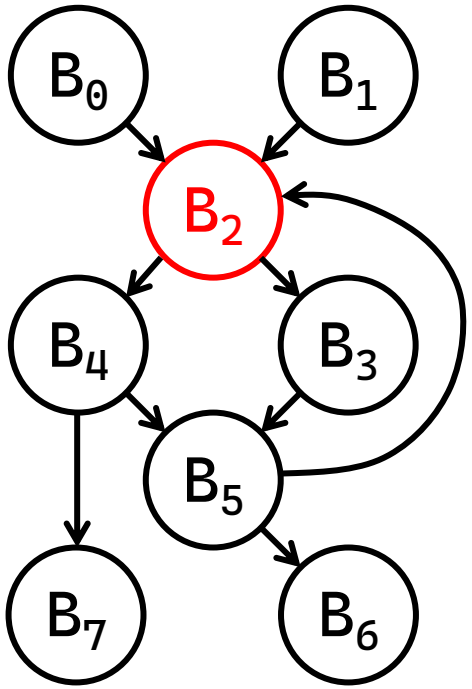
Естественные циклы. Терминология

- Обратное ребро
- Естественный цикл

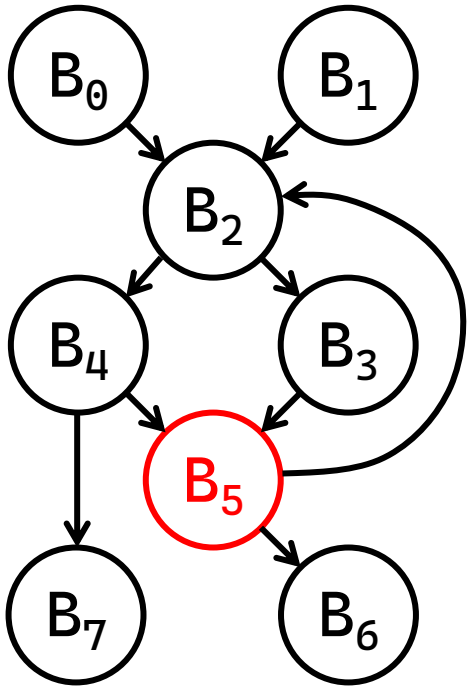


Естественные циклы. Терминология

- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)

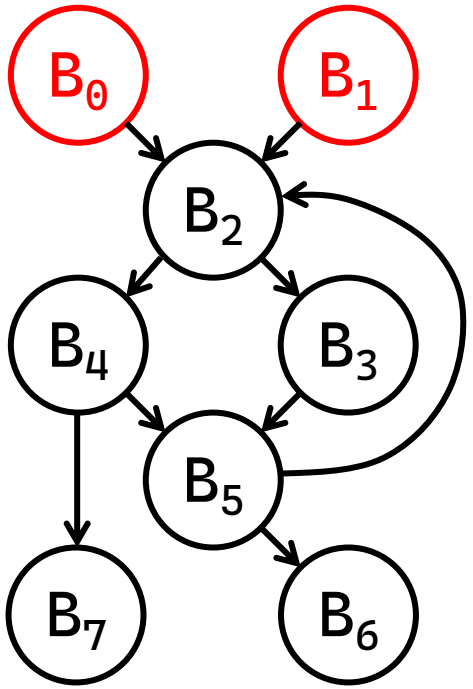


Естественные циклы. Терминология



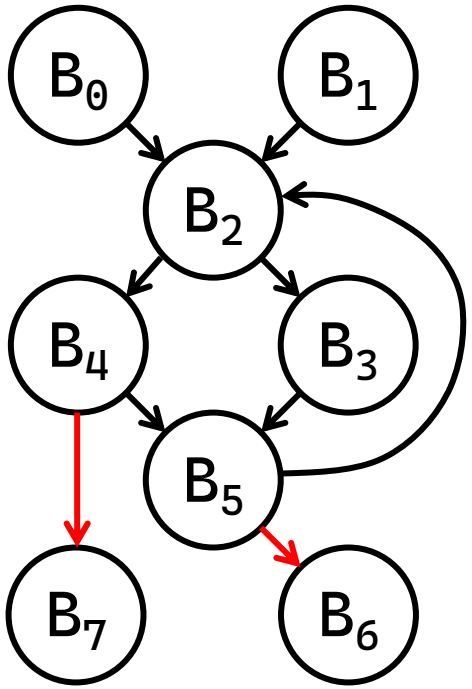
- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)
- Защёлка естественного цикла (latch) — узел, являющийся источником обратного ребра

Естественные циклы. Терминология



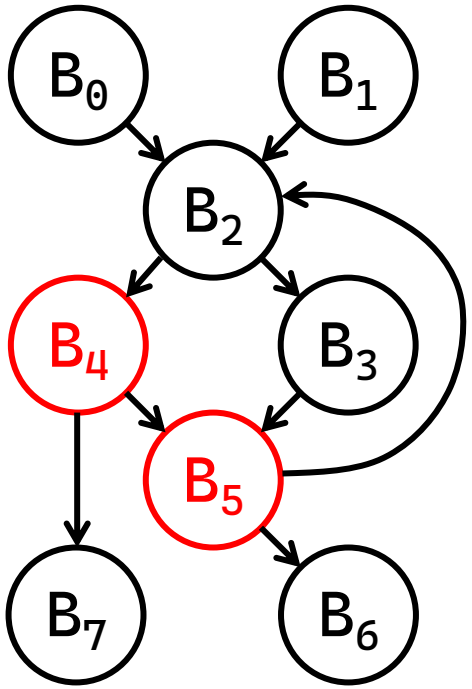
- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)
- Защёлка естественного цикла (latch) — узел, являющийся источником обратного ребра
- Входящие узлы
  (входящие в естественный цикл узлы,
  предшественники естественного цикла)

Естественные циклы. Терминология



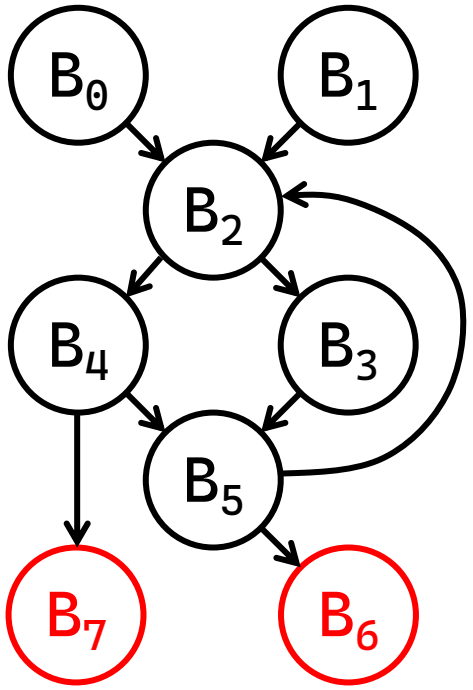
- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)
- Защёлка естественного цикла (latch) — узел, являющийся источником обратного ребра
- Входящие узлы
  (входящие в естественный цикл узлы,
  предшественники естественного цикла)
- Выходящие ребра

Естественные циклы. Терминология



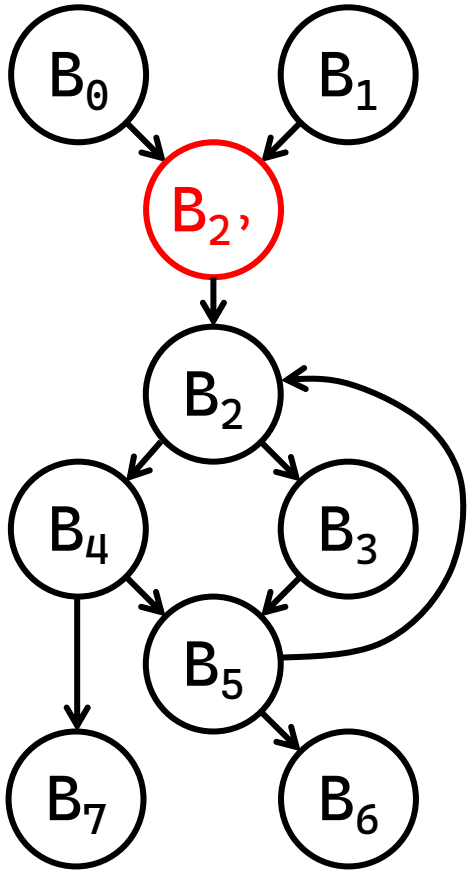
- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)
- Защёлка естественного цикла (latch) — узел, являющийся источником обратного ребра
- Входящие узлы
  (входящие в естественный цикл узлы,
  предшественники естественного цикла)
- Выходящие ребра
- Выходящие узлы (exiting)
  узлы, являющиеся выходом из цикла

Естественные циклы. Терминология



- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)
- Защёлка естественного цикла (latch) — узел, являющийся источником обратного ребра
- Входящие узлы
  (входящие в естественный цикл узлы,
  предшественники естественного цикла)
- Выходящие ребра
- Выходящие узлы (exiting)
  узлы, являющиеся выходом из цикла
- Пограничные узлы (exit)

Естественные циклы. Терминология

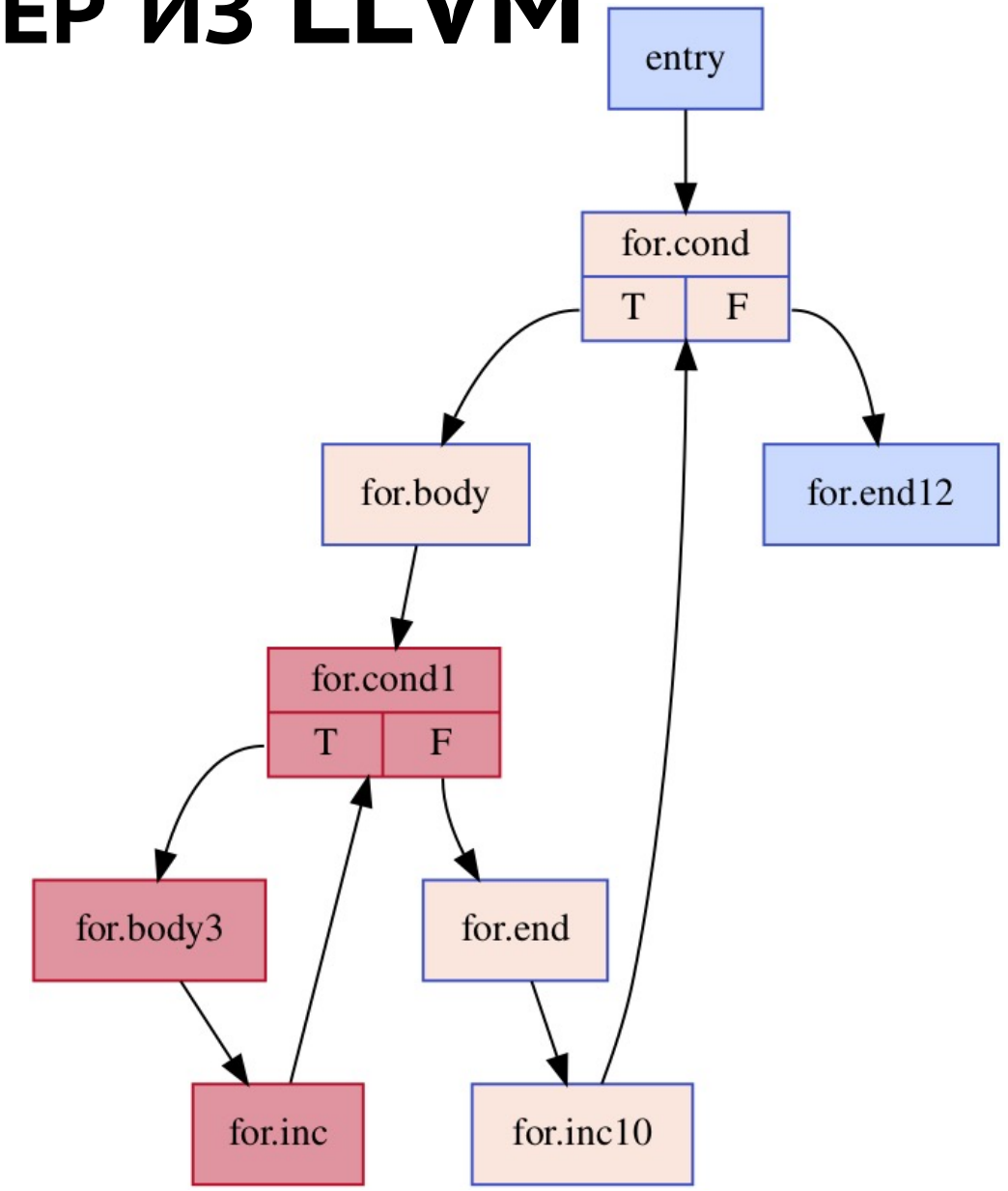


- Обратное ребро
- Естественный цикл
- Заголовок естественного цикла (header)
- Защёлка естественного цикла (latch) — узел, являющийся источником обратного ребра
- Входящие узлы
  (входящие в естественный цикл узлы,
  предшественники естественного цикла)
- Выходящие ребра
- Выходящие узлы (exiting)
  узлы, являющиеся выходом из цикла
- Пограничные узлы (exit)
- Предзаголовок цикла (preheader)

Естественные циклы. Пример из LLVM

```c
int array[N][N];

int licm(int n) {
  for (int i = 0; i < N; i++) {
    int l = i * (n + 2);
    for (int j = 0; j < N; j++)
      array[i][j] = 100 * n + 10 * l + j;
  }
  return 0;
}
```

```
$ clang -O0 -g0 -c -emit-llvm -S licm.c -o licm.ll
$ opt -mem2reg licm.ll -S -o licm-ssa.ll
$ opt -dot-cfg-only licm-ssa.ll -disable-output

$ opt -passes='print<loops>' licm-ssa.ll -disable-output
Loop at depth 1 containing:
%for.cond<header><exiting>,%for.body,
            %for.cond1,%for.end,%for.inc10<latch>,
            %for.body3,%for.inc
    Loop at depth 2
    containing: %for.cond1<header><exiting>,
                %for.body3,%for.inc<latch>
```



Естественные циклы. Предобработка естественных циклов

- Предобработка циклов необходима для приведения цикла к каноническому виду, упрощающему анализ и преобразование естественных циклов
- Предобработка циклов:
  - поиск *естественных циклов* (алгоритм уже рассмотрен)
  - приведение цикла к *каноническому виду*:
    - «вращение цикла» (Loop Rotate)
    - приведение цикла к упрощенной форме (Loop Simplify)
  - построение замкнутой относительно цикла SSA-формы (LCSSA-форма — Loop-Closed SSA-форма)
- *Каноническая форма цикла* — форма цикла в которой цикл содержит только одно обратное ребро и одну защёлку, а перед циклом вставлен предзаголовок цикла

Естественные циклы. Предобработка естественных циклов. Вращение цикла

- Преобразование цикла типа «**while**» в цикл типа «**do-while**»

```
/** Исходный цикл */
for (int i = 0; i < n; i++) {
    /** ... */
}
```

- трансформация возможна, только если компилятору удается доказать факт обязательного выполнения хотя бы одной итерации цикла
- в противном случае компилятор должен сгенерировать соответствующую проверку

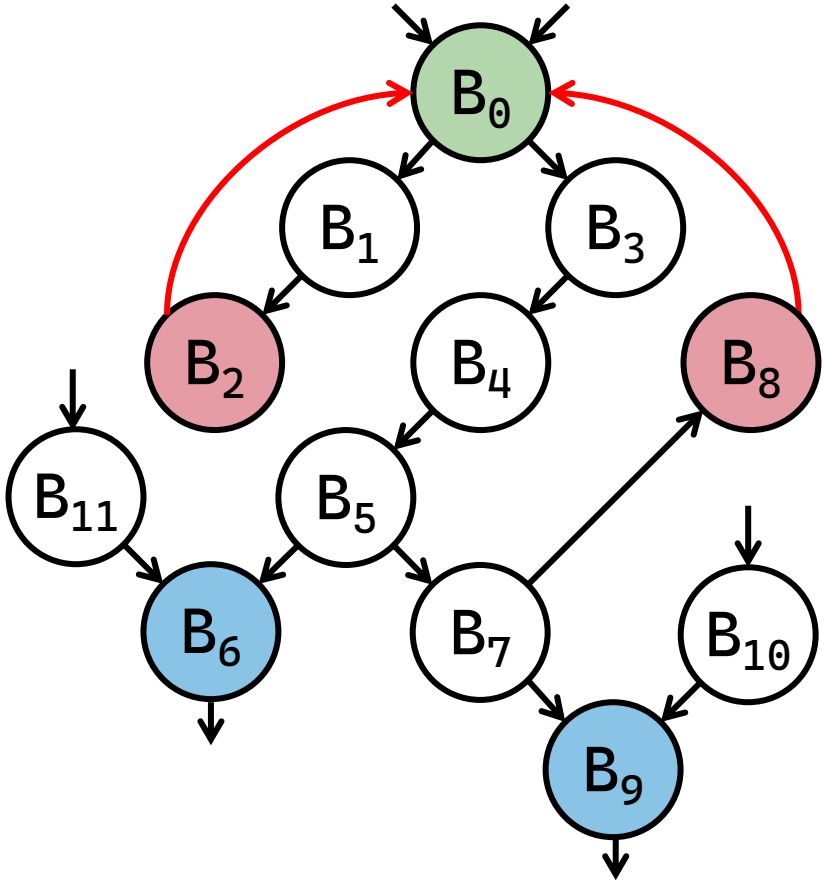
```
/** Псевдокод цикла после обычного
 *  понижения представления */
int i = 0;
while (i < n) {
    /* ... */
    i++;
} /** Branch */
```

```
/** Псевдокод цикла после
 *  вращения цикла */
int i = 0;
if (i < n) { /** Branch */
    do {
        /* ... */
        i++;
    } while (i < n); /** Branch */
}
```

Естественные циклы. Предобработка естественных циклов. Приведение цикла к упрощенной форме

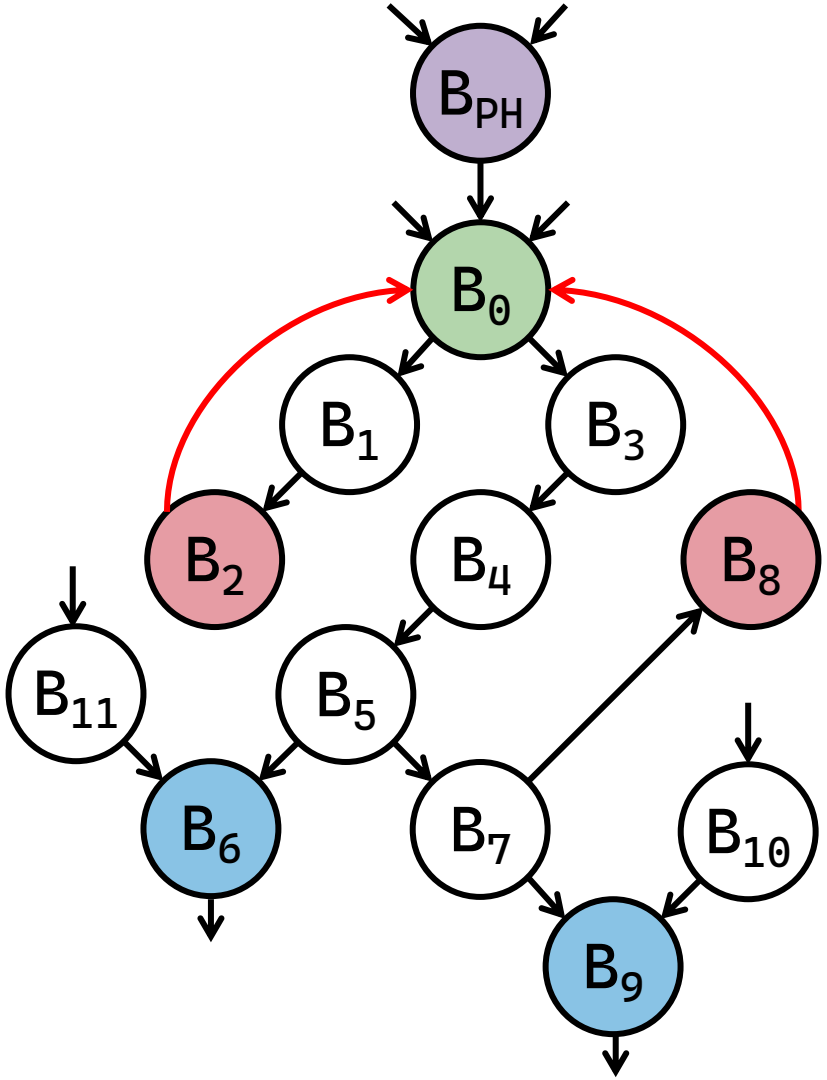
- Перед циклом добавляется предзаголовок
- Наличие предзаголовка гарантирует существование только **одного** входного ребра в цикл
- Цикл приводится к виду, содержащему только **одно единственное** обратное ребро
- Для цикла добавляются *выделенные пограничные блоки*, пограничные блоки, предшественники которых **обязаны** принадлежать циклу
- Подобная форма упрощает выполнение оптимизирующих преобразований циклов

Естественные циклы. Предобработка естественных циклов. Приведение цикла к упрощенной форме



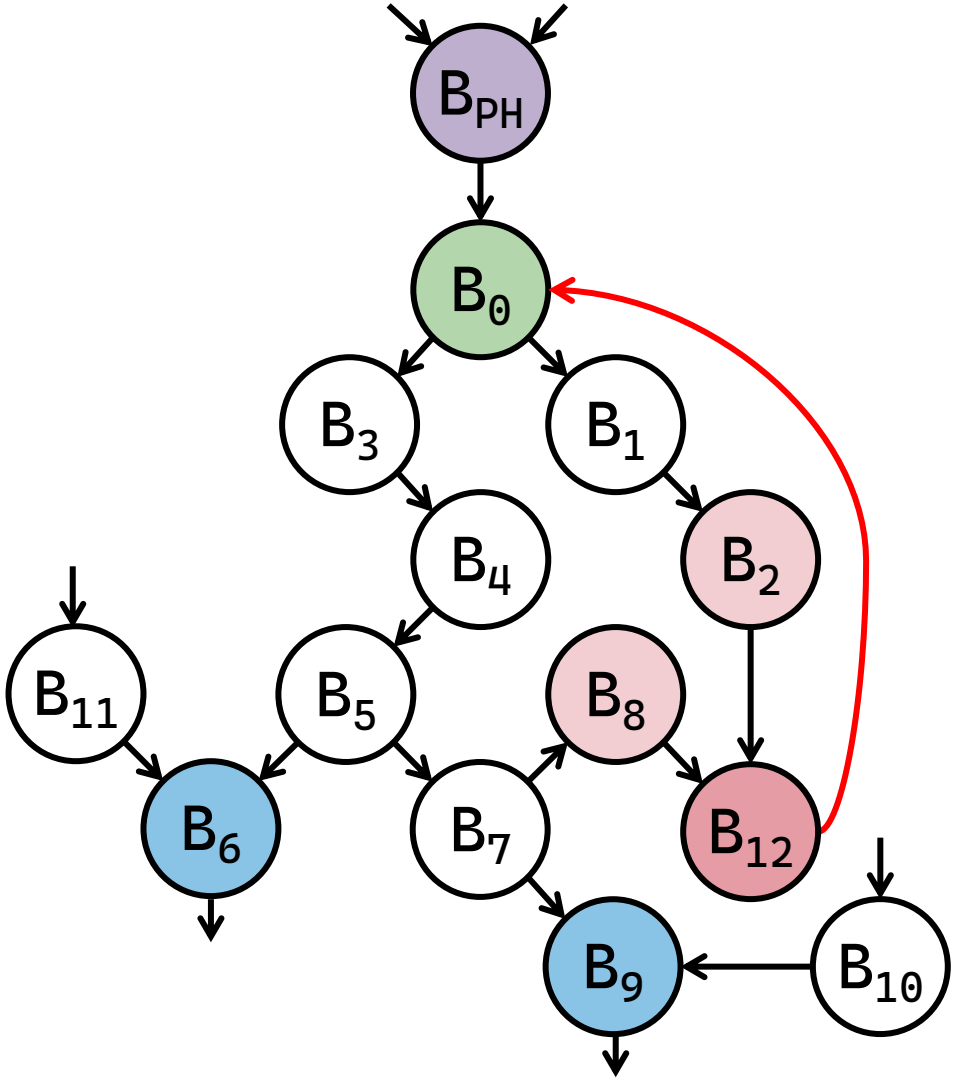
- Заголовок гнезда циклов: B_0
- Два обратных ребра $B_8 \rightarrow B_0$ и $B_2 \rightarrow B_0$
- Две защёлки циклов B_8 и B_2
- Два пограничных блока: B_9 и B_6

Естественные циклы. Предобработка естественных циклов. Приведение цикла к упрощенной форме



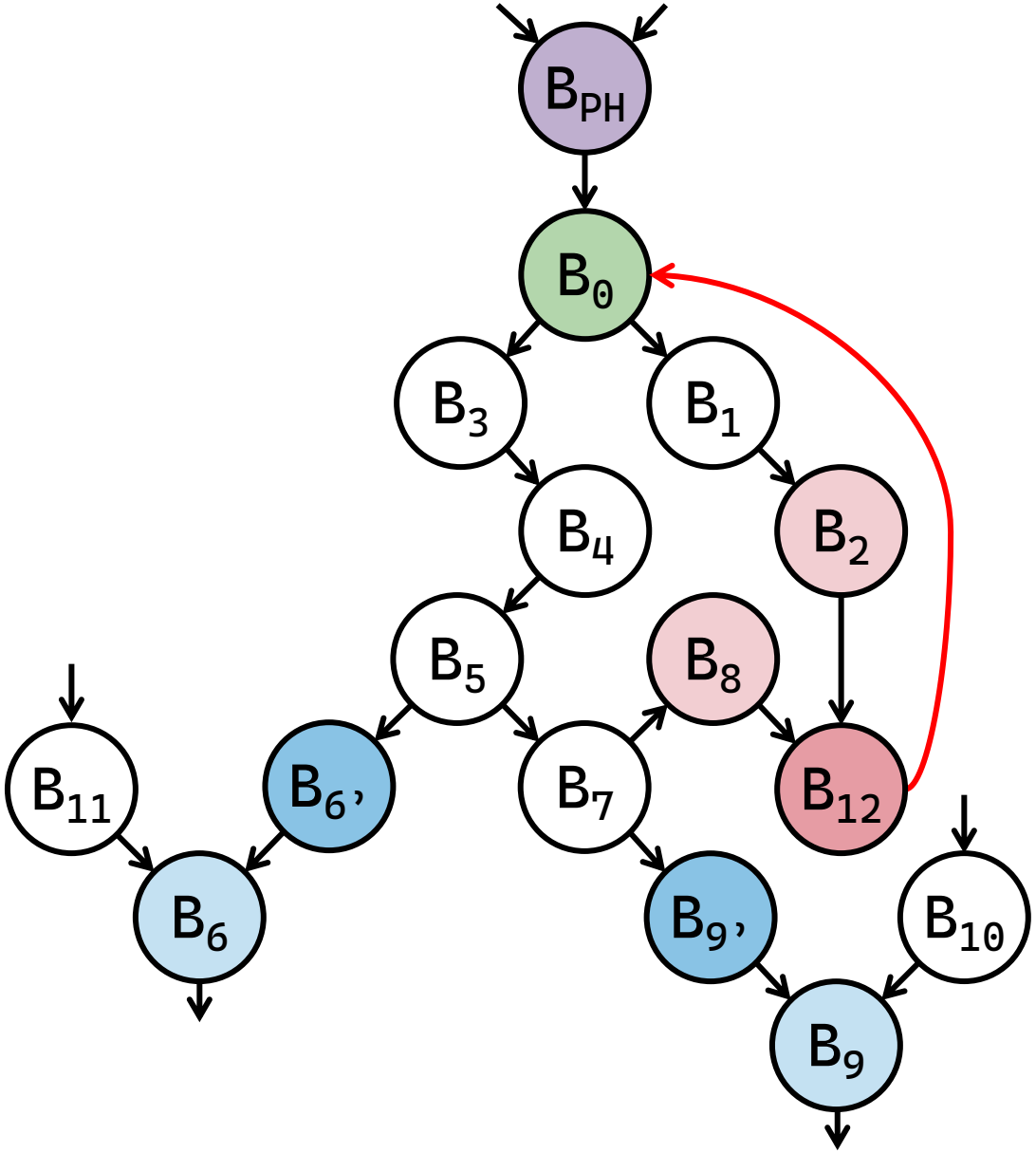
- Добавлен предзаголовок B_{PH}

Естественные циклы. Предобработка естественных циклов. Приведение цикла к упрощенной форме



- Добавлен предзаголовок B_{PH}
- Каждый цикл содержит только одно обратное ребро

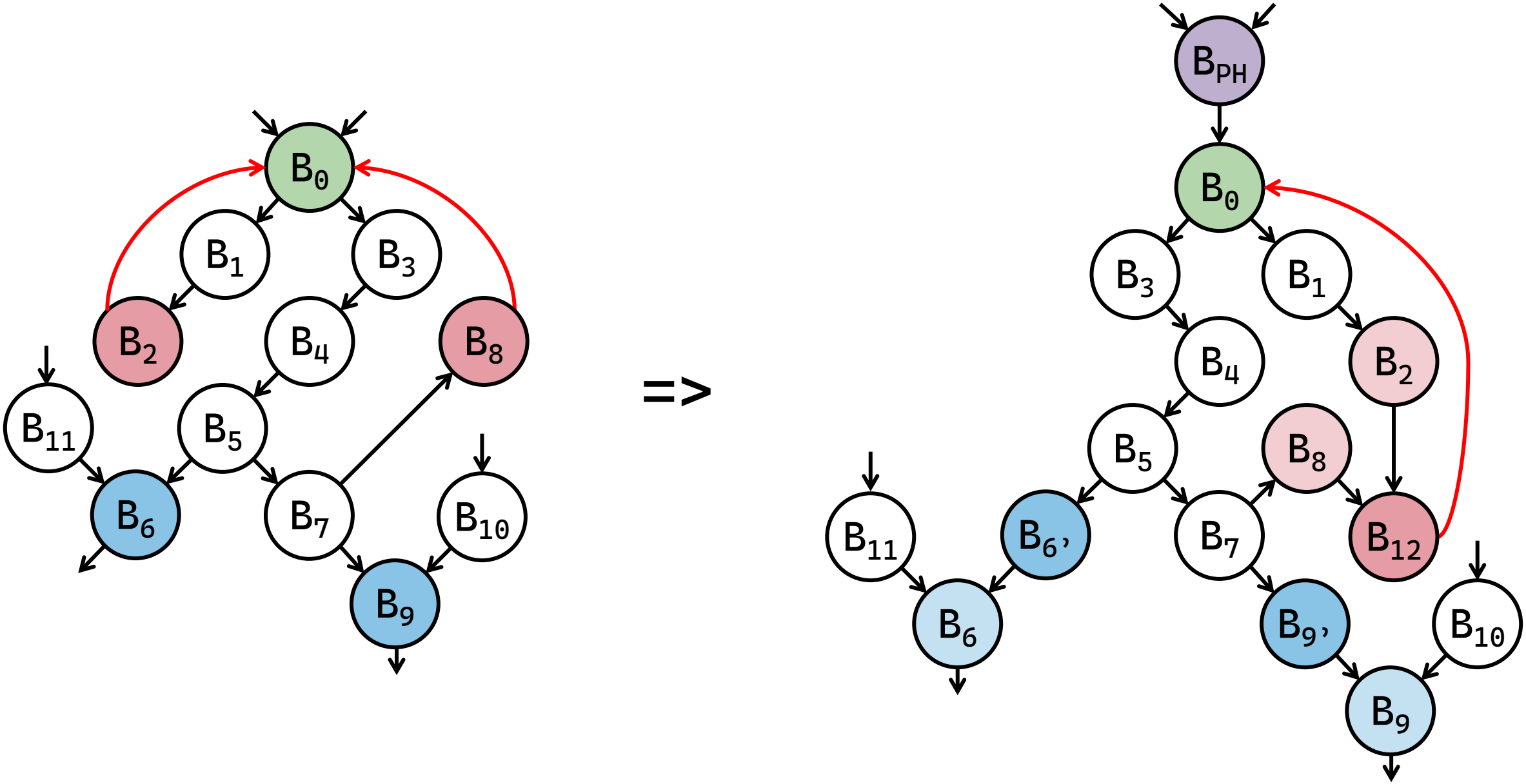
Естественные циклы. Предобработка естественных циклов. Приведение цикла к упрощенной форме



- Добавлен предзаголовок B_{PH}
- Каждый цикл содержит только одно обратное ребро
- Пограничные блоки $B_{9'}$ и $B_{6'}$ достижимы только из тела цикла

Естественные циклы. Предобработка естественных циклов.

Приведение цикла к упрощенной форме



Естественные циклы. Предобработка естественных циклов. Построение замкнутой относительно цикла SSA-формы (LCSSA-форма)

```
/** Обычная SSA-форма */
for (...)
  if (c)
    X1 = ...
  else
    X2 = ...
  X3 = phi(X1, X2)
... = X3 + 4
```

```
/** LCSSA-форма */
for (...)
  if (c)
    X1 = ...
  else
    X2 = ...
  X3 = phi(X1, X2)
X4 = phi(X3)
... = X4 + 4
```

- Преобразование изменяет циклы путем переноса φ-функций в конец циклов для всех значений, которые остаются живыми за границами цикла
- Дополнительные φ-функций являются излишними, но они легко удалятся в будущем
- Основная выгода данного преобразования – это упростить применение других преобразований, например LoopUnswitch

Перемещение кода, инвариантного относительно цикла. 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
    - это определение находится в базовом блоке, являющемся **доминатором всех выходов из цикла**
    - определяющая переменную v инструкция является **инвариантной относительно цикла**
- Инструкция *инвариантна относительно цикла*, если все ее операнды являются инвариантами цикла

Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла

- *Перемещение инвариантного по отношению к циклу кода* (Loop Invariant Code Motion — LICM) — оптимизация, выполняющая ***обнаружение*** вычислений, производящих ***одинаковый результат*** на каждой итерации и их ***перемещение*** за границы 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

Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 0

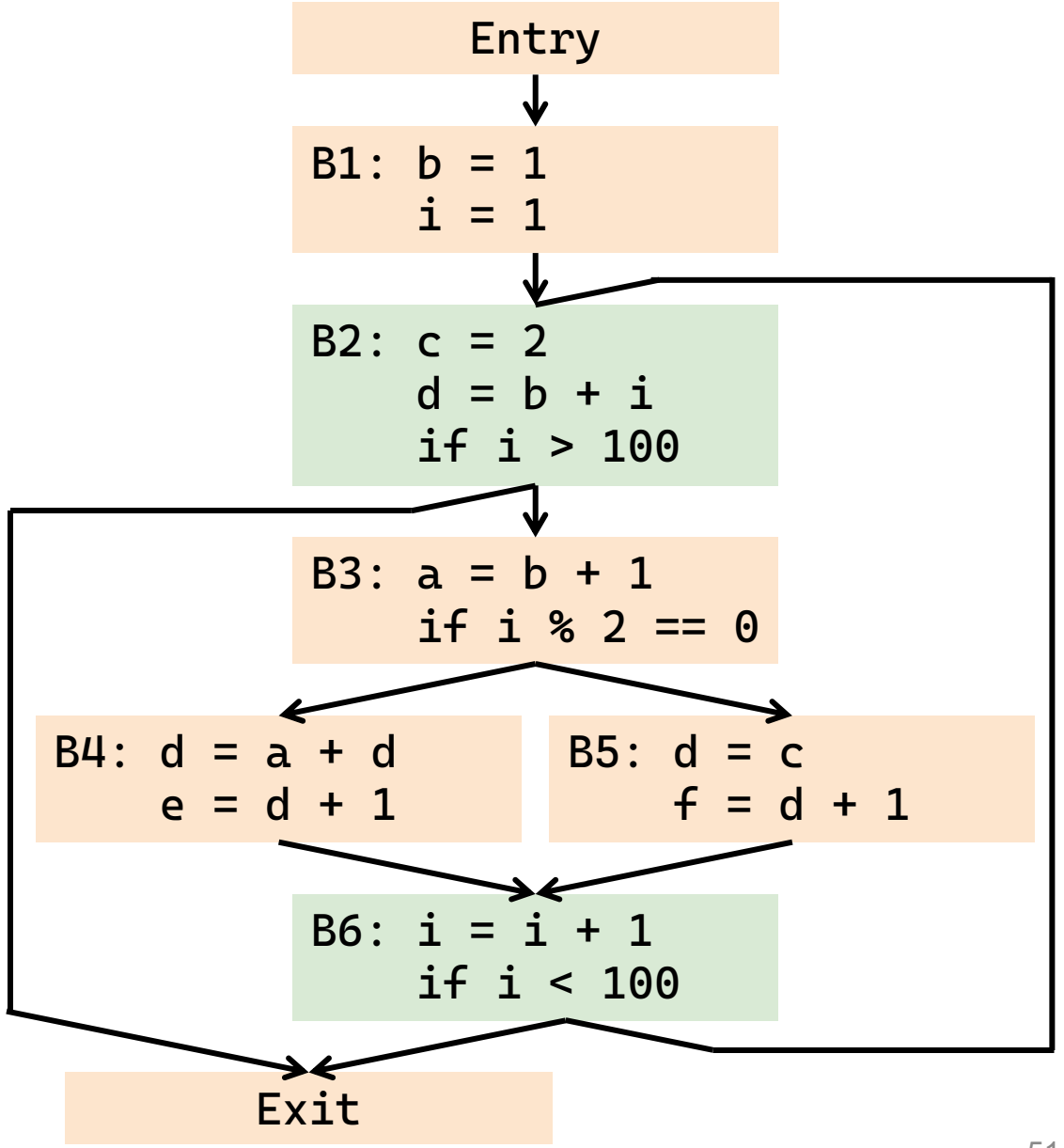
```
int licm(int n)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            array[i][j] = 100 * n +
                          10 * (i * (n + 2)) +
                          j;
        }
    }
    return 0;
}
```

```
int licm(int n)
{
    t1 = 10 * (n + 2);
    t2 = 100 * n;
    for (int i = 0; i < N; i++)
    {
        t3 = t2 + i * t1
        for (int j = 0; j < N; j++)
        {
            array[i][j] = t3 + j;
        }
    }
    return 0;
}
```

- Инварианты внутреннего цикла: **`10 * i * (n + 2)`**, **`100 * n`**
- Инвариант гнезда цикла: **`10 * (n + 2)`**, **`100 * n`**
- Вынос инвариантного по отношению к циклу кода за границы гнезда цикла позволят значительно сократить число выполняющихся операций сложения и умножения

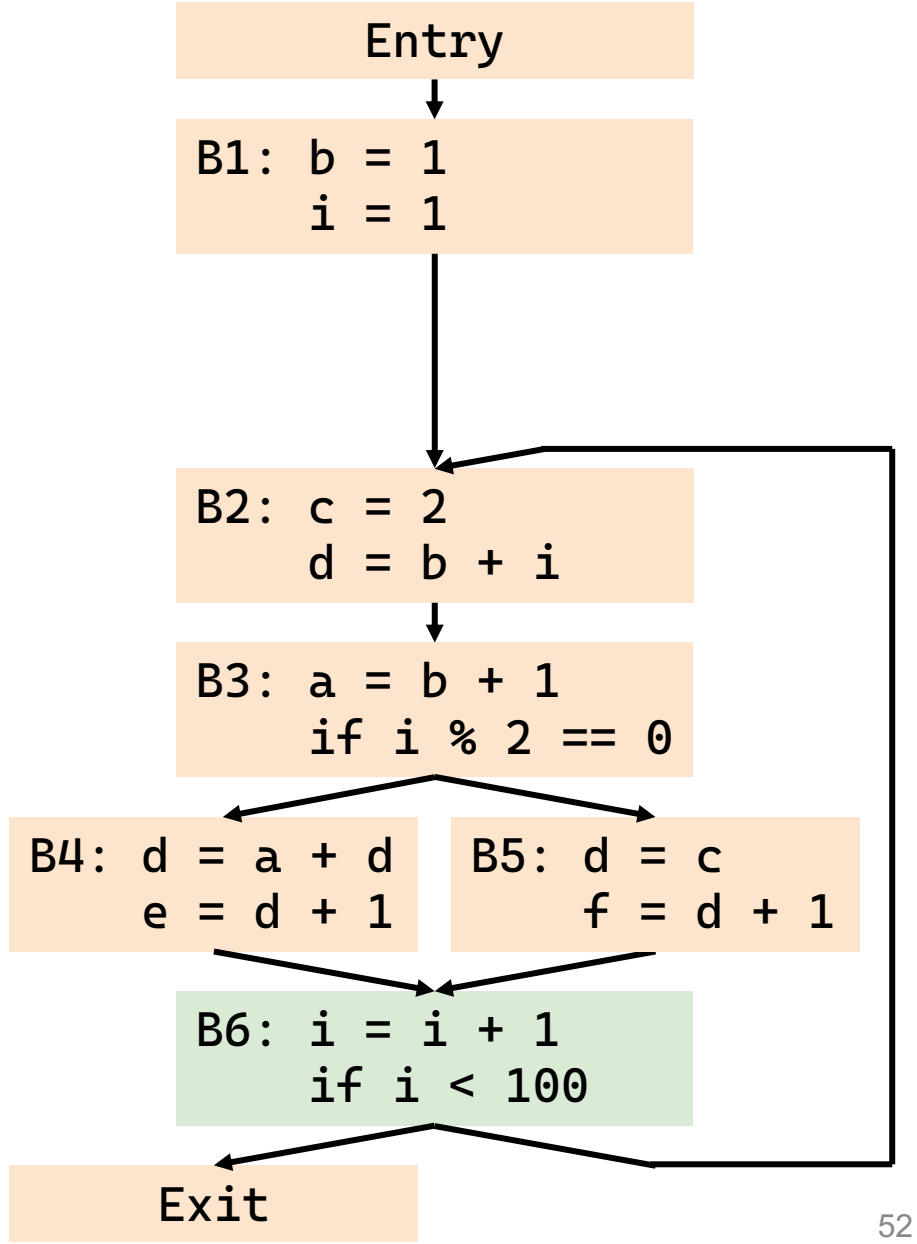
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 1

- Исходный цикл:
- Блок **B1** выполняется до цикла, все остальные – в цикле
- Выходы из цикла – блоки **B2** и **B6**
- Блоки **B3**, **B4**, **B5** и **B6** не являются доминаторами блока **B2**.
- Следовательно, из указанных блоков нельзя выносить код, так как иногда это может не оптимизировать, а пессимизировать программу



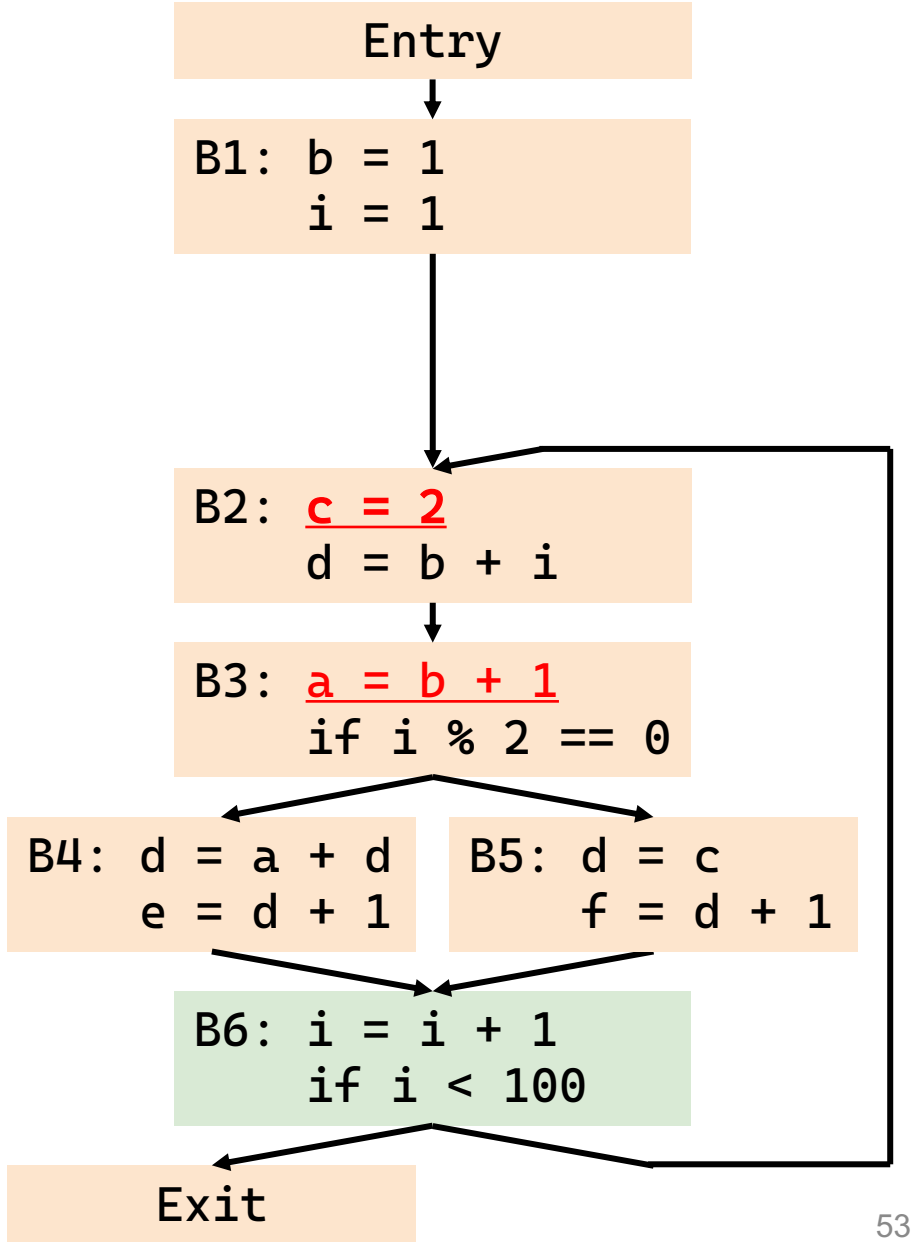
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 1

- Изменим исходный цикл,
  убрав из блока ненужное сравнение
- У цикла останется только один выход – блок **B6**



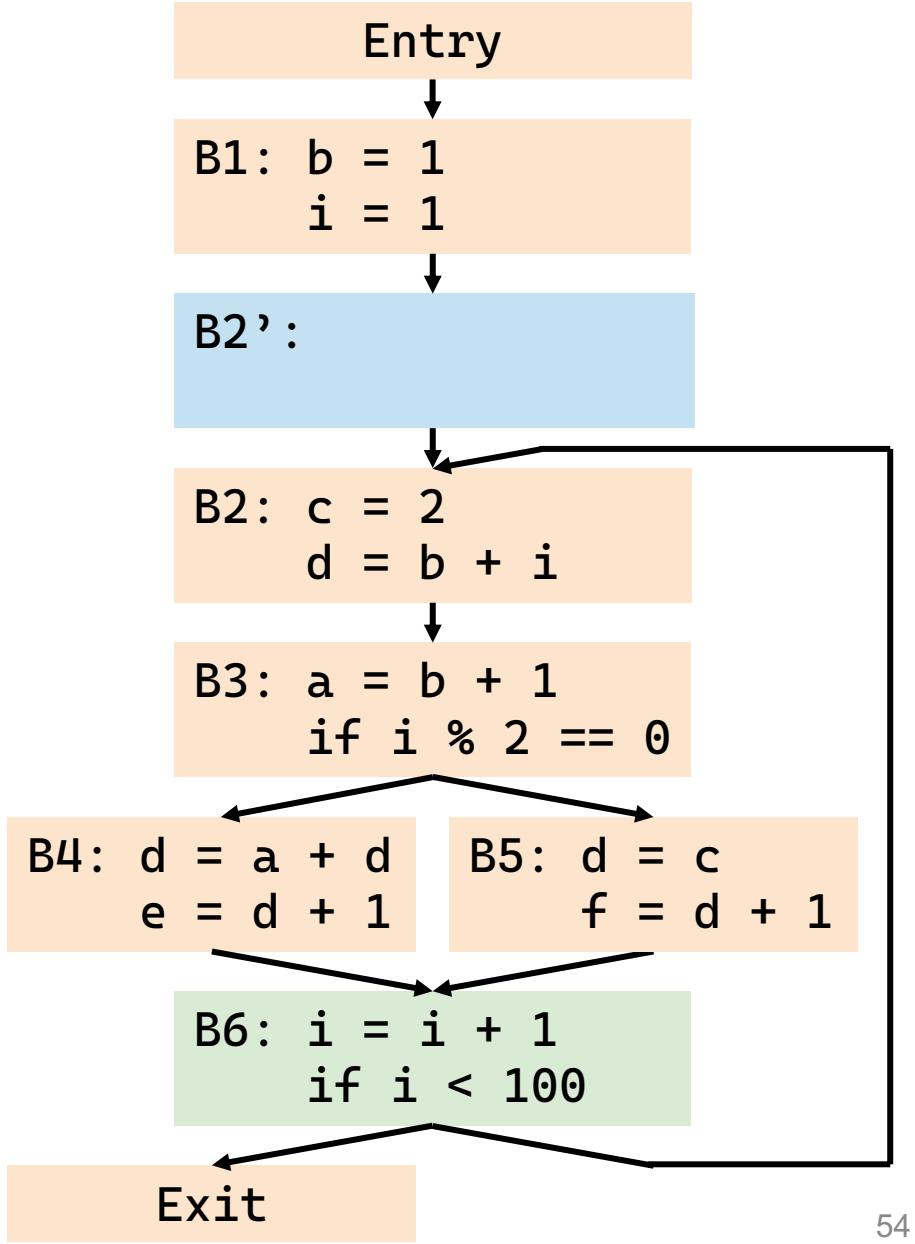
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 1

- Инструкции **`a = b + 1`**, **`c = 2`**, инвариантны относительно цикла:
- **`a = b + 1`**:
  - **`b`** – инвариант цикла, так как его определение находится вне цикла
  - **`1`** – инвариант цикла, как константа
- **`c = 2`**:
  - **`2`** – инвариант цикла, как константа



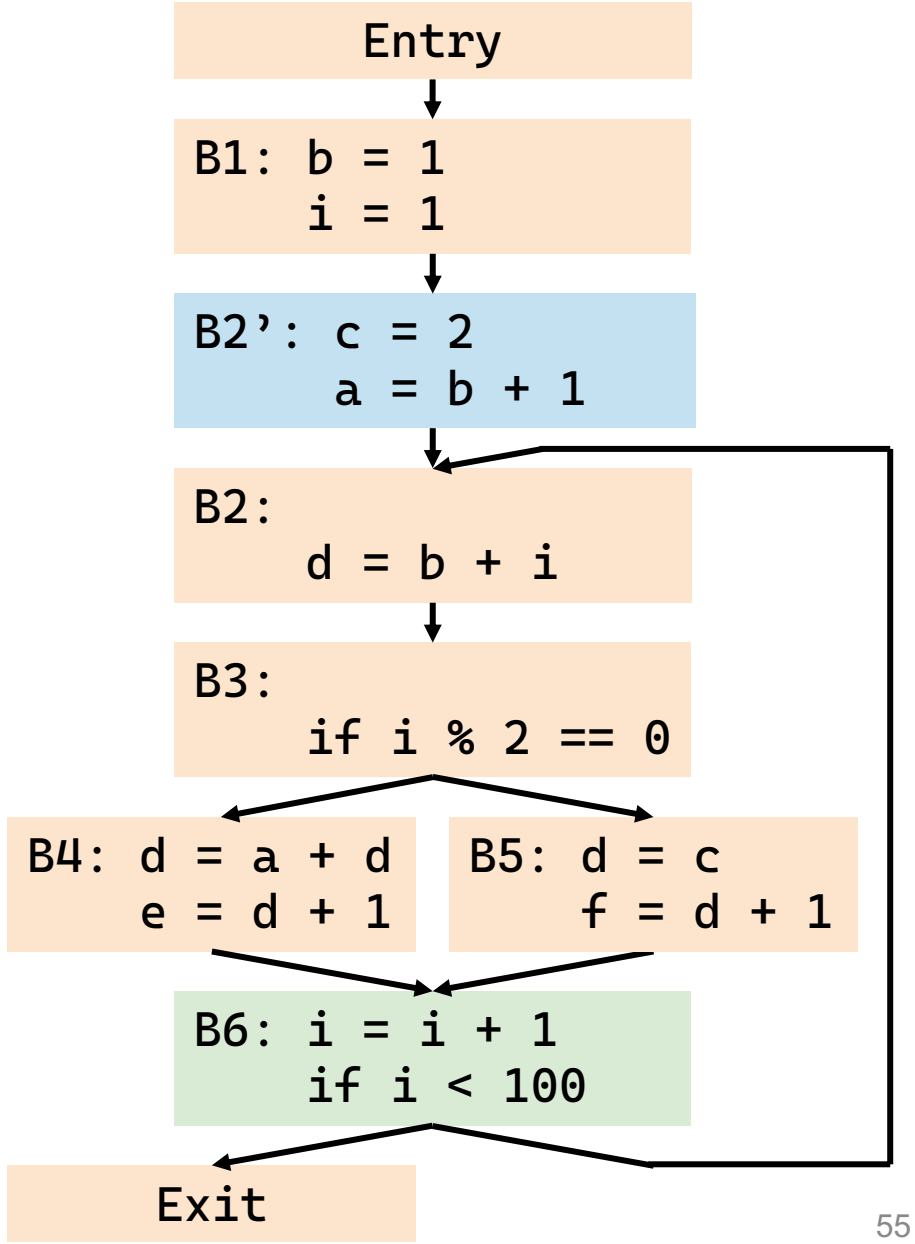
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 1

- После добавления предзаголовка (**B2’**)



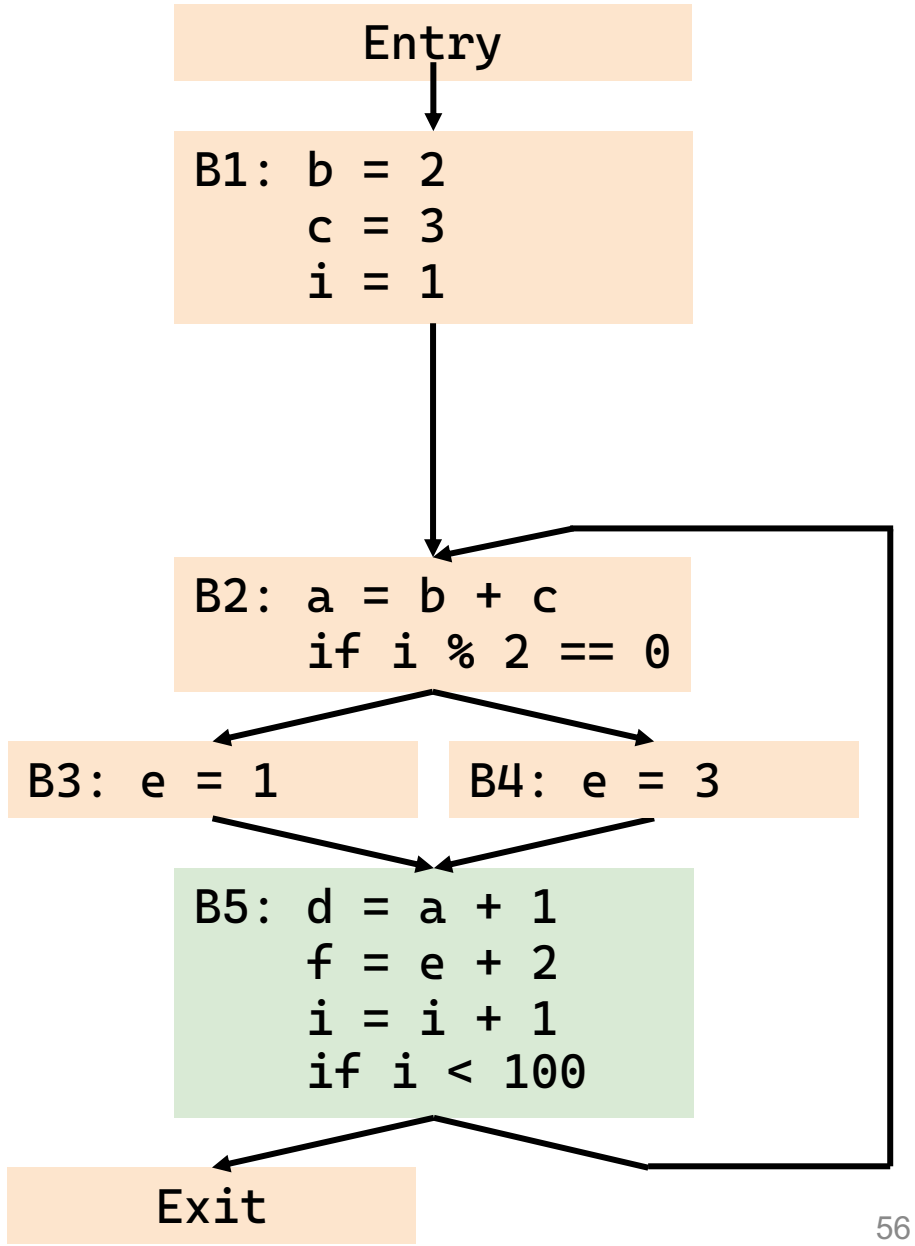
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 1

- После вынесения инвариантного кода в предзаголовок



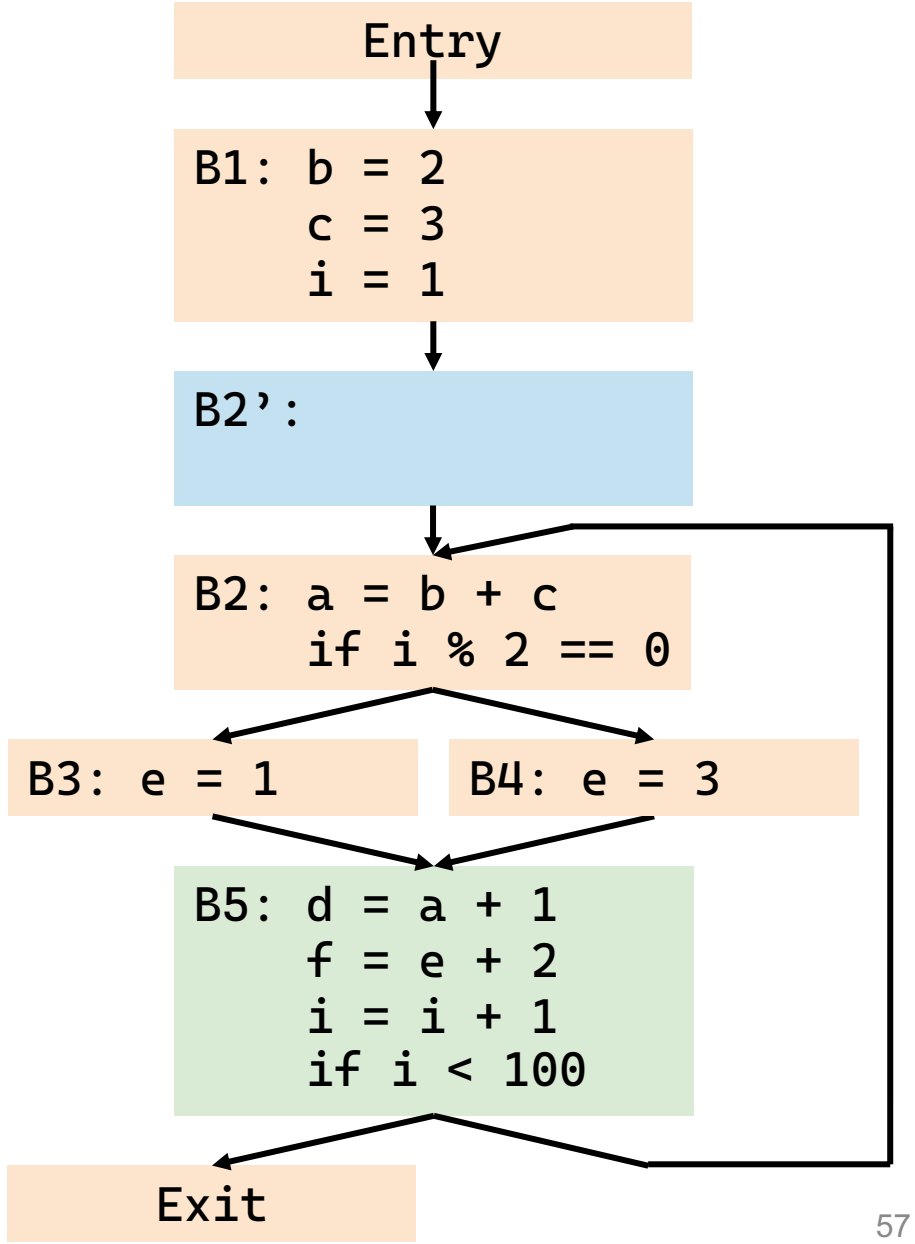
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- Исходный цикл
- Блок **B1** выполняется до цикла, все остальные — в цикле



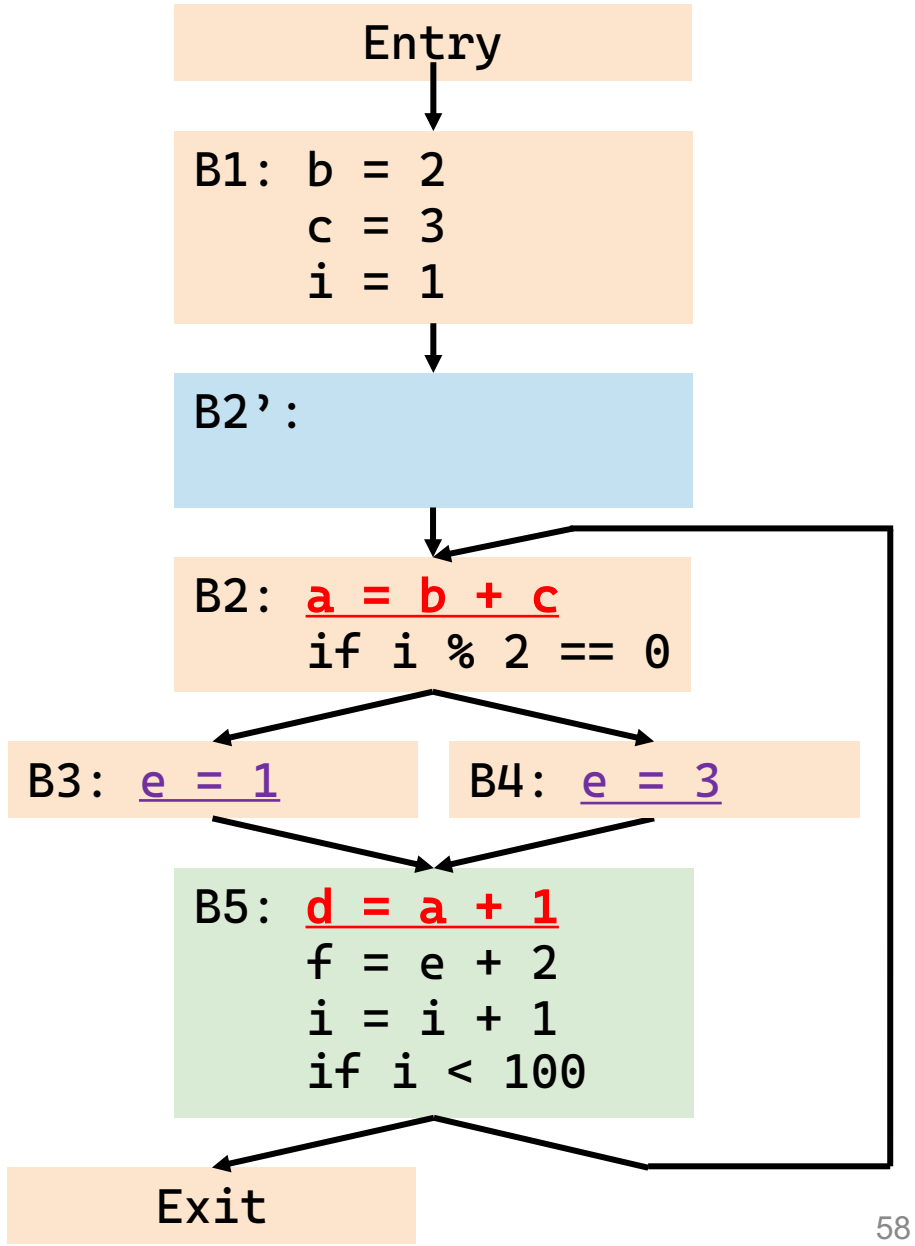
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- После добавления предзаголовка **B2’**



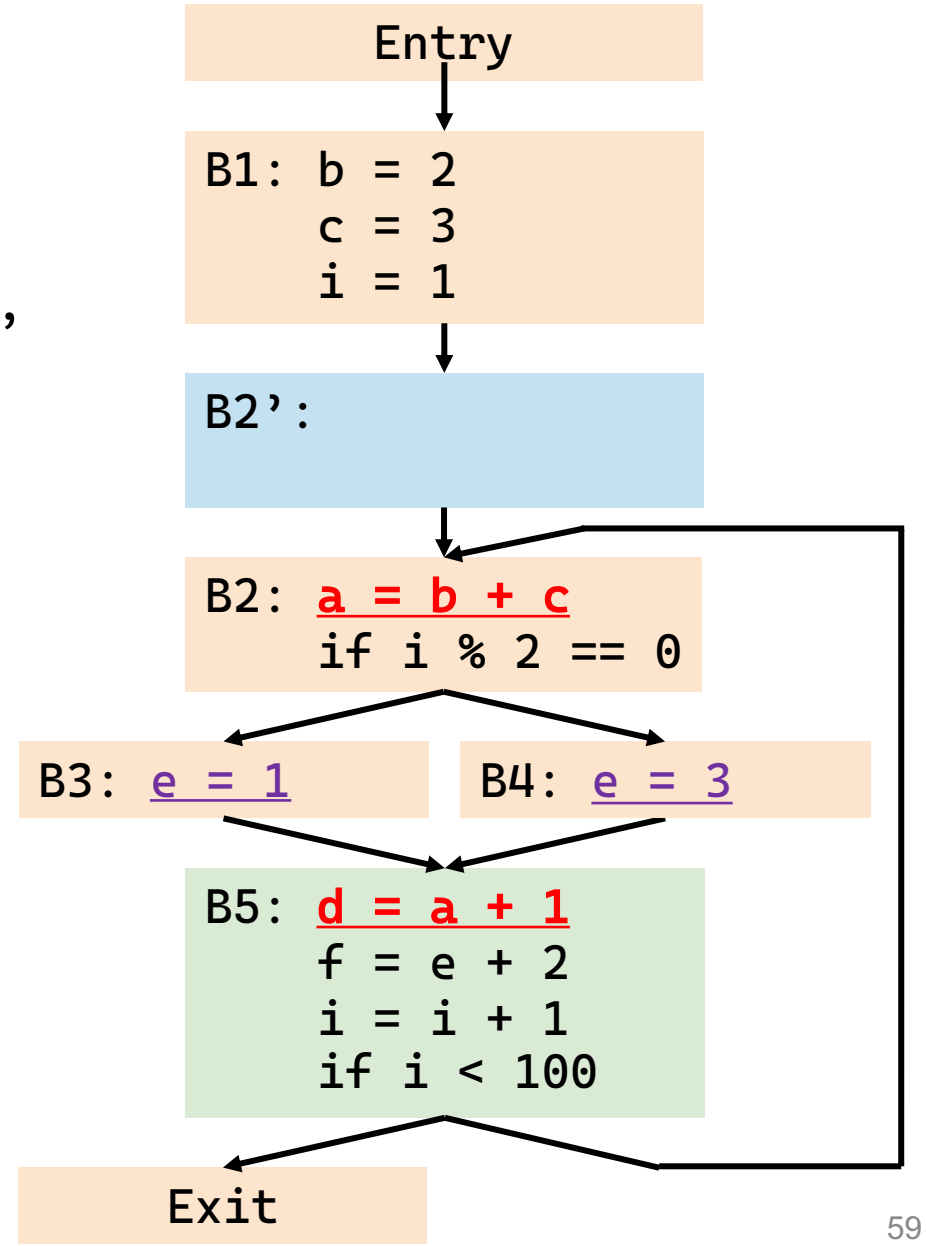
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- Инструкции `a = b + c`, `d = a + 1` инвариантны относительно цикла:
  - `a = b + c`
    определения b и c находятся вне цикла
  - `d = a + 1`:
    - `1` – константа
    - `a` – определено только один раз в блоке, который является доминатором выхода из цикла
- Инструкции `e = 2` и `e = 3` не инвариантны относительно цикла!
  почему?



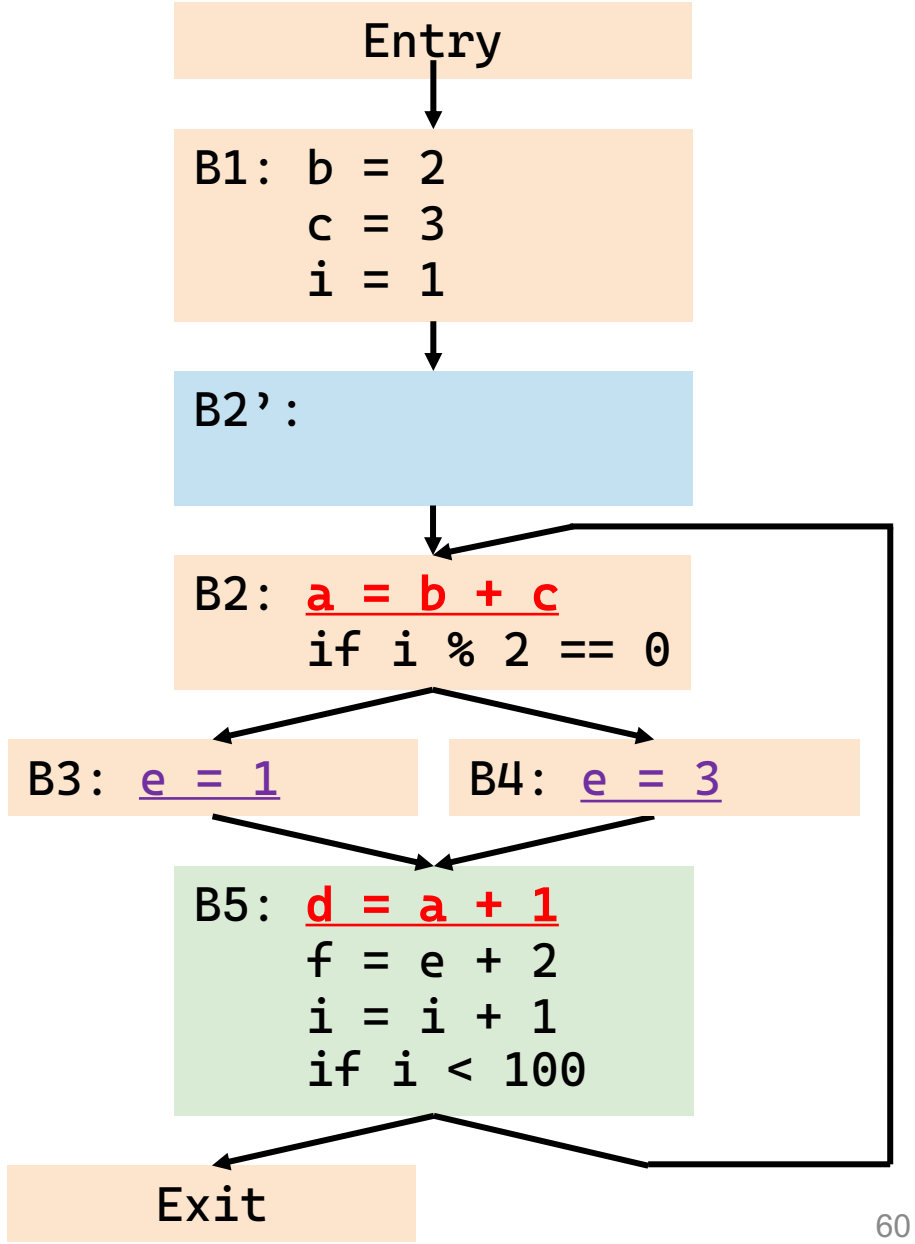
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- Инструкции **e = 2** и **e = 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 предзаголовок **e** будет определяться только один раз и в зависимости от порядка инструкций в предзаголовке будет всегда иметь только одно значение (**2** или **3**)



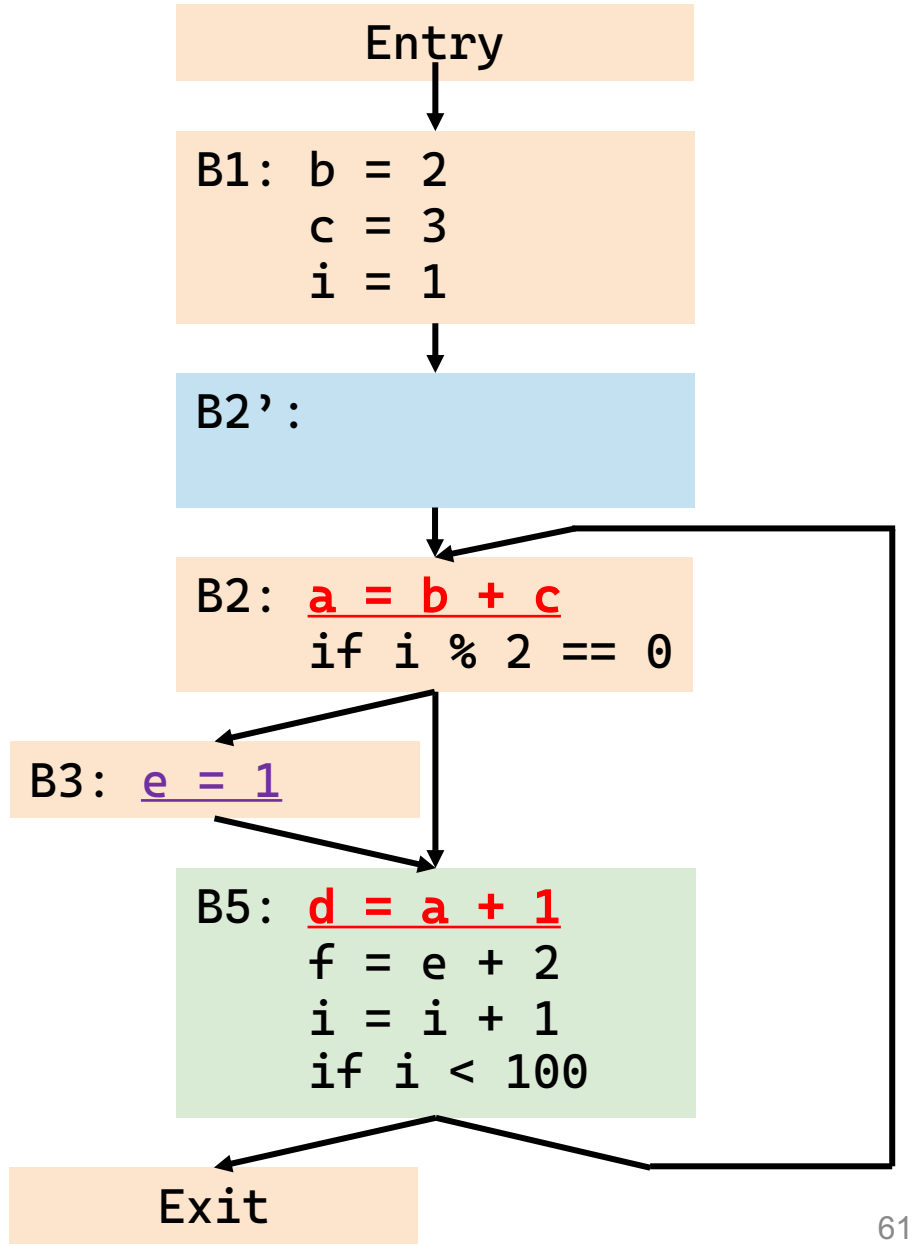
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- Инструкции `e = 2` и `e = 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 предзаголовок можно вынести только инструкции из базовых блоков, являющихся доминаторами всех выходов из цикла



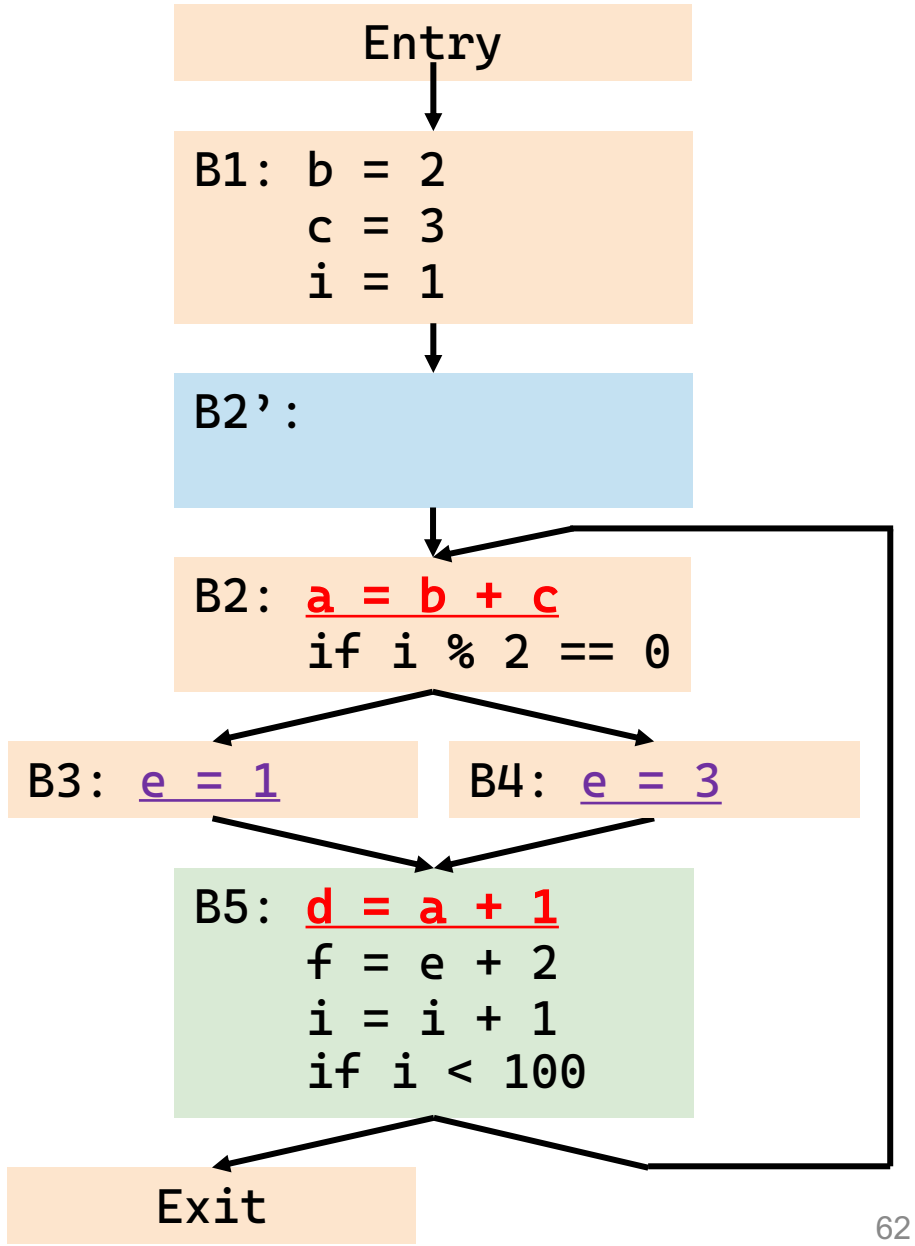
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- В предзаголовок можно выносить только инструкции из базовых блоков являющихся доминаторами **всех выходов из цикла**
- Если, например, удалить из цикла блок **B4**, то **e** будет присваиваться только одно значение
- Но даже в этом случае нельзя выносить инструкцию **e = 2** в предзаголовок, так как блок **B3** все равно не будет доминатором выхода и результат выполнения программы может измениться



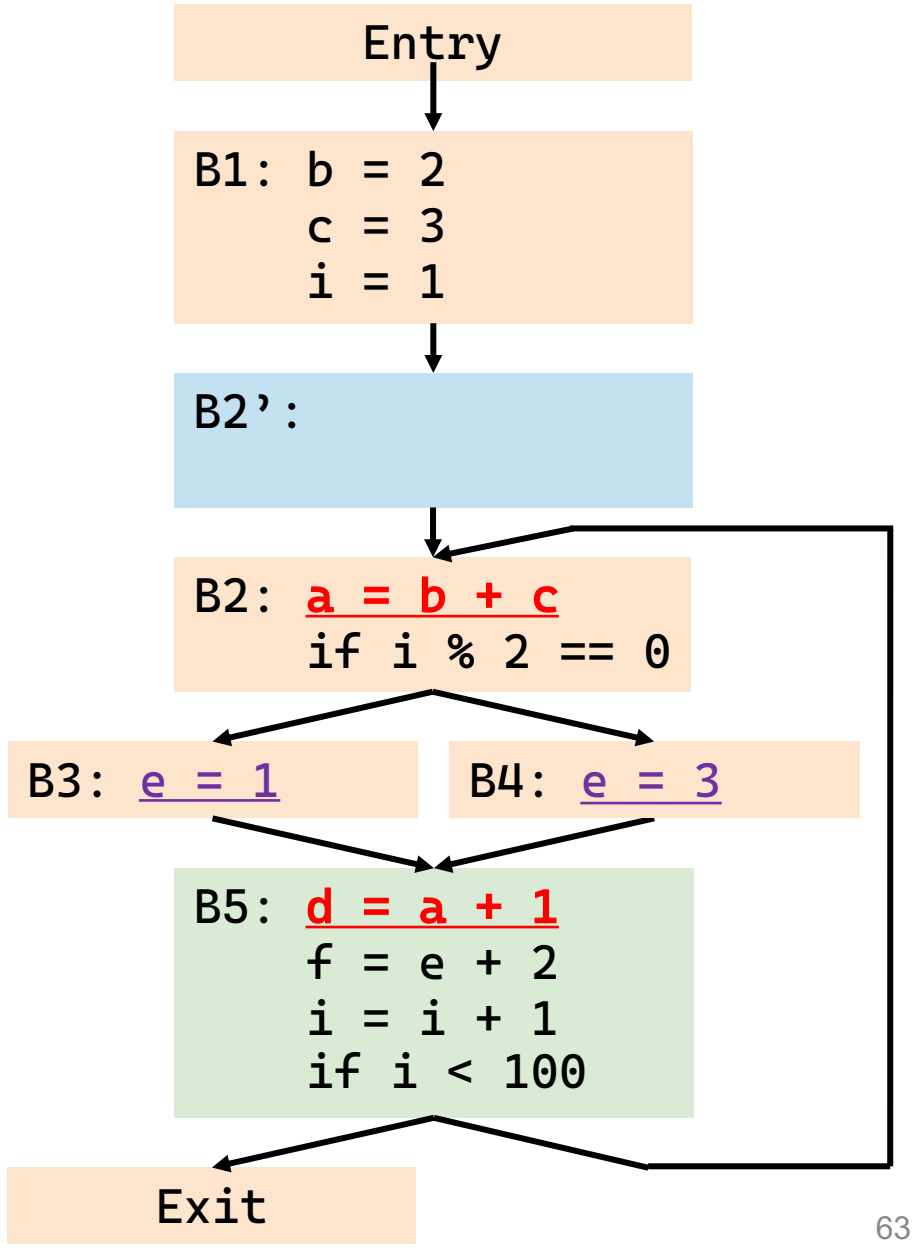
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- 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



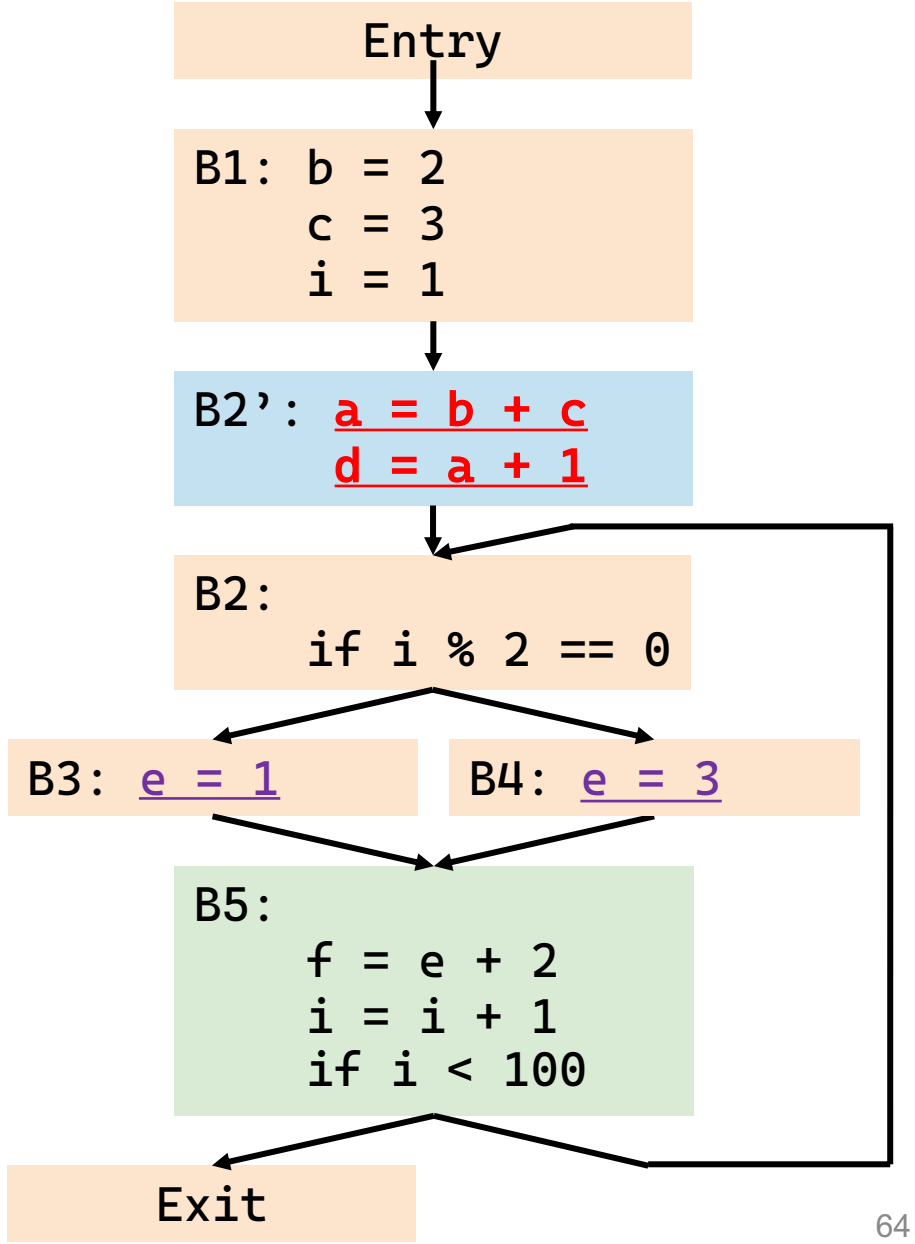
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- **a = b + c** инвариантный код
  и его можно вынести в предзаголовок **B2’**
  - **b** и **c** определяются вне цикла
  - **a** определяется в блоке **B2**,
    который доминирует над единственным
    выходом из цикла – блоком **B5**
- **d = a + 1** инвариантный код
  и его можно вынести в предзаголовок **B2’**
  - **a** определяется внутри цикла только
    в блоке **B2**, который доминирует
    над единственным
    выходом из цикла – блоком **B5**



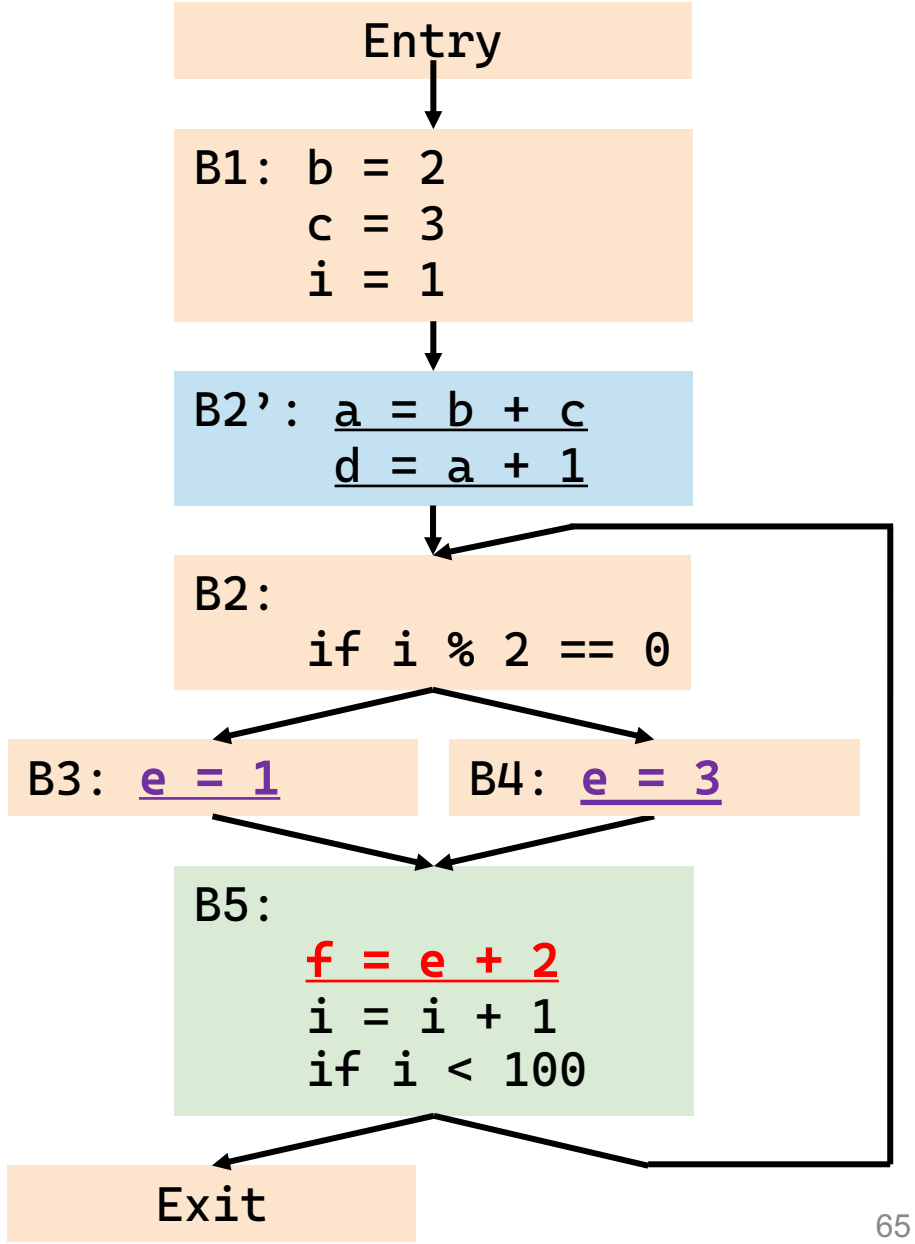
Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- **a = b + c** инвариантный код
  и его можно вынести в предзаголовок **B2'**
  - **b** и **c** определяются вне цикла
  - **a** определяется в блоке **B2**,
    который доминирует над единственным
    выходом из цикла – блоком **B5**
- **d = a + 1** инвариантный код
  и его можно вынести в предзаголовок **B2'**
  - **a** определяется внутри цикла только
    в блоке **B2**, который доминирует
    над единственным
    выходом из цикла – блоком **B5**



Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

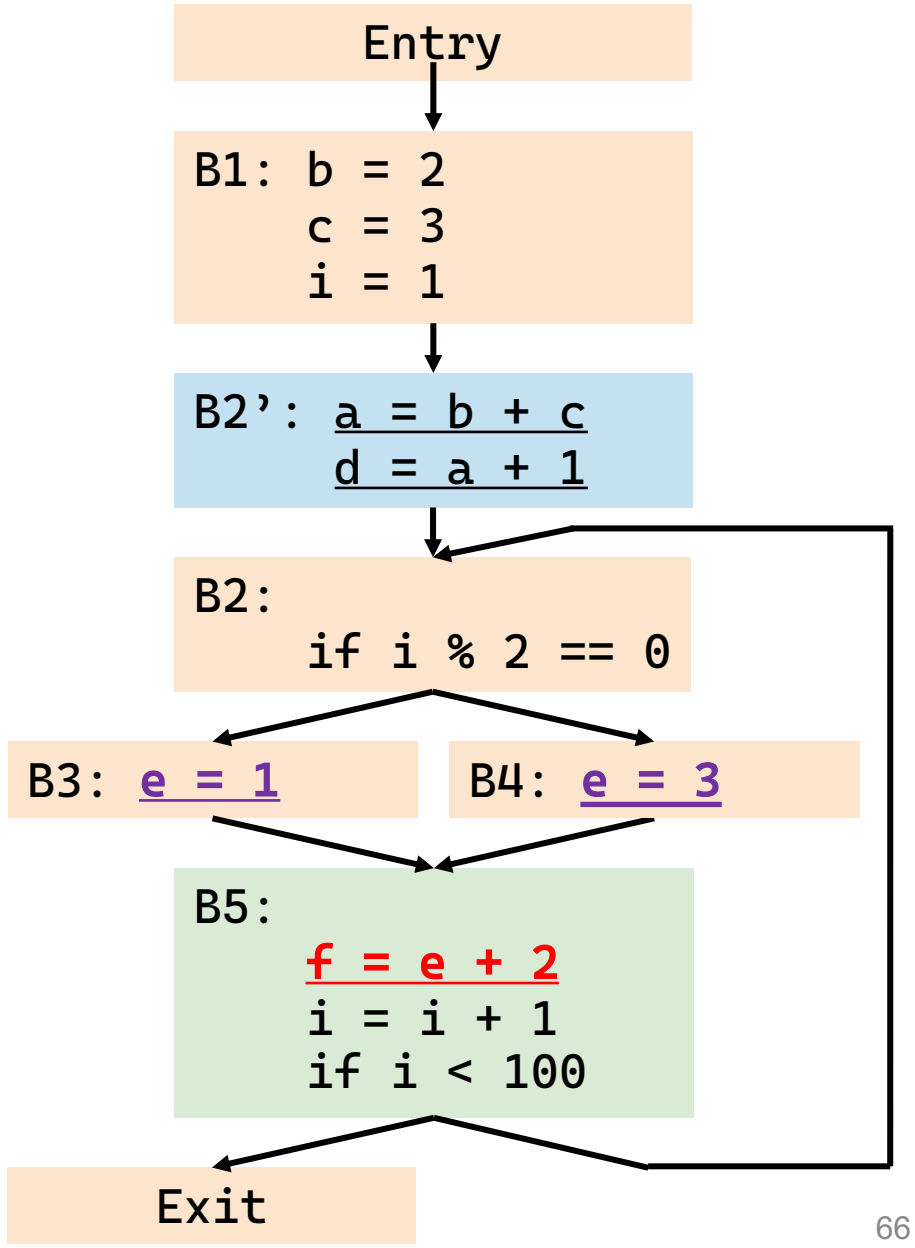
- Инструкции **e = 2** и **e = 3** не инвариантны относительно цикла, так как выполняются в блоках, не являющихся доминаторами выхода из цикла – блока **B5**
- Инструкция **f = e + 2** не инвариантна относительно цикла, так как **e** может изменяться во время выполнения цикла
- Эти инструкции нельзя выносить в предзаголовок **B2’**



Перемещение кода, инвариантного относительно цикла. Код, инвариантный относительно цикла. Пример 2

- Таким образом выражение присваивания **x = e**, где **e** – инвариант цикла, можно выносить в предзаголовок 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



Перемещение кода, инвариантного относительно цикла. Алгоритм перемещения инвариантного кода

1. Вставить пустой базовый блок (будущий предзаголовок) перед заголовком цикла
   - На каждом шаге формируется последовательность инструкций определенных как инвариант цикла
   - Для всех инструкций в теле цикла выполнить шаги:
2. Отметить как инвариантные все операнды-константы
3. Отметить как инвариантные операнды, **все** достигающие определения которых расположены за границами цикла
4. Отметить как инвариантные все инструкции, все операнды которых помечены как инвариант цикла, а также определяемые такими инструкциями переменные «**мертвы**» на входе в цикл
5. Шаги 2-4 повторяются до тех пор, пока будут добавляться новые операнды и инструкции, помеченные как инвариант цикла
6. Переместить все выделенные инструкции в предзаголовок

Другие оптимизации циклов
Обзор оптимизирующих преобразований циклов

- Вынесение инвариантного кода в предзаголовок (уже рассмотрели)
- Исключение умножений при вычислении индуктивных переменных (замена **`t = a * i + c`**, где **`c`** и **`a`** – инварианты цикла, в частности – константы, на **`t = 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)

Индуктивные переменные.
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 = 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 каждой итерации 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 удается обнаружить индуктивные переменные, становится возможным выполнить различные оптимизации этого цикла

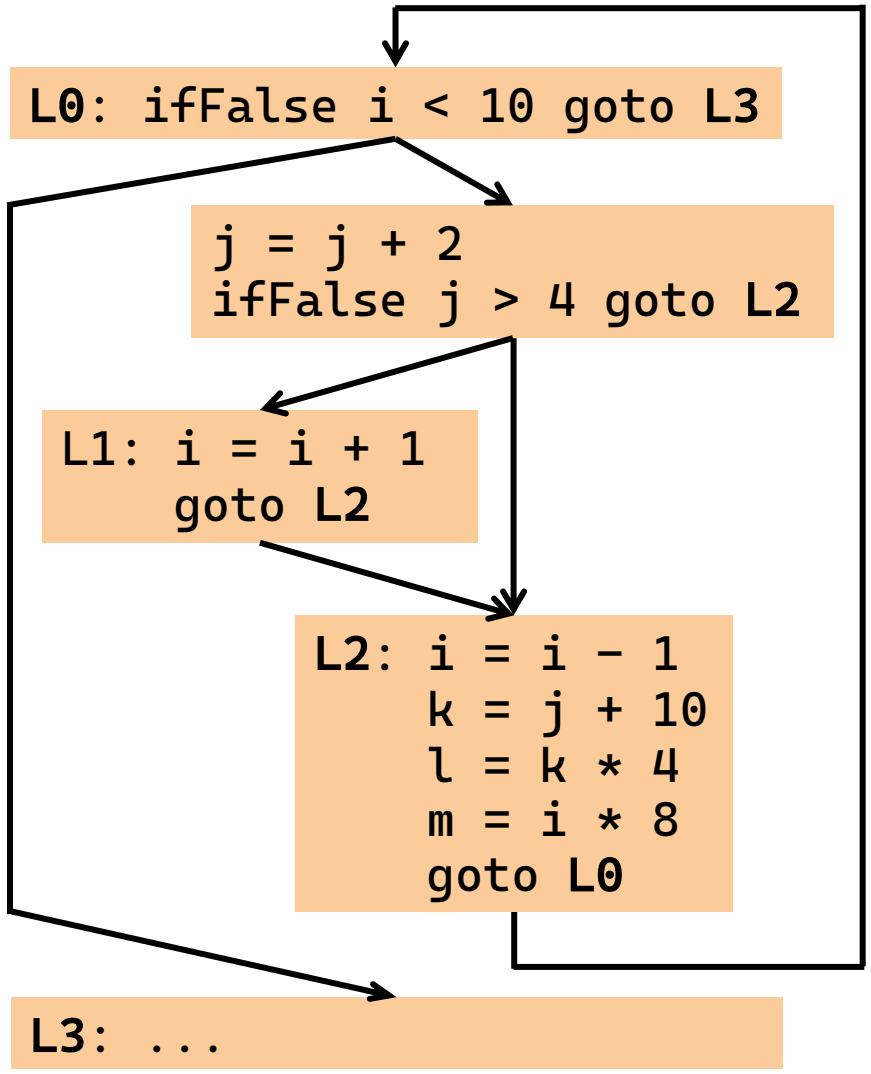
Индуктивные переменные. Семейства индуктивных переменных

- *Линейные функции от индуктивных переменных* также являются индуктивными переменными
- Следовательно, в циклах, могут встречаться *семейства индуктивных* переменных. Обычно в циклах бывает несколько таких семейств
- *Основной индуктивной переменной* по определению является индуктивная переменная **`i`**, все определения которой эквивалентны определению вида **`i = i + c`**, где **`c`** – инвариант цикла (как правило, константа)
  - значение **`c`** необязательно должно быть 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* **`j`** выражается через одну из основных индуктивных переменных **`i`** как **`c * i + d`**, где **`c`** и **`d`** – инварианты цикла

Индуктивные переменные. Семейства индуктивных переменных

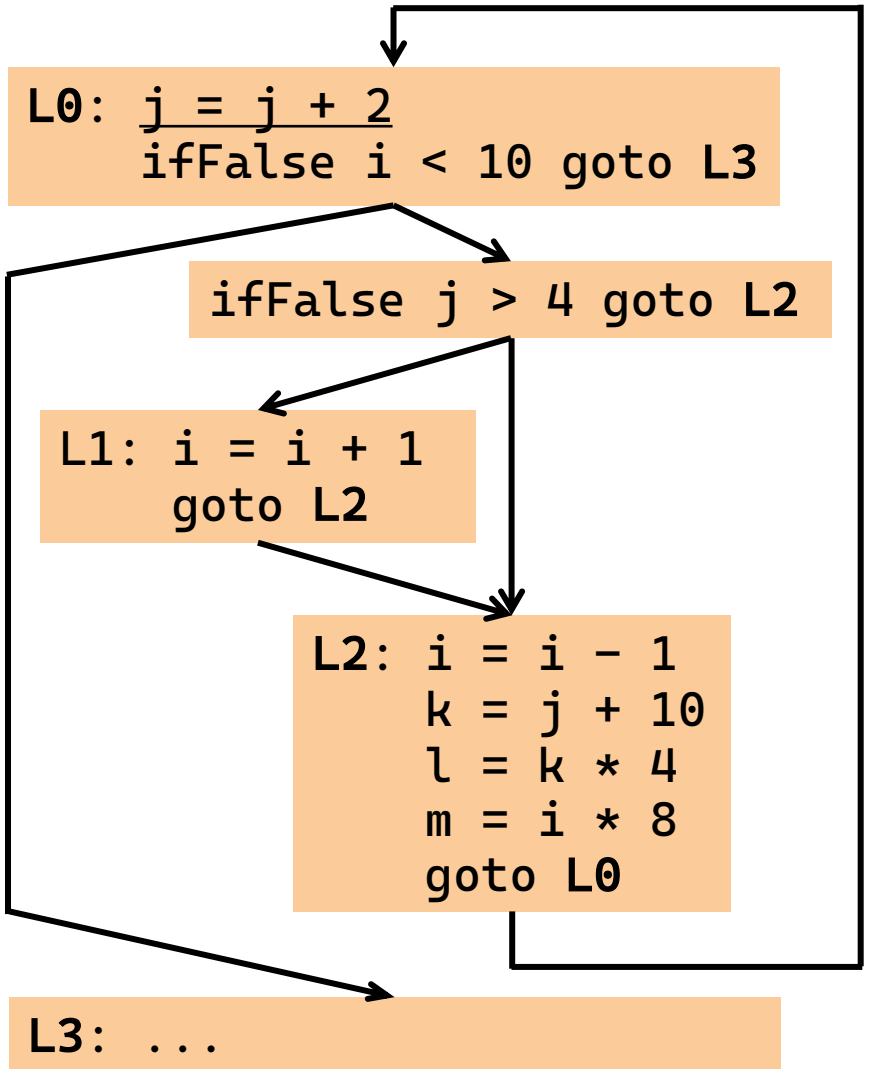
- Основная индуктивная переменная **`i`** и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*
- Для *производной индуктивной переменной* **`j`** удобно использовать обозначение **`j = ⟨i, c, d⟩`**
- Применяя это обозначение к основной индуктивной переменной **`i`**, получим **`i = ⟨i, 1, 0⟩`**

Индуктивные переменные. Семейства индуктивных переменных. Пример



- **i** и **j** – основные индуктивные переменные
  - т.к. определения переменных имеют вид:
    i = i + c
- **k** и **l** – линейные производные индуктивные переменные (семейства **j**)
- **m** – производная индуктивная переменная (семейства **i**)

Индуктивные переменные. Семейства индуктивных переменных. Пример



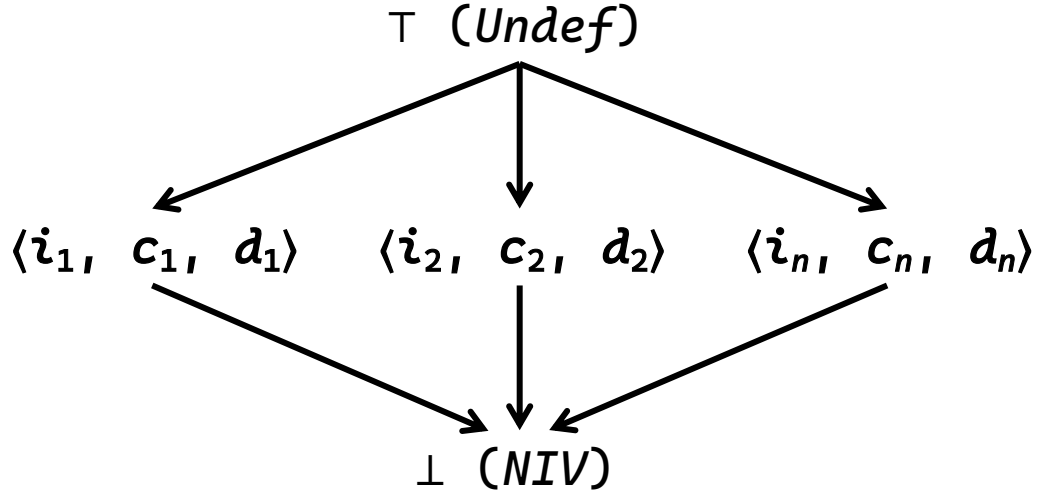
- **i** – основная индуктивная переменная
  - т.к. определение переменной имеет вид: **i = i + c**
- **j** – линейная основная и индуктивная переменная
  - т.к. присутствует только одно определение в базовом блоке, доминирующим все базовые блоки цикла
- **k** и **l** – линейные производные индуктивные переменные (семейства **j**)
- **m** – производная индуктивная переменная (семейства **i**)

Индуктивные переменные. Обнаружение индуктивных переменных

- Задача обнаружения индуктивных переменных может быть сформулирована как задача анализа потока данных в пределах цикла
  (необходимо задать направление анализа, определить область определения и правила работы оператора сбора, а также сформировать семейство передаточных функций)
- Каждая переменная программы отображается на элемент области определения (элемент полурешетки), определяющий состояние переменной (индуктивная переменная или неиндуктивная)
- Значение потока данных – отображение m каждой переменный v_i
  на элемент полурешетки l_i (аналогично задаче распространения констант)
- Значение переменной v в отображении m обозначается как $m(v)$

Индуктивные переменные.
Обнаружение индуктивных переменных

- Область определения содержит множество «троек» $\langle i_k, c_k, d_k \rangle$, описывающих индуктивные переменные (элементы полурешетки)
- Верхний элемент (⊤) полурешетки соответствует неопределенному состоянию переменной (неизвестно, индуктивная или неиндуктивная – *Undef*)
- Нижний элемент (⊥) полурешетки соответствует состоянию неиндуктивная переменная (*Not Induction Variable – NIV*)



- Если m_1 и m_2 отображают переменную v на значения l_1 и l_2, то $m_1 \wedge m_2$ отображают переменную v на значение $l_1 \wedge l_2$
- Например:

$$\{i \to \langle i,1,0\rangle, j \to \langle i,2,4\rangle, k \to \top\} \wedge$$
$$\{i \to \langle i,1,0\rangle, j \to \langle i,4,4\rangle, k \to \langle i,1,1\rangle\} =$$
$$= \{i \to \langle i,1,0\rangle, j \to \bot, k \to \langle i,1,1\rangle\}$$

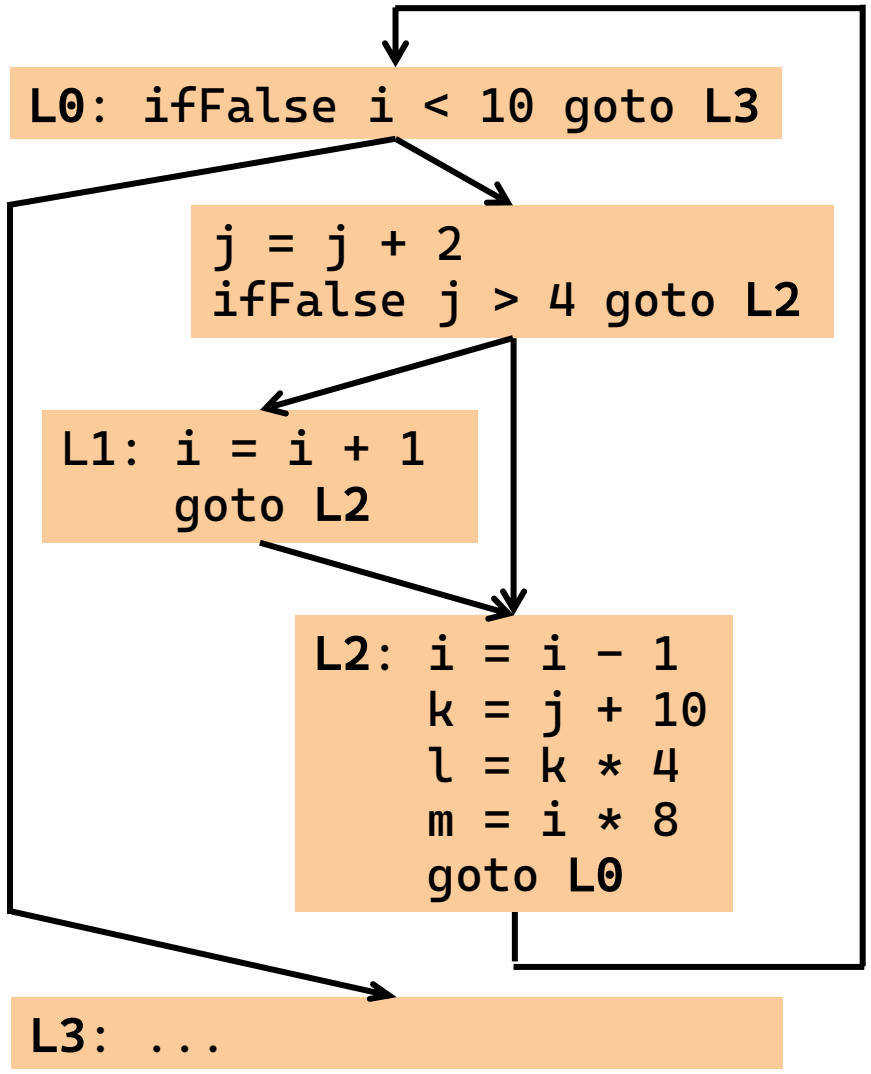
Индуктивные переменные. Обнаружение индуктивных переменных

- **Семейство передаточных функций**. Пусть f_s передаточная функция инструкции **s**, и $m` = f_s(m)$. Определим $f_s$ в терминах отношений между $m`$ и m
  - если **s** не является инструкцией присваивания, то f_s – тождественная функция
  - если **s** инструкция присваивания вида **x = i +/-/* c**
    - для всех переменных **k != x**:
      - $m`(k) = m(k)$ если **k** не является производной индуктивной переменной семейства **x**
      - $m`(k) =$ **⟨x, a, b – a * c⟩** если инструкция **s** имеет вид **x = x + c** и **k** производная индуктивная переменная семейства **x**: $m(k) =$ **⟨x, a, b⟩**
    - для всех переменных **k == x**:
      - $m`(k) =$ **⟨k, 1, 0⟩** если **k** основная индуктивная переменная
      - абстрактная интерпретация инструкции s если **k** не основная индуктивная переменная
- Правила абстрактной интерпретации инструкции **s**
  - если **j** индуктивная переменная $m(j) =$ **⟨i, a, b⟩**:
    - **x = j +(–) c** => $m`(x) =$ **⟨i, a, b +(–) c⟩**
    - **x = j * c** => $m`(x) =$ **⟨i, a * c, b * c⟩**
  - если **j, k** индуктивные переменные $m(j) =$ **⟨i, a, b⟩** и $m(k) =$ **⟨i, c, d⟩**:
    - **x = j +(–) k** => $m`(x) =$ **⟨i, a +(–) c, b +(–) d⟩**
    - $m`(x) = \perp$ во всех остальных случаях

Индуктивные переменные. Обнаружение индуктивных переменных

- Анализ потоков данных выполняется только над телом цикла
- **Шаг 0.** Перед запуском итеративного алгоритма необходимо найти все основные индуктивные переменные
- Таким образом, начальное значение потока данных будет инициализировано следующим образом:
  - $m(k) =$ **⟨k, 1, 0⟩** если **k** основная индуктивная переменная (определение вида **k = k + c**)
  - $m(k) = \top$ для всех переменных, не являющихся основными индуктивными переменными
- **Шаг 1**. Решение системы уравнений итеративным алгоритмом
- В процессе обработки базовых блоков цикла алгоритм сканирует каждую инструкцию базового блока и применяет передаточную функцию на каждой инструкции
- Оператор сбора применяется в точках слияния путей выполнения на входе в базовый блок
- В результате анализа будет получено несколько семейств индуктивных переменных и станут возможными оптимизации, связанные с индуктивными переменными – снижение сложности операций и исключение индуктивных переменных

Индуктивные переменные.
Семейства индуктивных переменных. Пример



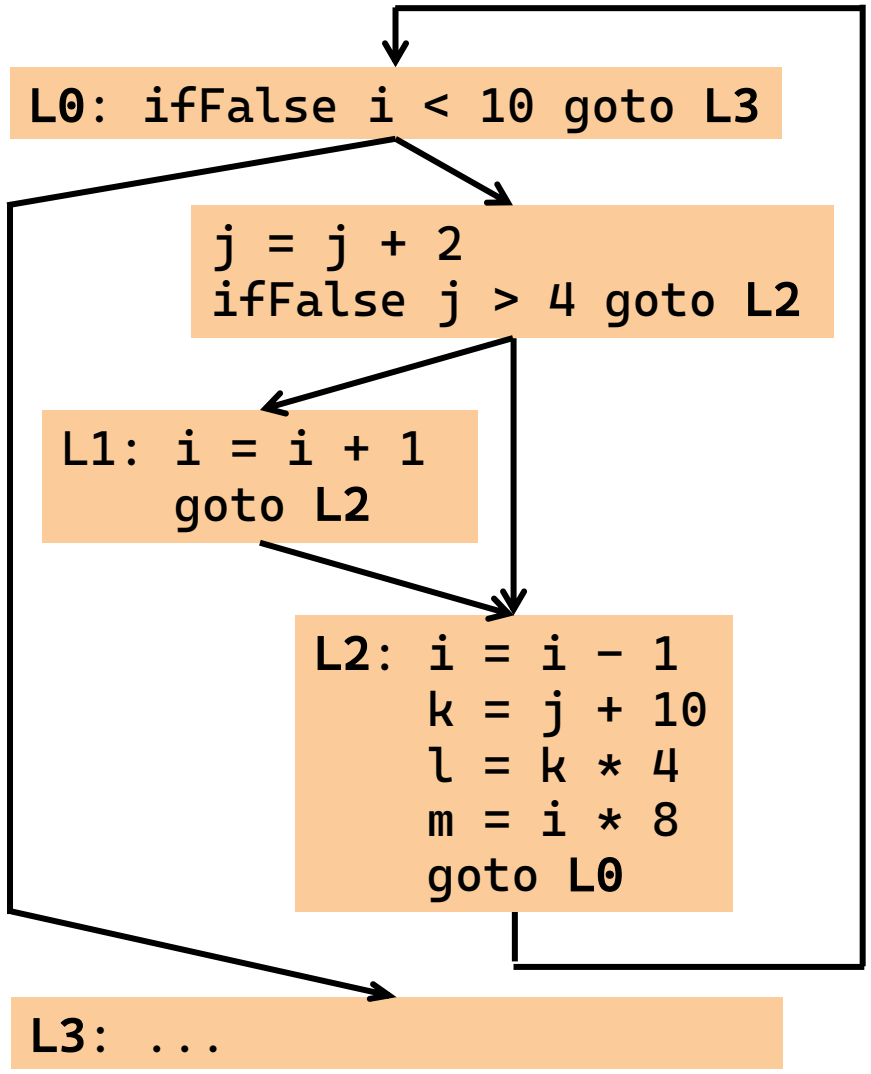
Шаг 0. Поиск базовых индуктивных переменных

`M: {j -> ⟨j, 1, 0⟩, i -> ⟨i, 1, 0⟩}`

Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
M:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> T
l -> T
m -> T
}
```

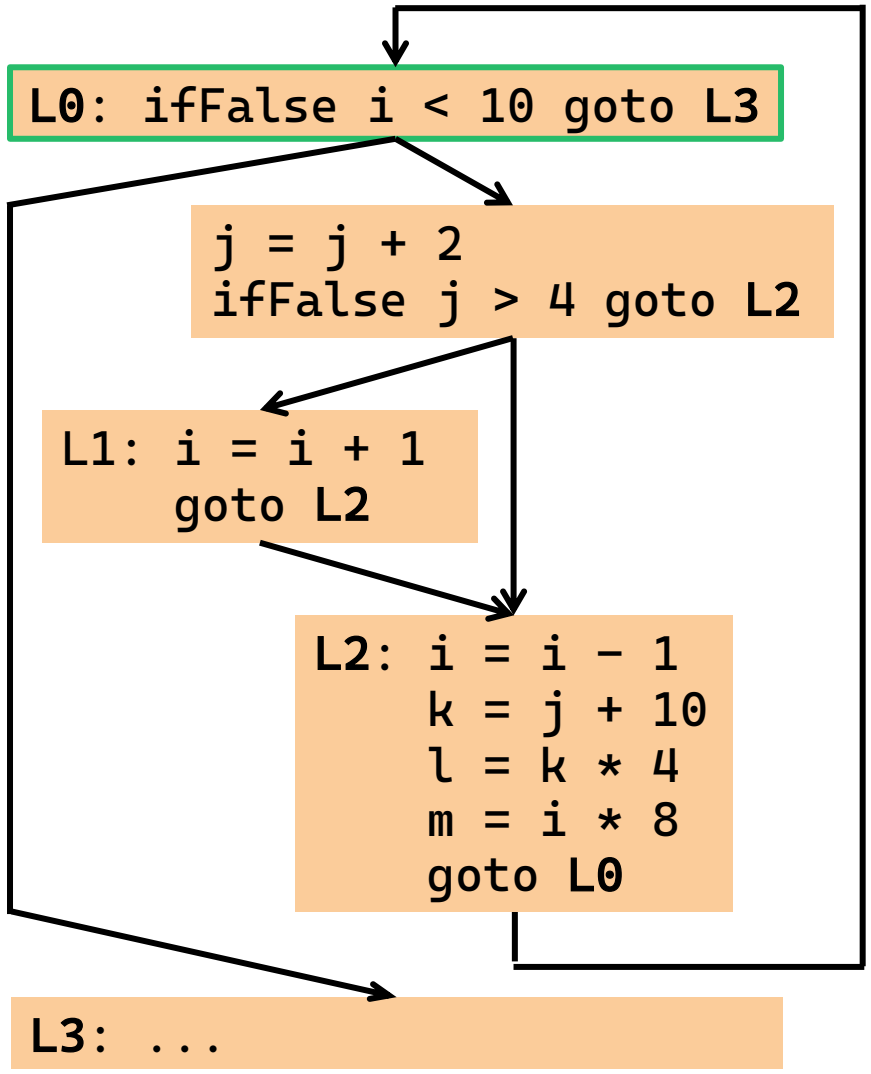
Индуктивные переменные.
Семейства индуктивных переменных. Пример



Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
M:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> T
l -> T
m -> T
}
```

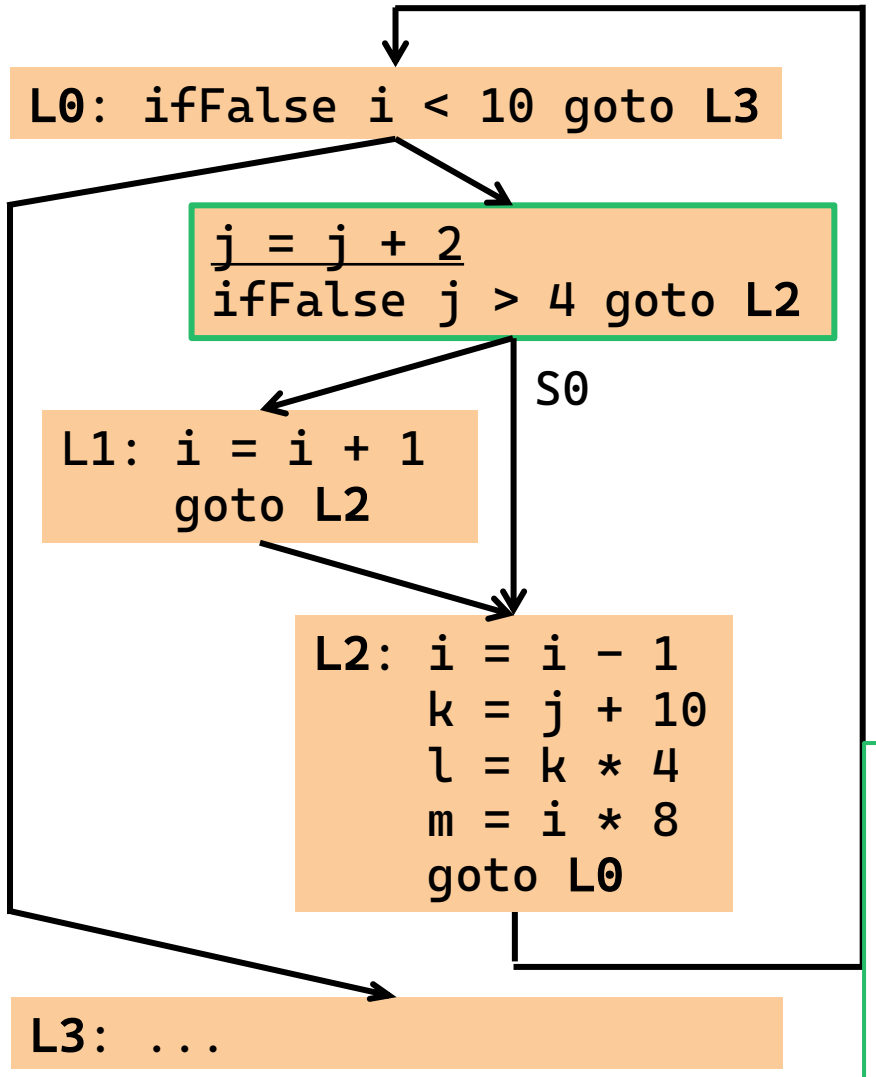
Индуктивные переменные.
Семейства индуктивных переменных. Пример



Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
M:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> T
l -> T
m -> T
}
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример

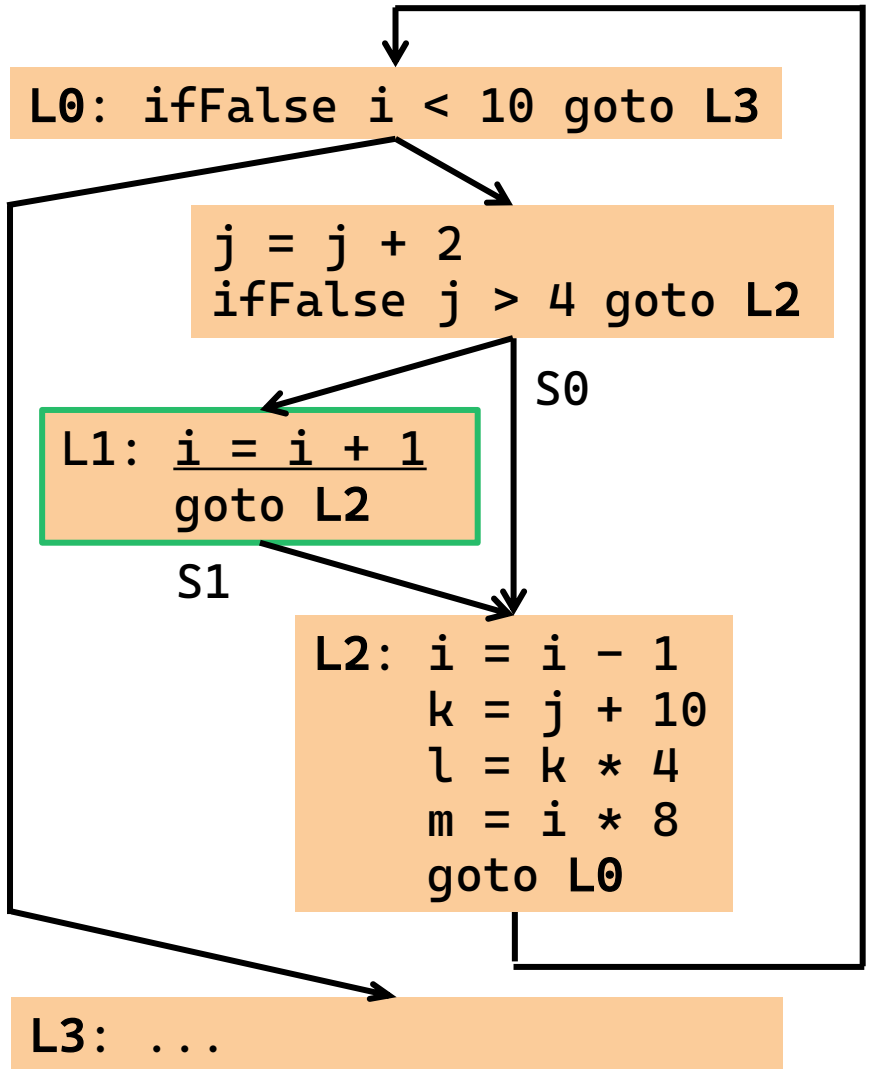



```
S0:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> T
l -> T
m -> T
}
```

Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
j = j + 2 // основная индуктивная переменная =>
          // по правилу работы передаточной функции
          // j не изменяется, изменяются только все
          // производные индуктивные переменные
          // семейства j (на текущий момент таких нет)

S0 – состояние на выходе из базового блока
```

Индуктивные переменные. Семейства индуктивных переменных. Пример

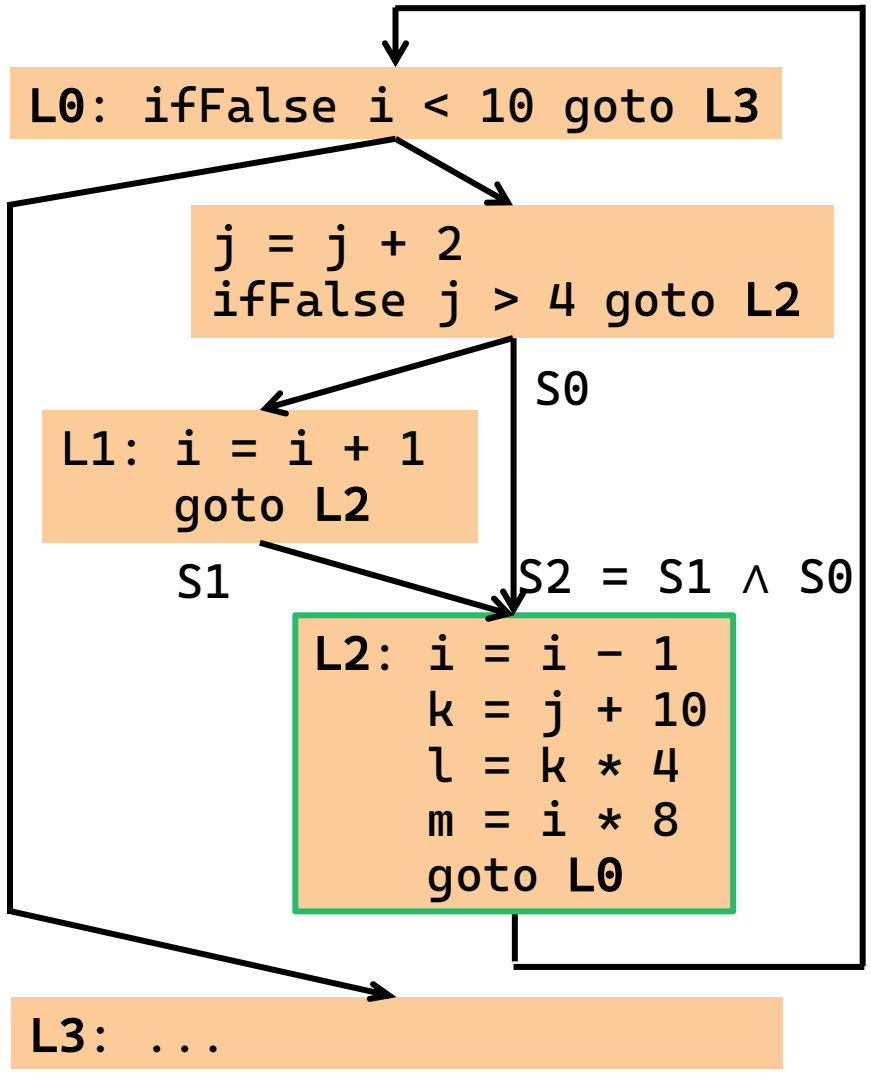


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
i = i + 1 // основная индуктивная переменная =>
          // по правилу работы передаточной функции
          // i не изменяется, изменяются только все
          // производные индуктивные переменные
          // семейства i (на текущий момент таких нет)

S1 – состояние на выходе из базового блока
```

```
S0:{                S1:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> T              k -> T
l -> T              l -> T
m -> T              m -> T
}                   }
```

Индуктивные переменные. Семейства индуктивных переменных. Пример

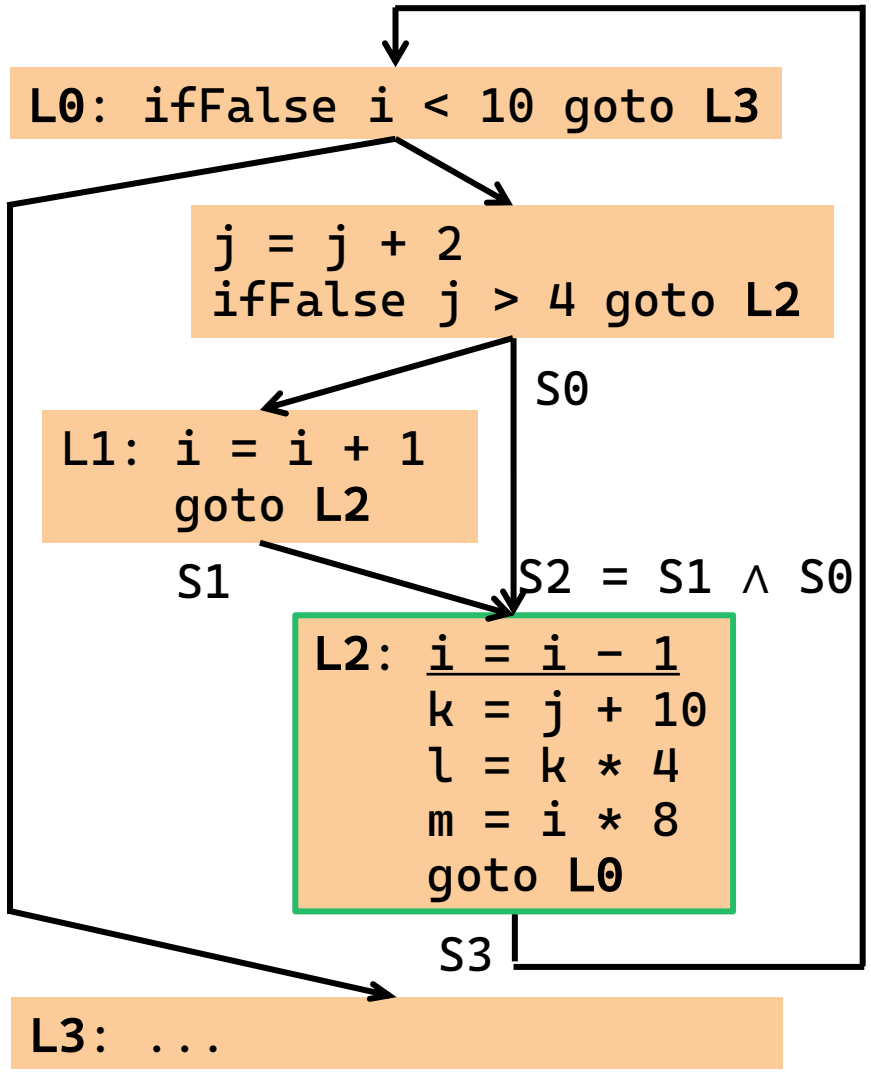


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S2 на входе в базовый блок
S2 = S1 ∧ S0 =
   {i -> ⟨i, 1, 0⟩, j -> ⟨j, 1, 0⟩, k->T, l->T, m->T}
   ∧
   {i -> ⟨i, 1, 0⟩, j -> ⟨j, 1, 0⟩, k->T, l->T, m->T}
   =
   {i -> ⟨i, 1, 0⟩, j -> ⟨j, 1, 0⟩, k->T, l->T, m->T}
```

```
S0:{                S1:{                S2:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> T              k -> T              k -> T
l -> T              l -> T              l -> T
m -> T              m -> T              m -> T
}                   }                   }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример

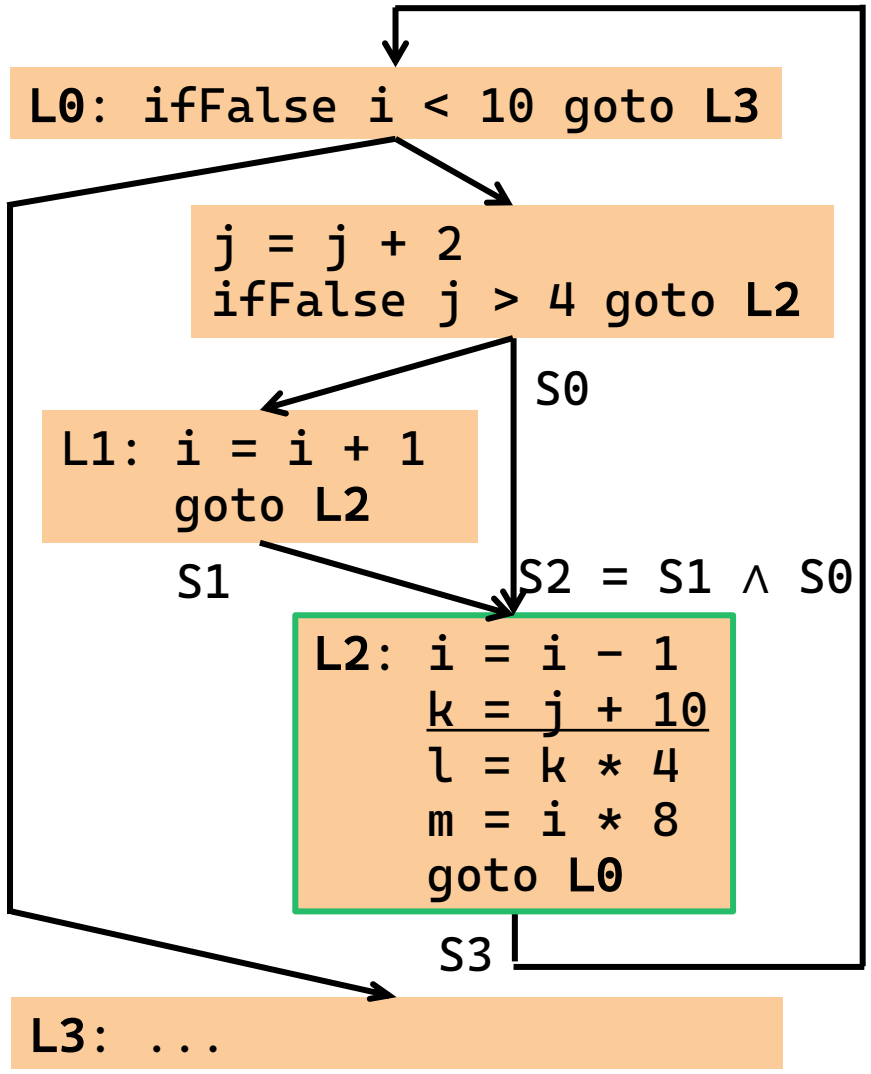


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S3 на выходе из базового блока
i = i + 1 // основная индуктивная переменная =>
          // по правилу работы передаточной функции
          // i не изменяется, изменяются только все
          // производные индуктивные переменные
          // семейства i (на текущий момент таких нет)
```

```
S0:{                S1:{                S2:{                S3:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> T              k -> T              k -> T              k -> T
l -> T              l -> T              l -> T              l -> T
m -> T              m -> T              m -> T              m -> T
}                   }                   }                   }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример

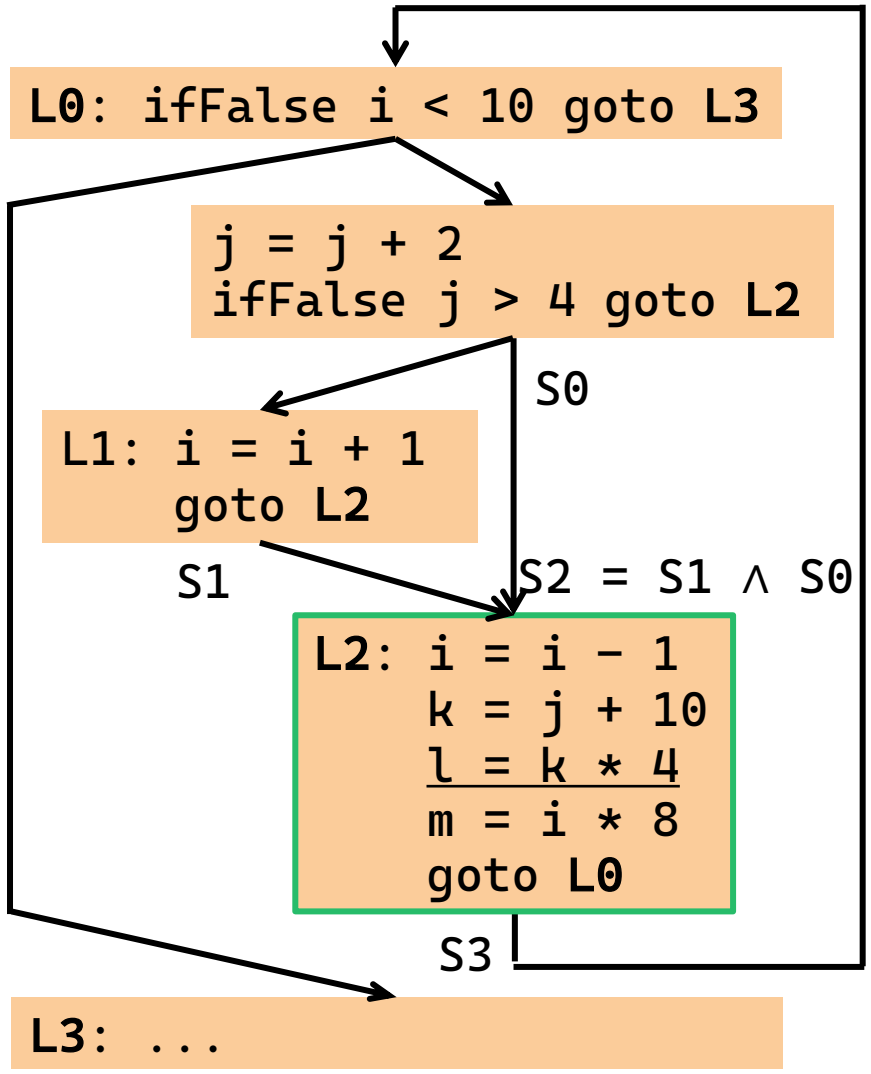


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S3 на выходе из базового блока
k = j + 10 // применяем правила
           // абстрактной интерпретации для +
k -> ⟨j, 1, 10⟩
```

```
S0:{                S1:{                S2:{                S3:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> T              k -> T              k -> T              k -> ⟨j, 1, 10⟩
l -> T              l -> T              l -> T              l -> T
m -> T              m -> T              m -> T              m -> T
}                   }                   }                   }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример

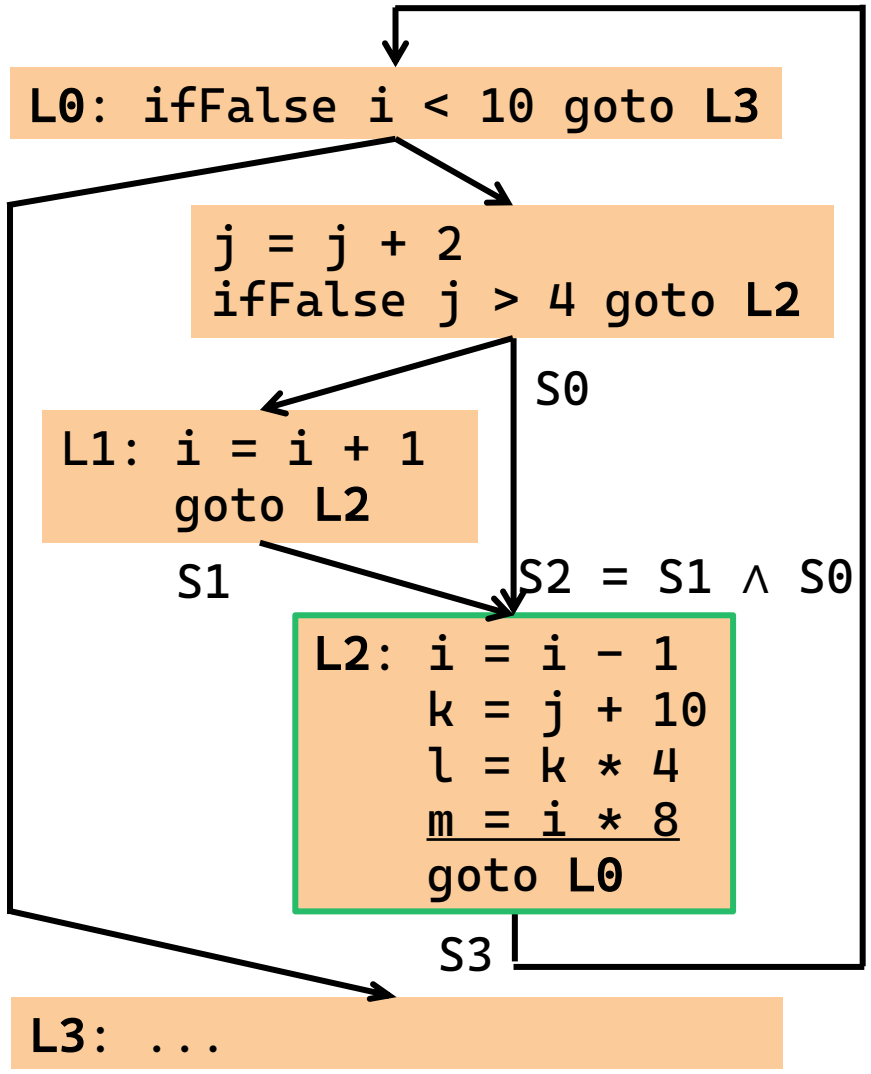


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S3 на выходе из базового блока
l = k * 4 // применяем правила
          // абстрактной интерпретации для *
l -> ⟨j, 4, 40⟩
```

```
S0:{              S1:{              S2:{              S3:{
i -> ⟨i, 1, 0⟩    i -> ⟨i, 1, 0⟩    i -> ⟨i, 1, 0⟩    i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩    j -> ⟨j, 1, 0⟩    j -> ⟨j, 1, 0⟩    j -> ⟨j, 1, 0⟩
k -> T            k -> T            k -> T            k -> ⟨j, 1, 10⟩
l -> T            l -> T            l -> T            l -> ⟨j, 4, 40⟩
m -> T            m -> T            m -> T            m -> T
}                 }                 }                 }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример

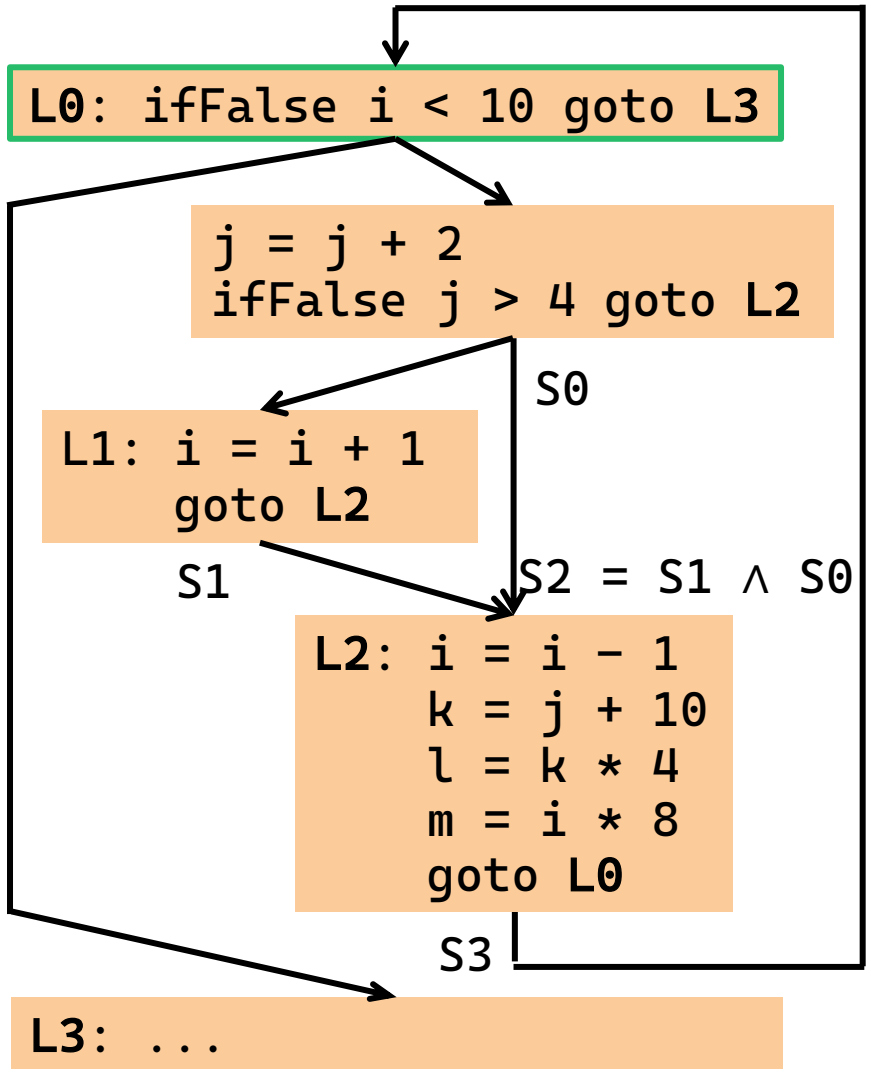


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S3 на выходе из базового блока
m = i * 8 // применяем правила
          // абстрактной интерпретации для *
m -> ⟨i, 8, 0⟩
```

```
S0:{                S1:{                S2:{                S3:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> T              k -> T              k -> T              k -> ⟨j, 1, 10⟩
l -> T              l -> T              l -> T              l -> ⟨j, 4, 40⟩
m -> T              m -> T              m -> T              m -> ⟨i, 8, 0⟩
}                   }                   }                   }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример

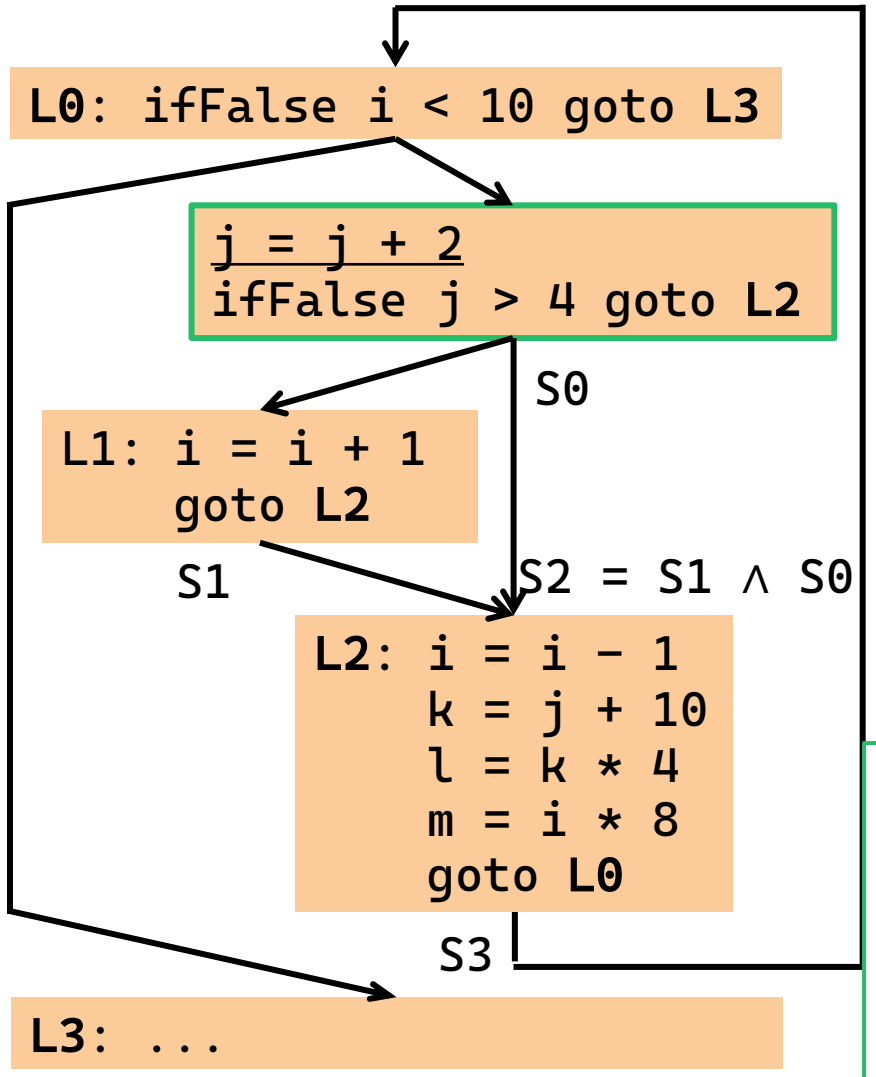


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Состояние на выходе из базового блока = S3
```

```
S0:{                S1:{                S2:{                S3:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> T              k -> T              k -> T              k -> ⟨j, 1, 10⟩
l -> T              l -> T              l -> T              l -> ⟨j, 4, 40⟩
m -> T              m -> T              m -> T              m -> ⟨i, 8, 0⟩
}                   }                   }                   }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример



Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Обновим состояние S0
j = j + 2 // основная индуктивная переменная =>
          // по правилу работы передаточной функции
          // j не изменяется, изменяются только все
          // производные индуктивные переменные
          // семейства j
m'(k) = ⟨j, 1, 10 - 1 * 2⟩ = ⟨j, 1, 8⟩
m'(l) = ⟨j, 4, 40 - 4 * 2⟩ = ⟨j, 4, 32⟩
```

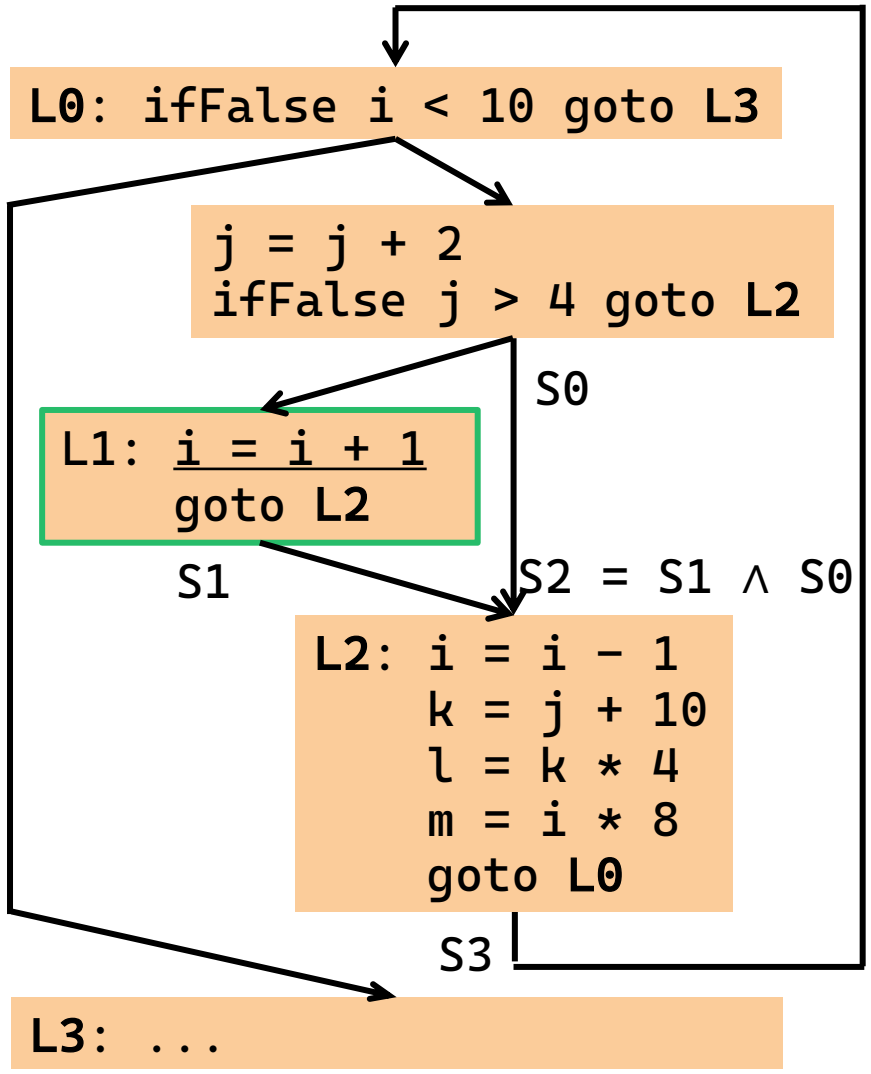
```
S0:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> ⟨j, 1, 8⟩
l ->⟨j, 4, 32⟩
m -> T
}
```

```
S1:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> T
l -> T
m -> T
}
```

```
S2:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> T
l -> T
m -> T
}
```

```
S3:{
i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩
k -> ⟨j, 1, 10⟩
l -> ⟨j, 4, 40⟩
m -> ⟨i, 8, 0⟩
}
```

Индуктивные переменные. Семейства индуктивных переменных. Пример

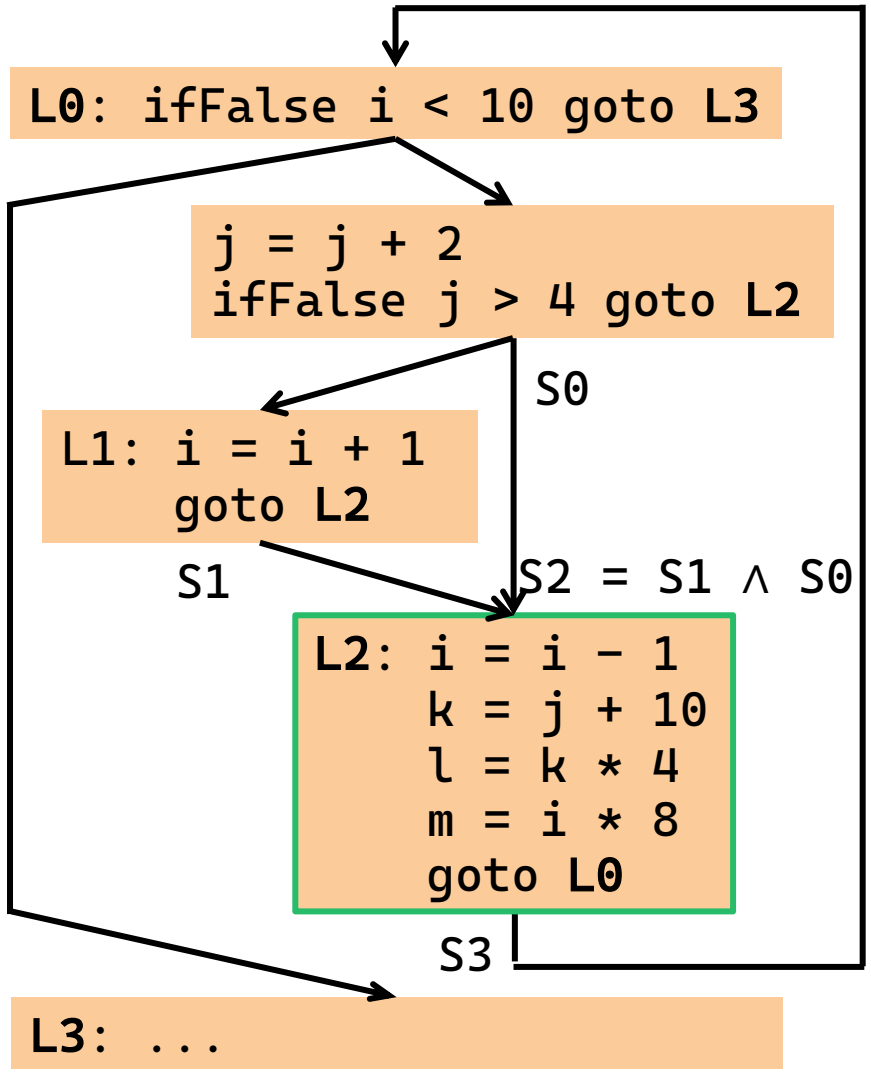


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Обновим состояние S1
i = i + 1 // основная индуктивная переменная =>
          // по правилу работы передаточной функции
          // i не изменяется, изменяются только все
          // производные индуктивные переменные
          // семейства i
m'(m) = ⟨i, 8, 0 - 8 * 1⟩ = ⟨i, 8, -8⟩
```

```
S0:{                S1:{                S2:{                S3:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> ⟨j, 1, 8⟩      k -> ⟨j, 1, 8⟩      k -> T              k -> ⟨j, 1, 10⟩
l ->⟨j, 4, 32⟩      l ->⟨j, 4, 32⟩      l -> T              l -> ⟨j, 4, 40⟩
m -> ⟨i, 8, 0⟩      m ->⟨i, 8, -8⟩      m -> T              m -> ⟨i, 8, 0⟩
}                   }                   }                   }
```

Индуктивные переменные. Семейства индуктивных переменных. Пример

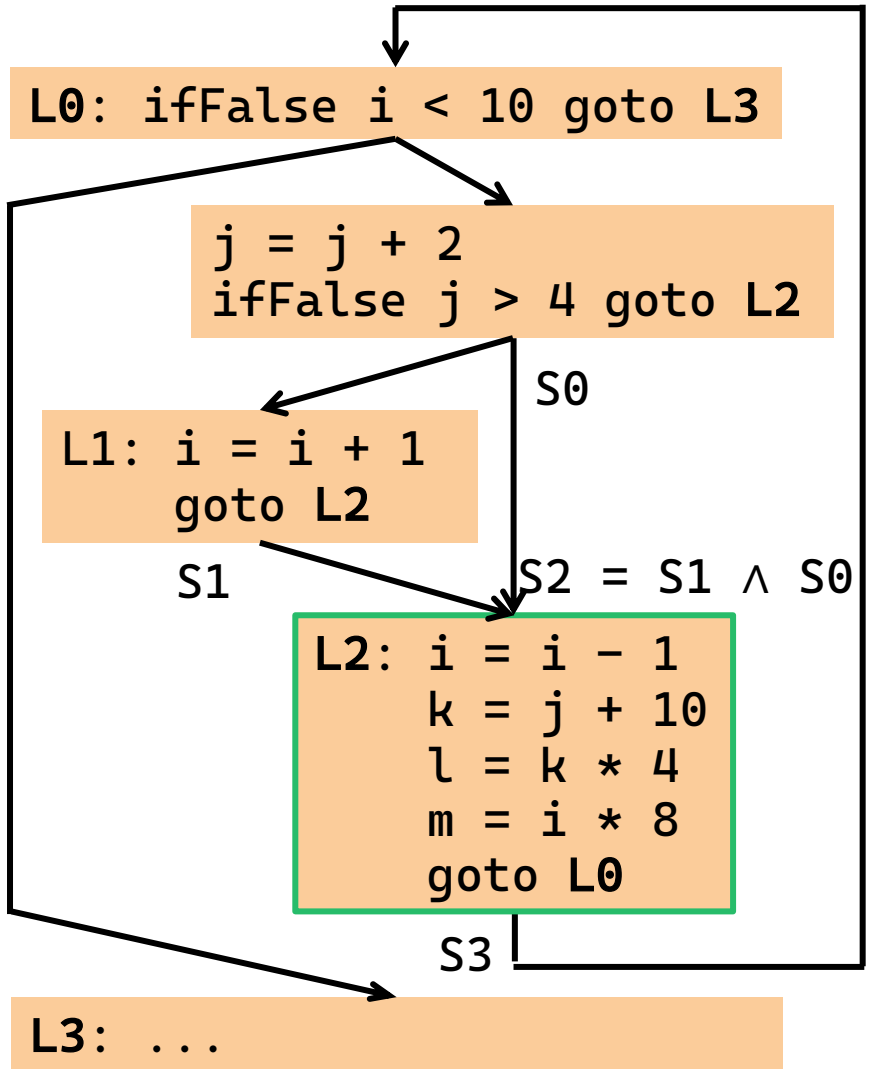


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S2 на входе в базовый блок
S2 = S1 ∧ S0 =
  {i->⟨i,1,0⟩,j->⟨j,1,0⟩,k->⟨j,1,8⟩,l->⟨j,4,32⟩,m->⟨i,8,-8⟩}
  ∧
  {i->⟨i,1,0⟩,j->⟨j,1,0⟩,k->⟨j,1,8⟩,l->⟨j,4,32⟩,m->⟨i,8,0⟩
  =
  {i->⟨i,1,0⟩,j->⟨j,1,0⟩,k->⟨j,1,8⟩,l->⟨j,4,32⟩,⊥}
```

S0:	S1:	S2:	S3:
i -> ⟨i, 1, 0⟩	i -> ⟨i, 1, 0⟩	i -> ⟨i, 1, 0⟩	i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩	j -> ⟨j, 1, 0⟩	j -> ⟨j, 1, 0⟩	j -> ⟨j, 1, 0⟩
k -> ⟨j, 1, 8⟩	k -> ⟨j, 1, 8⟩	k -> ⟨j, 1, 8⟩	k -> ⟨j, 1, 10⟩
l ->⟨j, 4, 32⟩	l ->⟨j, 4, 32⟩	l ->⟨j, 4, 32⟩	l -> ⟨j, 4, 40⟩
m -> ⟨i, 8, 0⟩	m ->⟨i, 8, -8⟩	m -> ⊥	m -> ⟨i, 8, 0⟩
}	}	}	}

Индуктивные переменные.
Семейства индуктивных переменных. Пример

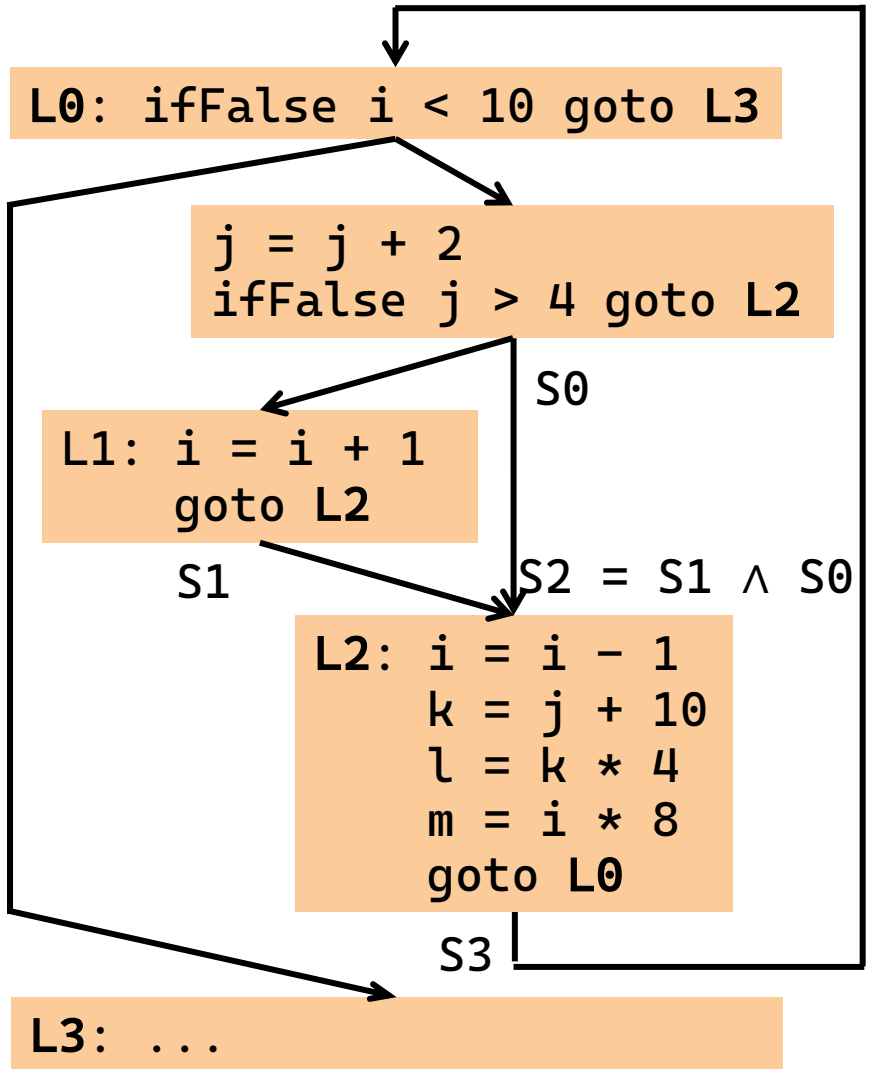


Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S3 на выходе из базового блока
// Значения S3 не изменяются
```

```
S0:{                S1:{                S2:{                S3:{
i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩      i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩      j -> ⟨j, 1, 0⟩
k -> ⟨j, 1, 8⟩      k -> ⟨j, 1, 8⟩      k -> ⟨j, 1, 8⟩      k -> ⟨j, 1, 10⟩
l ->⟨j, 4, 32⟩      l ->⟨j, 4, 32⟩      l ->⟨j, 4, 32⟩      l -> ⟨j, 4, 40⟩
m -> ⟨i, 8, 0⟩      m ->⟨i, 8, -8⟩      m -> ⊥              m -> ⟨i, 8, 0⟩
}                   }                   }                   }
```

Индуктивные переменные.
Семейства индуктивных переменных. Пример



Шаг 1. Запуск итеративного алгоритма и поиск максимальной фиксированной точки

```
// Вычислим состояние S3 на выходе из базового блока
// Значения S3 не изменяются
```

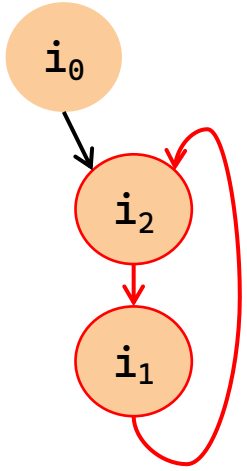
В условиях отсутствия SSA-формы необходимо учитывать *линейность основной индуктивной переменной*

Так как основная индуктивная переменная **i** не является линейной, то *производная индуктивная переменная* **m** на выходе из базового блока **L2** не может быть использована, например, при снижении сложности операций

```
S0:{            S1:{            S2:{            S3:{
i -> ⟨i, 1, 0⟩  i -> ⟨i, 1, 0⟩  i -> ⟨i, 1, 0⟩  i -> ⟨i, 1, 0⟩
j -> ⟨j, 1, 0⟩  j -> ⟨j, 1, 0⟩  j -> ⟨j, 1, 0⟩  j -> ⟨j, 1, 0⟩
k -> ⟨j, 1, 8⟩  k -> ⟨j, 1, 8⟩  k -> ⟨j, 1, 8⟩  k -> ⟨j, 1, 10⟩
l ->⟨j, 4, 32⟩  l ->⟨j, 4, 32⟩  l ->⟨j, 4, 32⟩  l -> ⟨j, 4, 40⟩
m -> ⟨i, 8, 0⟩  m ->⟨i, 8, -8⟩  m -> ⊥           m -> ⟨i, 8, 0⟩
}               }               }               }
```

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме

- Анализ индуктивных переменных существенно проще проводить над SSA-формой
- Поскольку каждая переменная определяется один раз, нет необходимости вычислять различные абстрактные значения на каждом ребре
- Оператор сбора необходимо применять только в φ-функциях
- Базовые индуктивные переменные определяются φ-функциями шаблоном следующего вида:
  $i_2 = \varphi(i_0, i_1)$
  $i_1 = i_2 + c$
- Такие шаблоны соответствуют компонентам сильной связности в SSA-графе (SSA-графе потока данных):



  - Анализ индуктивных переменных в SSA-форме выполняется над SSA-графом:
    - вершины в таком графе — инструкции, определяющие SSA-переменные
    - ребра задают отношение def-use
  - Заголовок цикла компоненты сильной связности в SSA-графе формирует базовую индуктивную переменную ($i_2 \rightarrow \langle i_2, 1, 0 \rangle$)

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме

- **Передаточная функция для φ-функций**.
  Пусть f_s передаточная функция инструкции **x = φ(x₀, ...xₙ)** и $m` = f_s(m)$.
  Определим f_s в терминах отношений между $m`$ и m
- К аргументам φ-функции применяется полурешеточный оператор сбора
  - $in = m(x_0) \wedge m(x_1) \wedge ... \wedge m(x_n)$;
- Если $in \neq \perp$
  - для всех переменных **k != x**: $m`(k) = m(k)$
  - если **x = φ(x₀, ...xₙ)** – заголовок цикла компоненты сильной связности в SSA-графе
    - $m`(x) =$ **⟨x, 1, 0⟩**,
  - если **x = φ(x₀, ...xₙ)** – **НЕ** заголовок цикла компоненты сильной связности в SSA-графе:
    - $m`(x) = in$,
- Если $in = \perp$
  - $m`(x) = \perp$
  - для всех переменных **k != x**: $m`(k) = \perp$
    если **k** производная индуктивная переменная семейства **x**: $m(k) =$ **⟨x, a, b⟩**

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме

- Рассмотрим алгоритм обнаружения индуктивных переменных в SSA-форме
- Алгоритм *выполняется над телом каждого цикла по отдельности*
- Анализ выполняется только для *естественных циклов*
- Компонента сильной связности определяет семейство индуктивных переменных
- *Основной индуктивной переменной* является *заголовок* компоненты сильной связности
- *Заголовок* компоненты сильной связности SSA-графа:
  - узел компоненты, представляющий φ-функцию
  - φ-функция расположена в базовом блоке графа потока управления, который является *заголовком естественного цикла*

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме

- **Алгоритм обнаружения индуктивных переменных в SSA-форме**
- Вход: **G** – SSA-граф цикла
- Выход: **M** – отображение SSA-переменных на индуктивные переменные: **v -> <i, a, b>**

```
find_IVs(G):
  foreach n in G:
    m(n) = T
  nextNum = 0
  foreach n in G:
    dfs(n);
```

- Узел SSA-графа описывается структурой:

{ Num, Visited, Low }

Num – количество ранее обработанных вершин

```
dfs(n):
  n.<Num, Visited, Low> = <nextNum, true, nextNum++>
  push(n)
  foreach o in getOperands(n):
    if o.Visited == false:
      dfs(o)
      n.Low = min(n.Low, o.Low)
    if o.Num < n.Num and isOnStack(0):
      n.Low = min(n.Low, o.Num)
  if n.Low == n.Num:
    SCC = {}
    do: x = pop(); SCC += {x}
    while x != n
    process(SCC)
```

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме

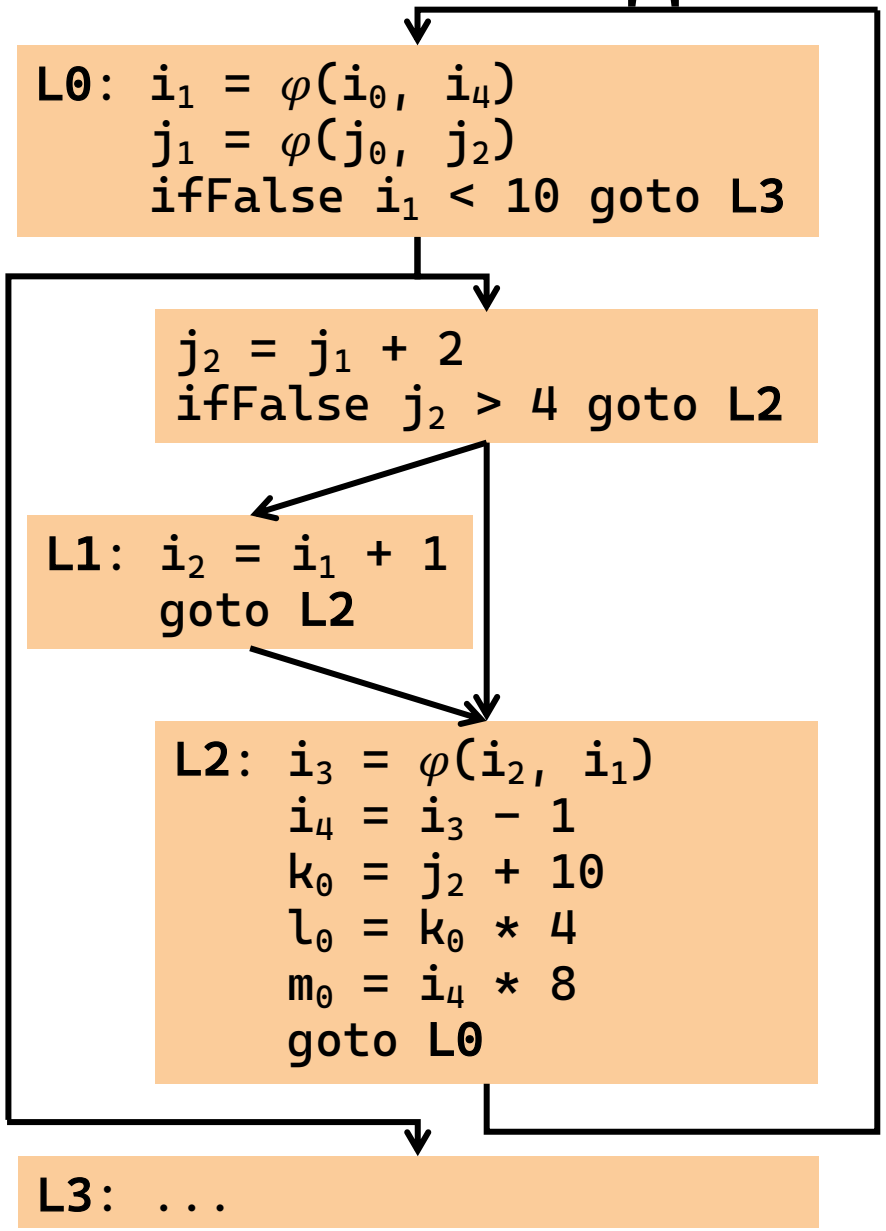
- **Алгоритм обнаружения индуктивных переменных в SSA-форме. Продолжение**

```
process(SCC):
  foreach n in SCC:
    if !isCandidate(n):
      return
  worklist = rpo(SCC)
  repeat
    n = take(worklist)
    oldVal = m(n)
    f(n) // передаточная функция
    if oldVal != m(n):
        worklist += succ(n)
  while empty(worklist) == false
```

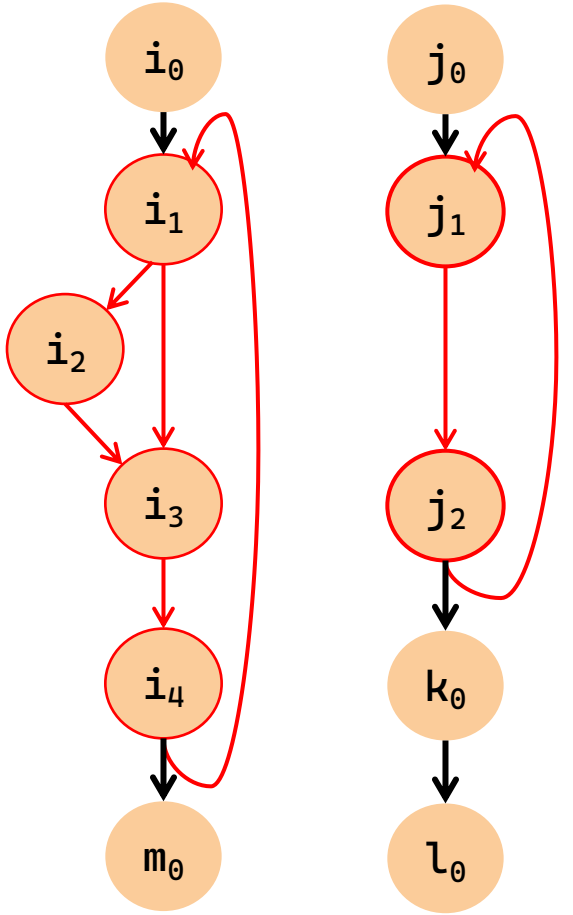
```
isCandidate(n):
  match n:
      x = c * i
    | x = i * c
    | x = c + i
    | x = i + c
    | x = i - c
    | x = φ(...)
    => return true
  return false
```

- Вспомогательные функции:
  - `rpo(SCC)` – возвращает список узлов компоненты сильной связности **SCC** в RPO порядке
  - `take(worklist)` – извлекает из начала списка один элемент и возвращает его
  - `succ(n)` – возвращает список последователей узла **n** SSA-графа

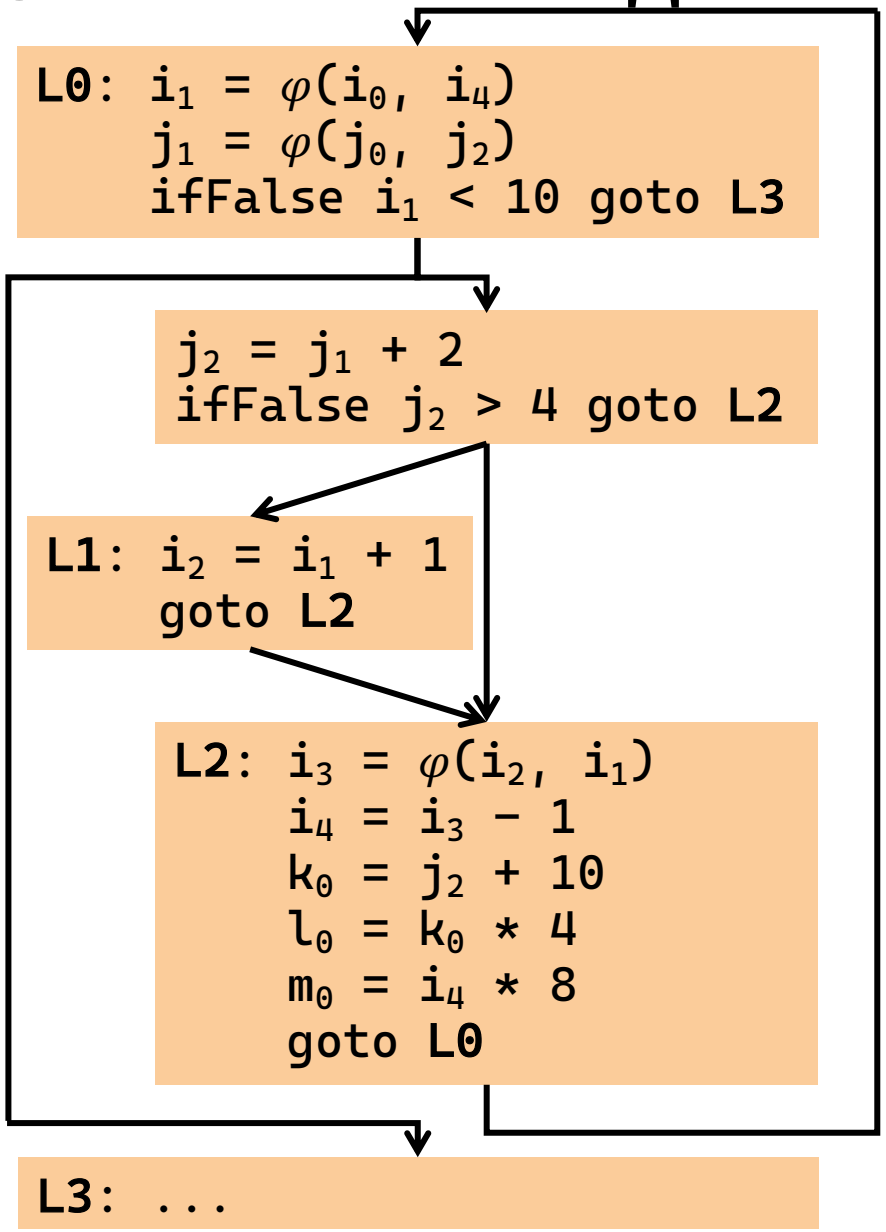
Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

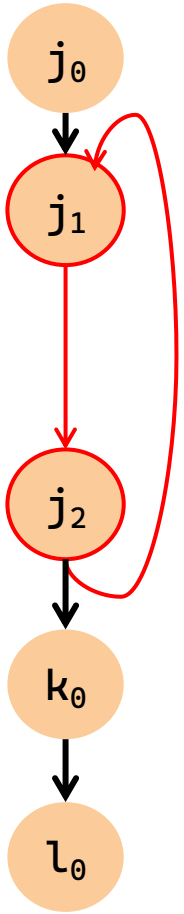



```
L0: i1 = φ(i0, i4)
    j1 = φ(j0, j2)
    ifFalse i1 < 10 goto L3

    j2 = j1 + 2
    ifFalse j2 > 4 goto L2

L1: i2 = i1 + 1
    goto L2

L2: i3 = φ(i2, i1)
    i4 = i3 - 1
    k0 = j2 + 10
    l0 = k0 * 4
    m0 = i4 * 8
    goto L0

L3: ...
```



Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

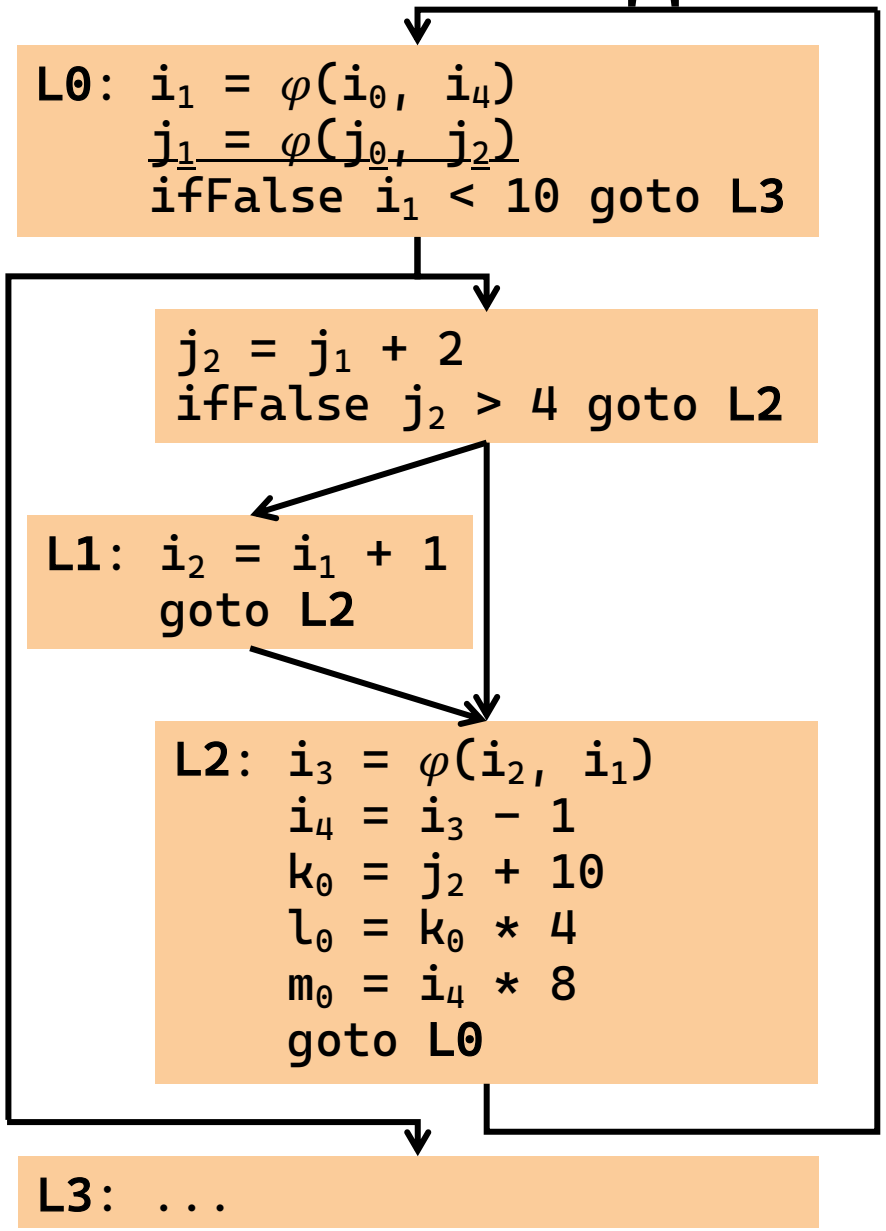


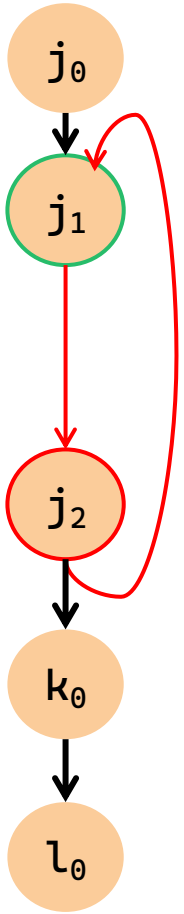


Worklist = {j_1, j_2}

M:{
j_1 -> T
j_2 -> T
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



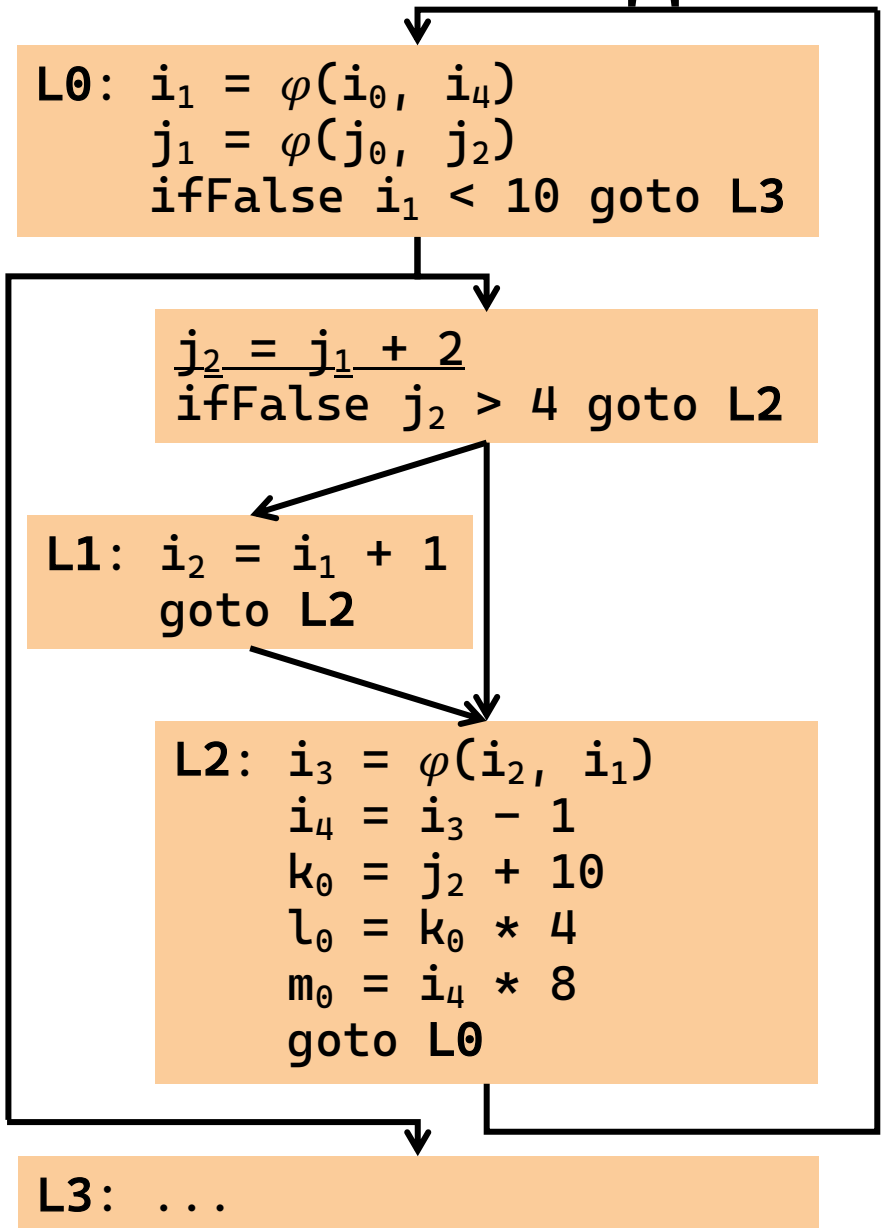


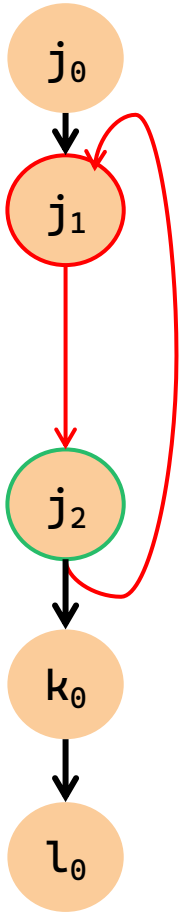
Worklist = {j_2}

n == j_1:
in = m(j_0) ∧ m(j_2) = ⊤ => in != ⊥
newVal = f(n) // j_1 – заголовок SCC, поэтому
m'(n) = ⟨j_1, 1, 0⟩ // основная индуктивная
// переменная
m'(n) != m(n) => Worklist += succ(j_1)

M:{
j_1 -> ⊤
j_2 -> ⊤
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



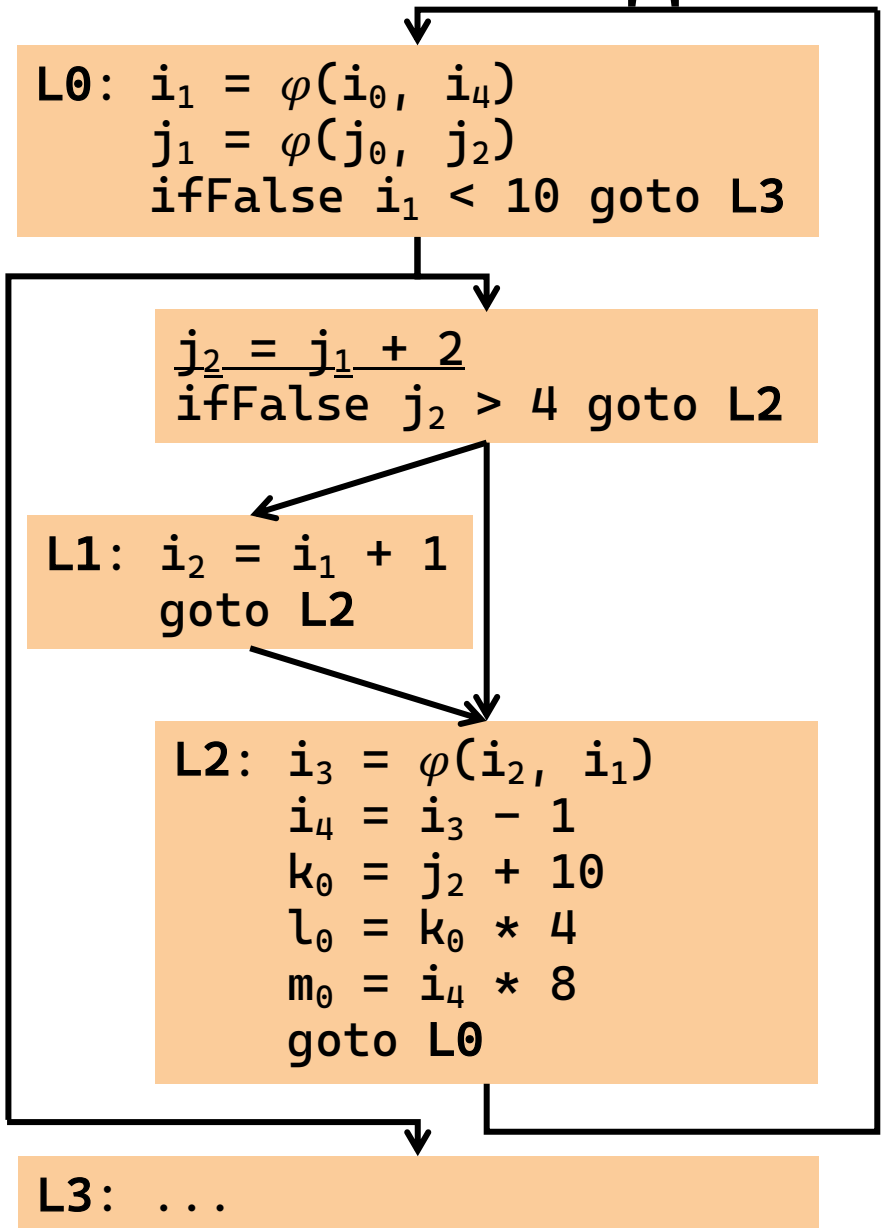


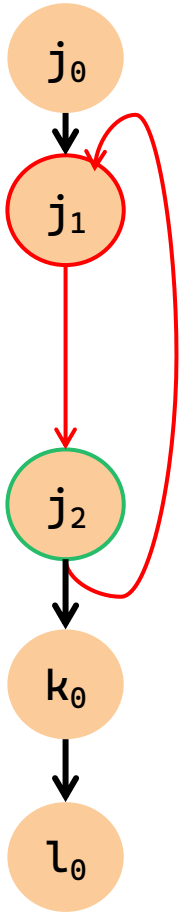
Worklist = {}

n == j_2:

M:{
j_1 -> ⟨j_1, 1, 0⟩
j_2 -> T
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



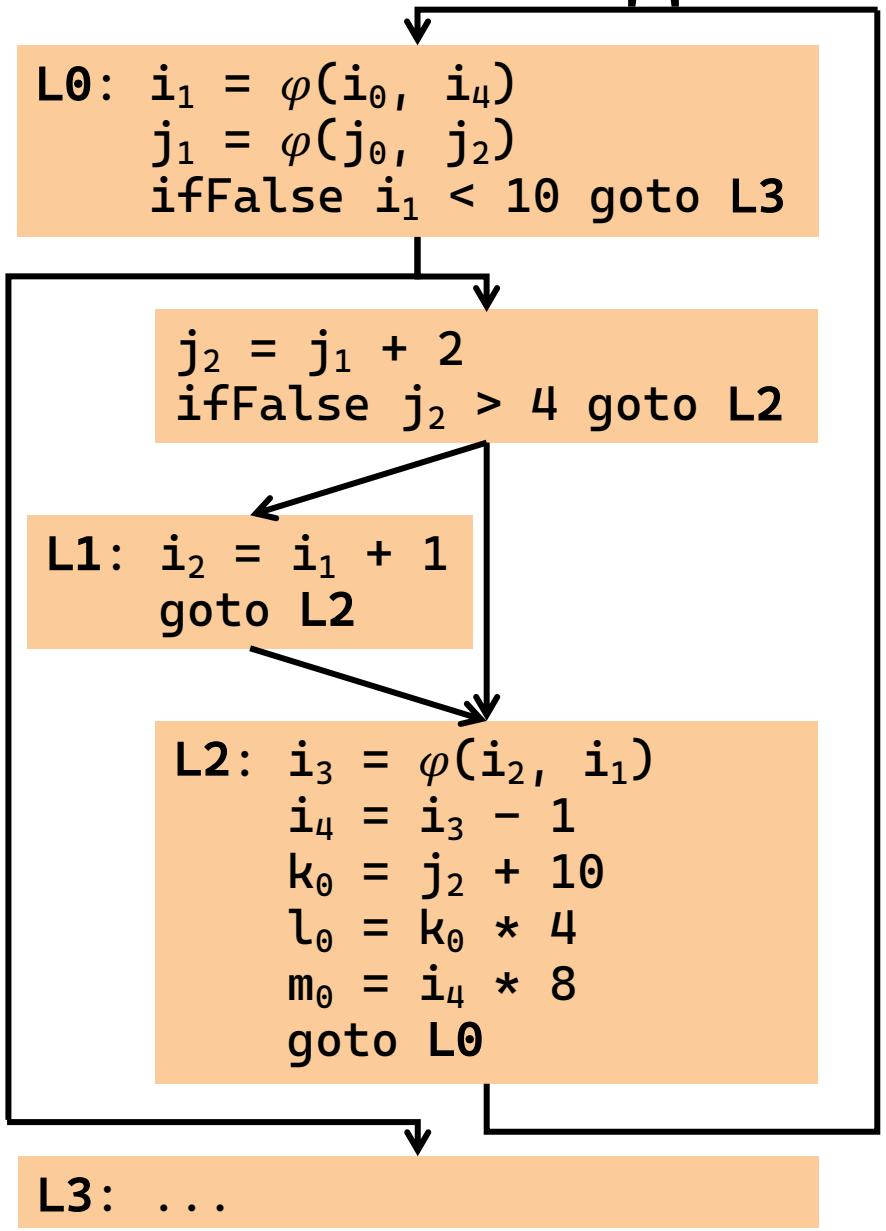


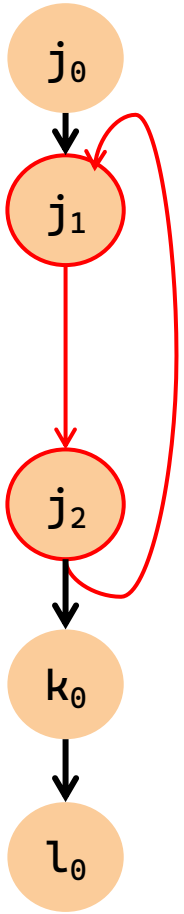
Worklist = {}

n == j_2:
newVal = f(n) // $j_2 = j_1 + 2$
// j_1 основная индуктивная
// переменная
m'(n) = ⟨j_1, 1, 2⟩
m'(n) != m(n) => Worklist += succ(j_2)

M:{
j_1 -> ⟨j_1, 1, 0⟩
j_2 -> T
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

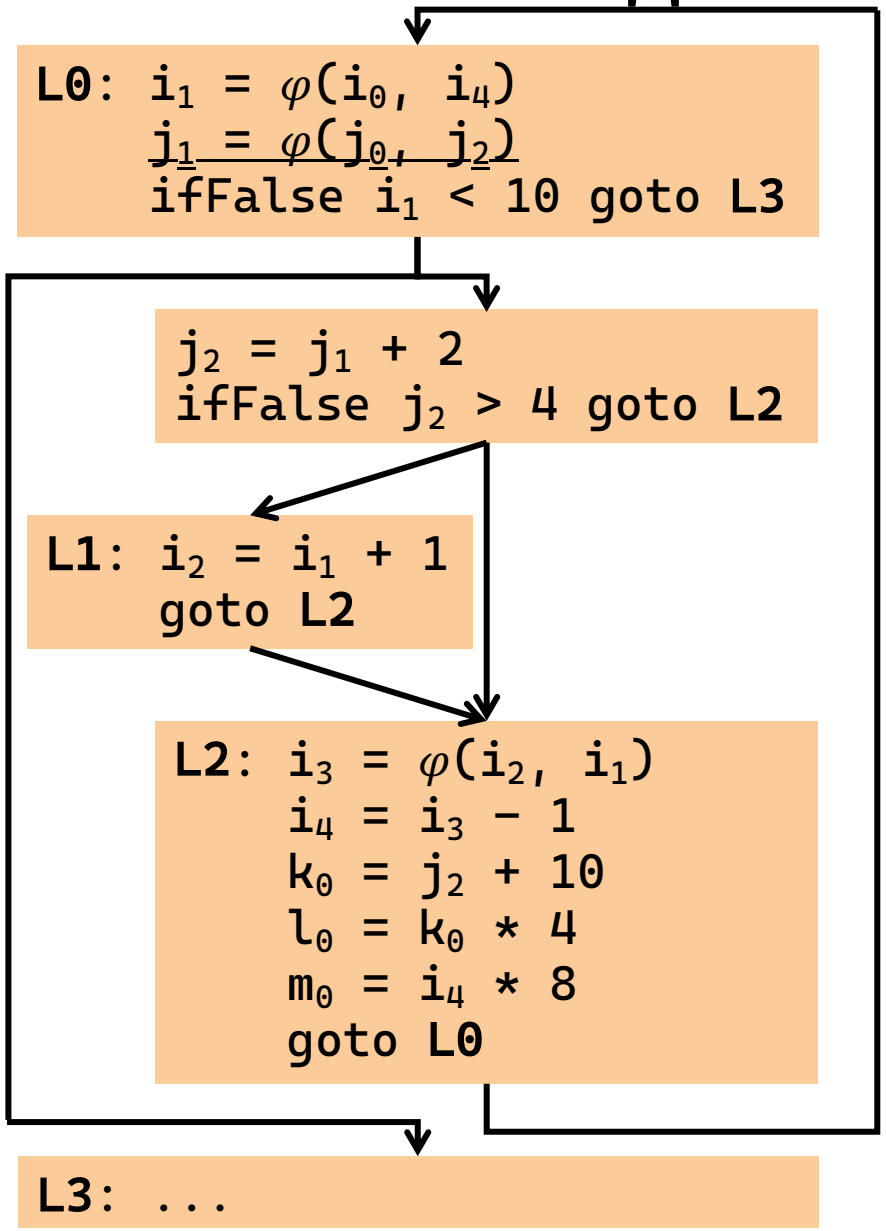


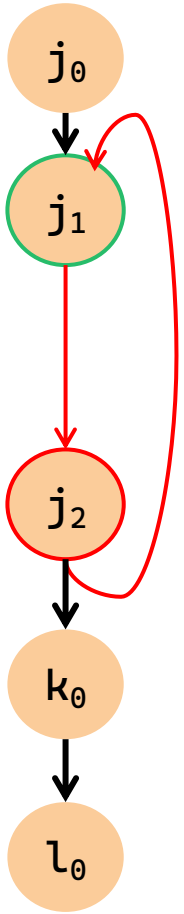


Worklist = {j_1}

M:{
j_1 -> ⟨j_1, 1, 0⟩
j_2 -> ⟨j_1, 1, 2⟩
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

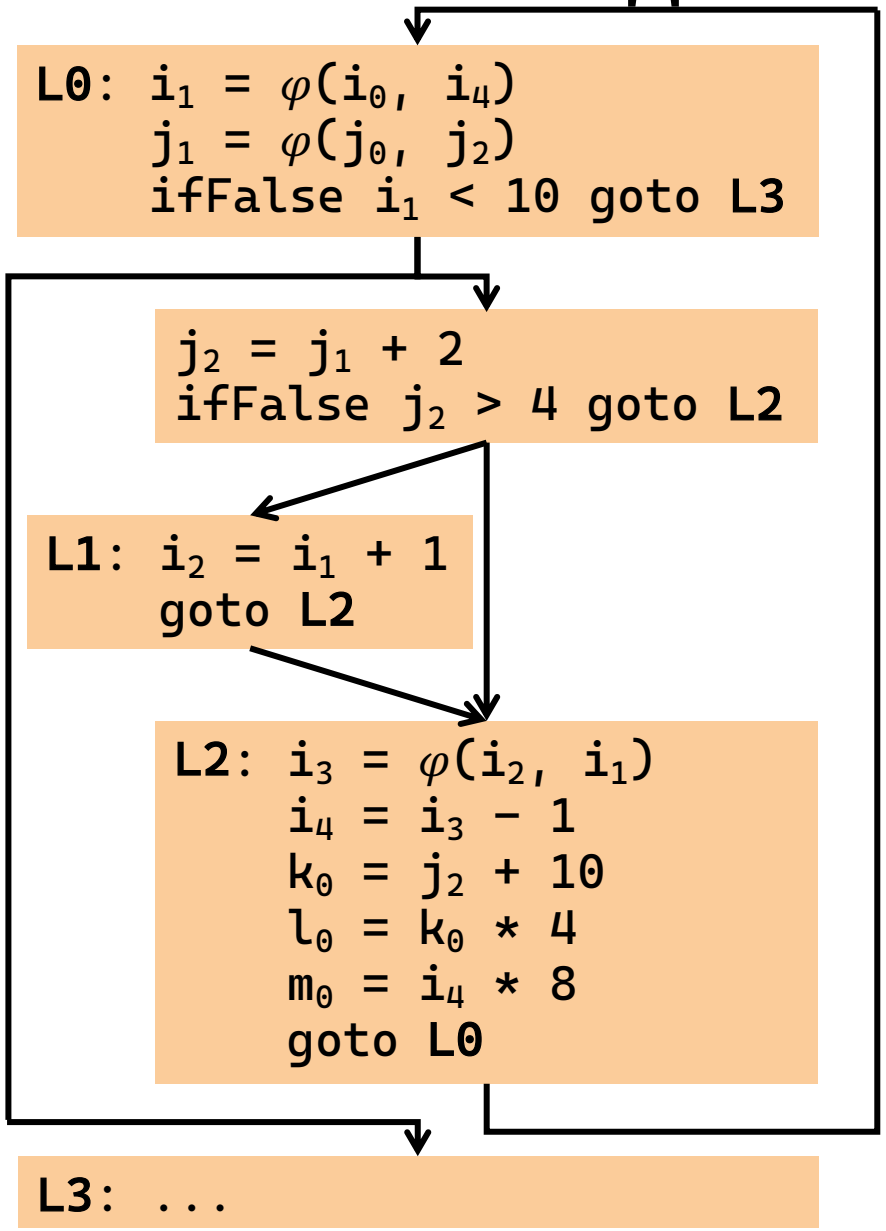


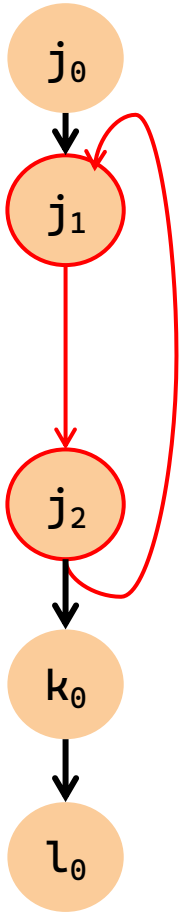



```
Worklist = {}

n == j1:
in = m(j0) ∧ m(j2) = ⟨j1, 1, 2⟩ => in != ⊥
newVal = f(n) // правила для φ
              // j1 – основная индуктивная
              // переменная =>
m'(n) = ⟨j1, 1, 0⟩
m'(n) == m(n) => continue;
empty(Worklist) == True
```

```
M:{
j1 -> ⟨j1, 1, 0⟩
j2 -> ⟨j1, 1, 2⟩
}
```

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

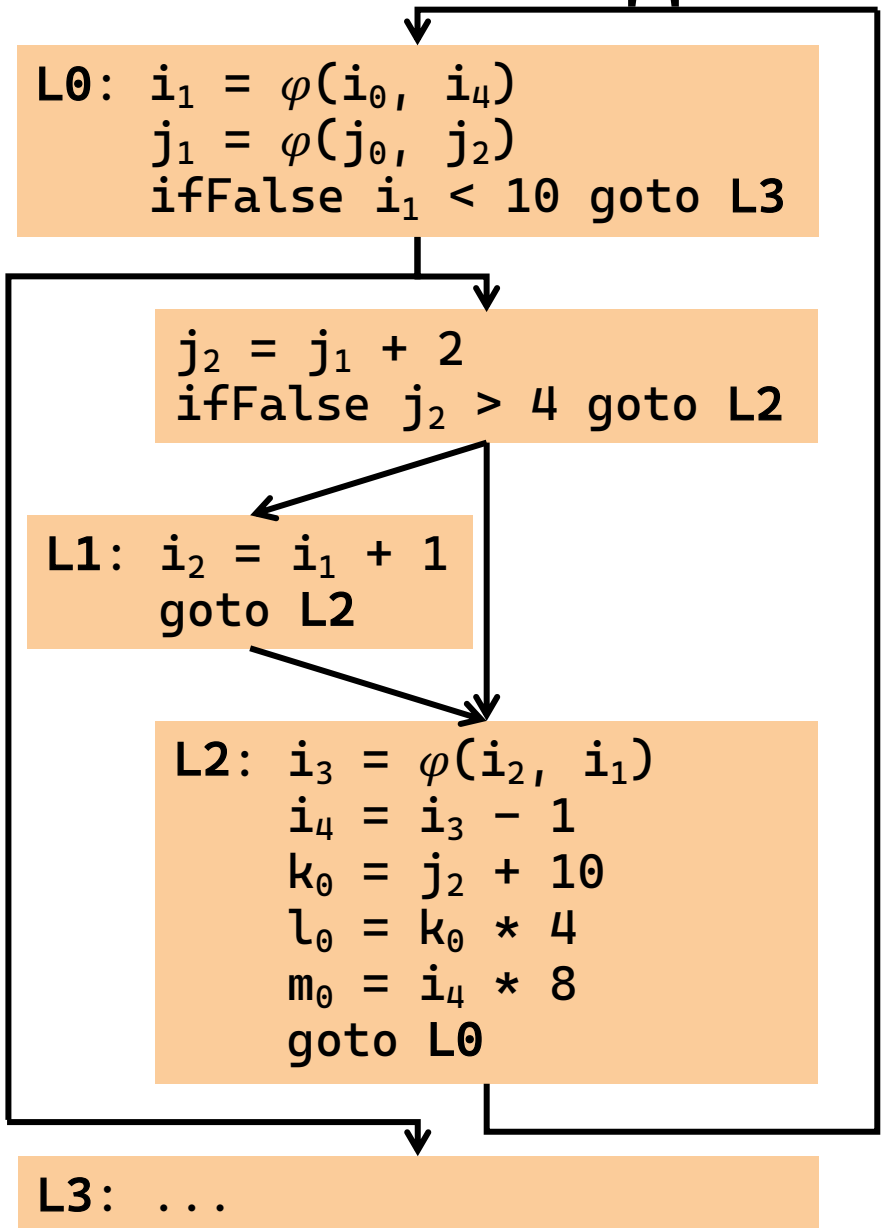


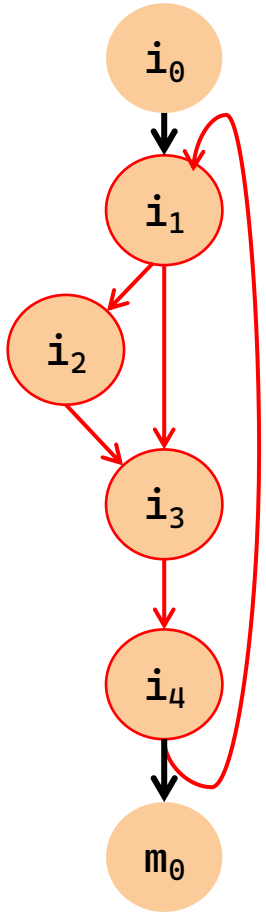


Worklist = {}

M:{
j_1 -> ⟨j_1, 1, 0⟩
j_2 -> ⟨j_1, 1, 2⟩
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

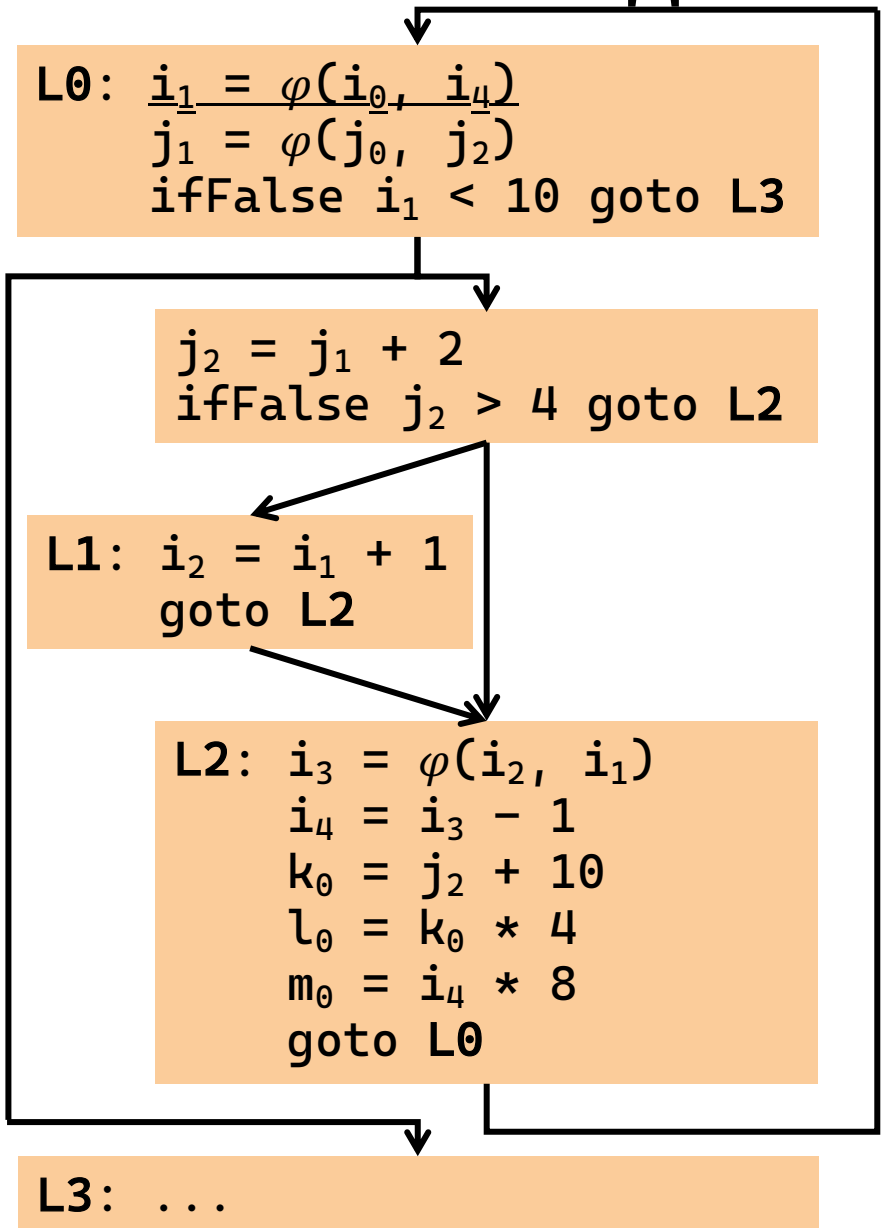


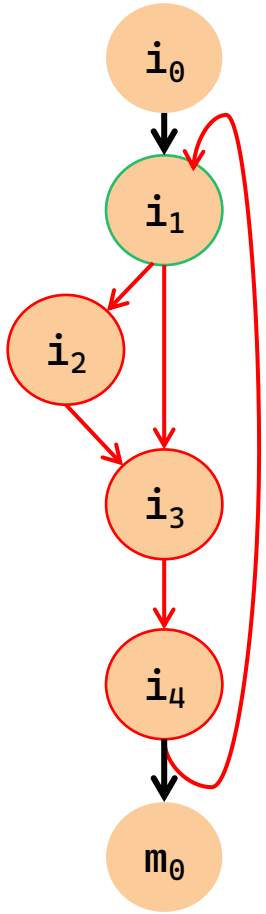


Worklist = {i_1, i_2, i_3, i_4}

M:{
i_1 -> T
i_2 -> T
i_3 -> T
i_4 -> T
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



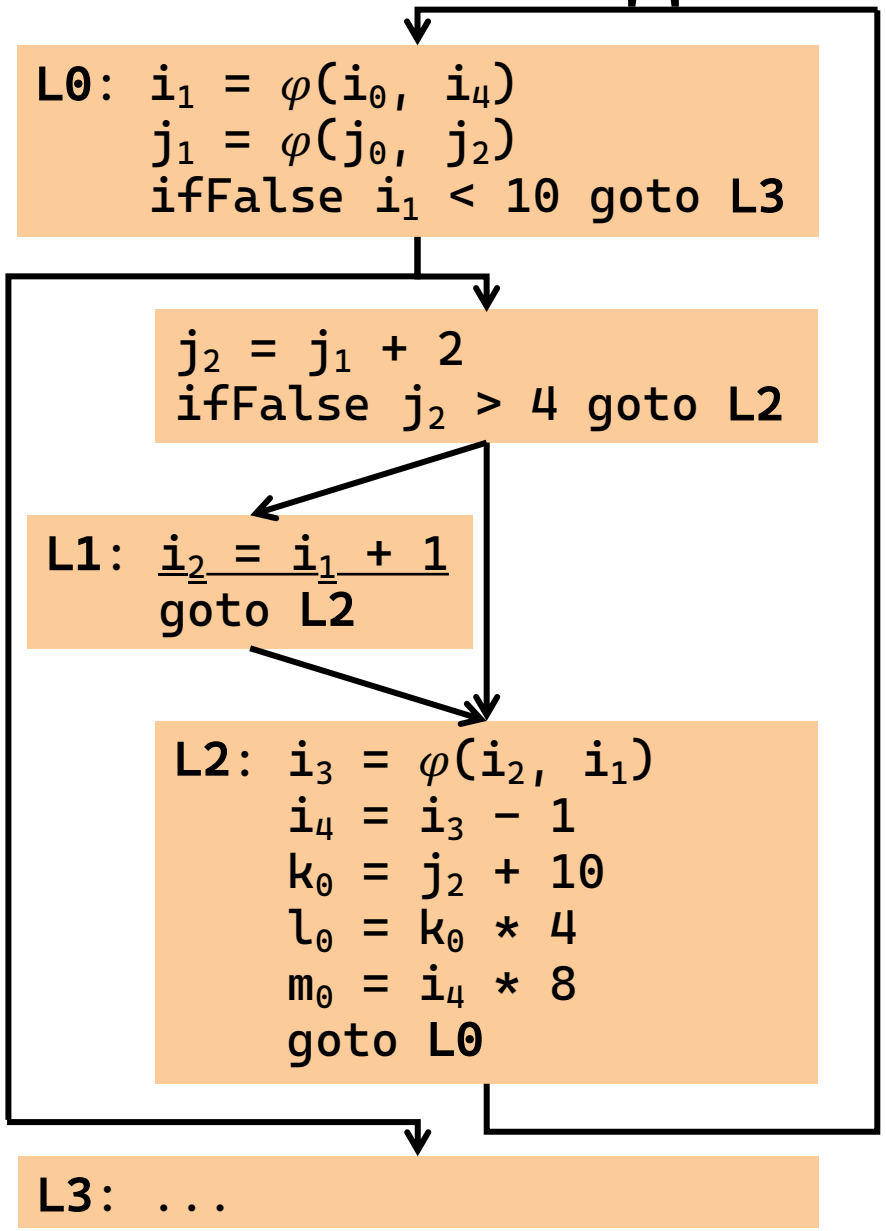


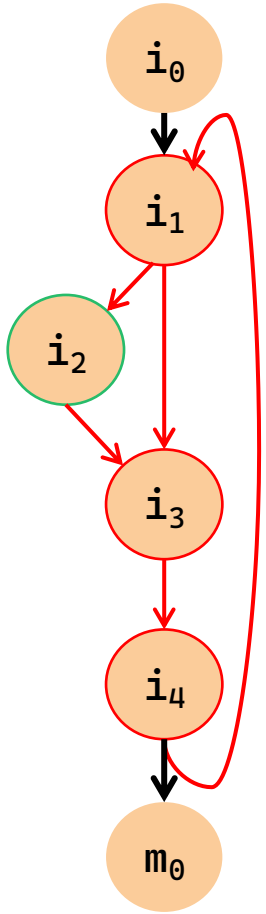
Worklist = {i_2, i_3, i_4}

n == i_1:
in = m(i_0) ∧ m(i_4) = ⊤ => in != ⊥
m'(i_1) = ⟨i_1, 1, 0⟩ // основная индуктивная
// переменная
m'(i_1) != m(i_1) => Worklist += succ(i_1)

M:{
i_1 -> ⊤
i_2 -> ⊤
i_3 -> ⊤
i_4 -> ⊤
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



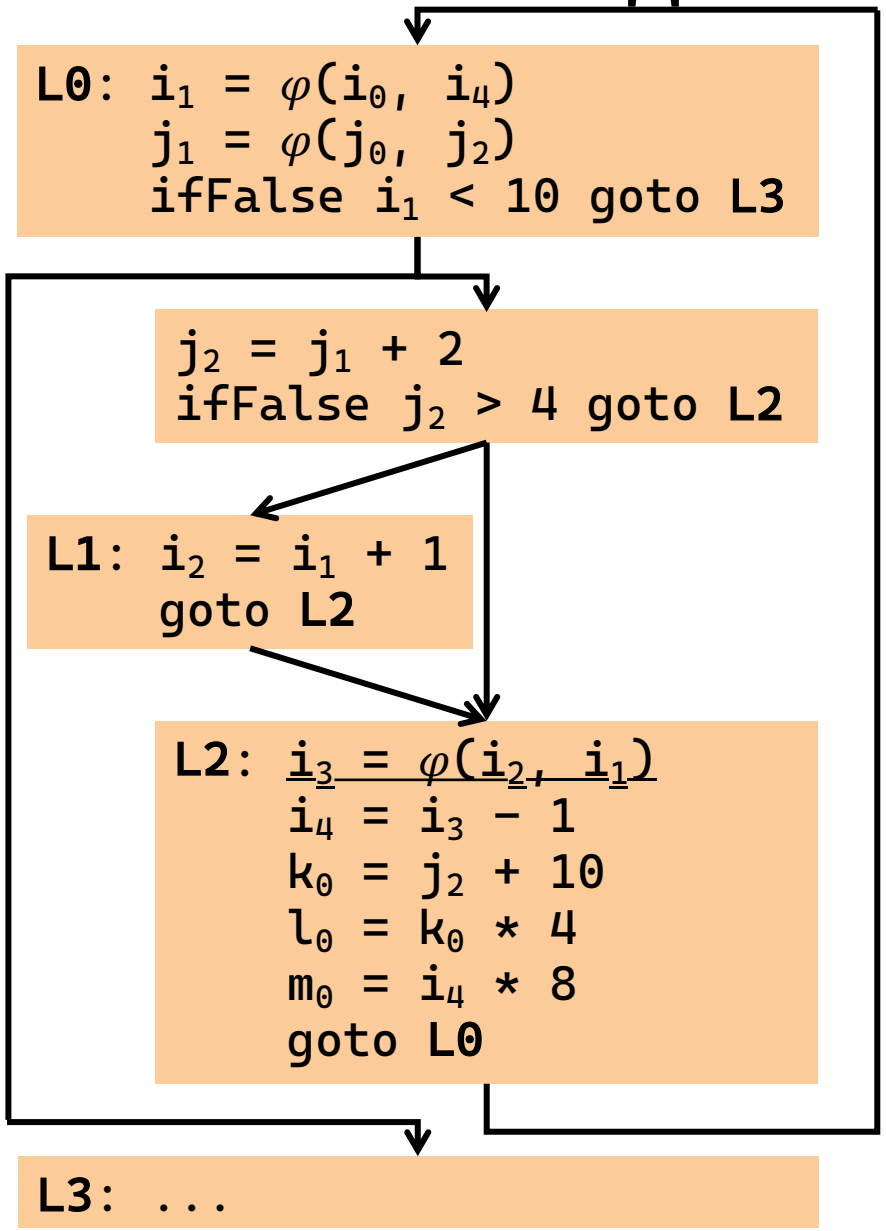


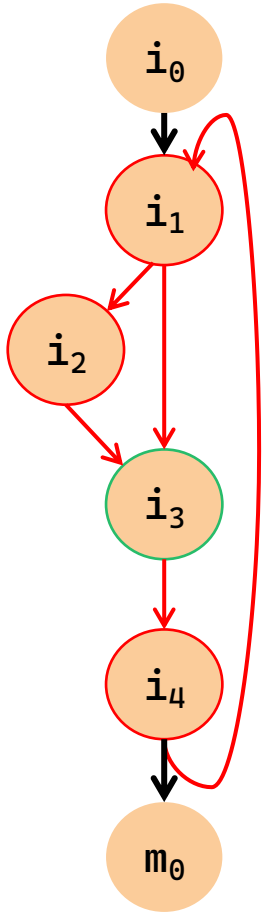
Worklist = {i_3, i_4}

n == j_2:
m'(i_2) = ⟨i_1, 1, 1⟩
m'(i_2) != m(i_2) => Worklist += succ(i_2)

M:{
i_1 -> ⟨i_1, 1, 0⟩
i_2 -> T
i_3 -> T
i_4 -> T
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



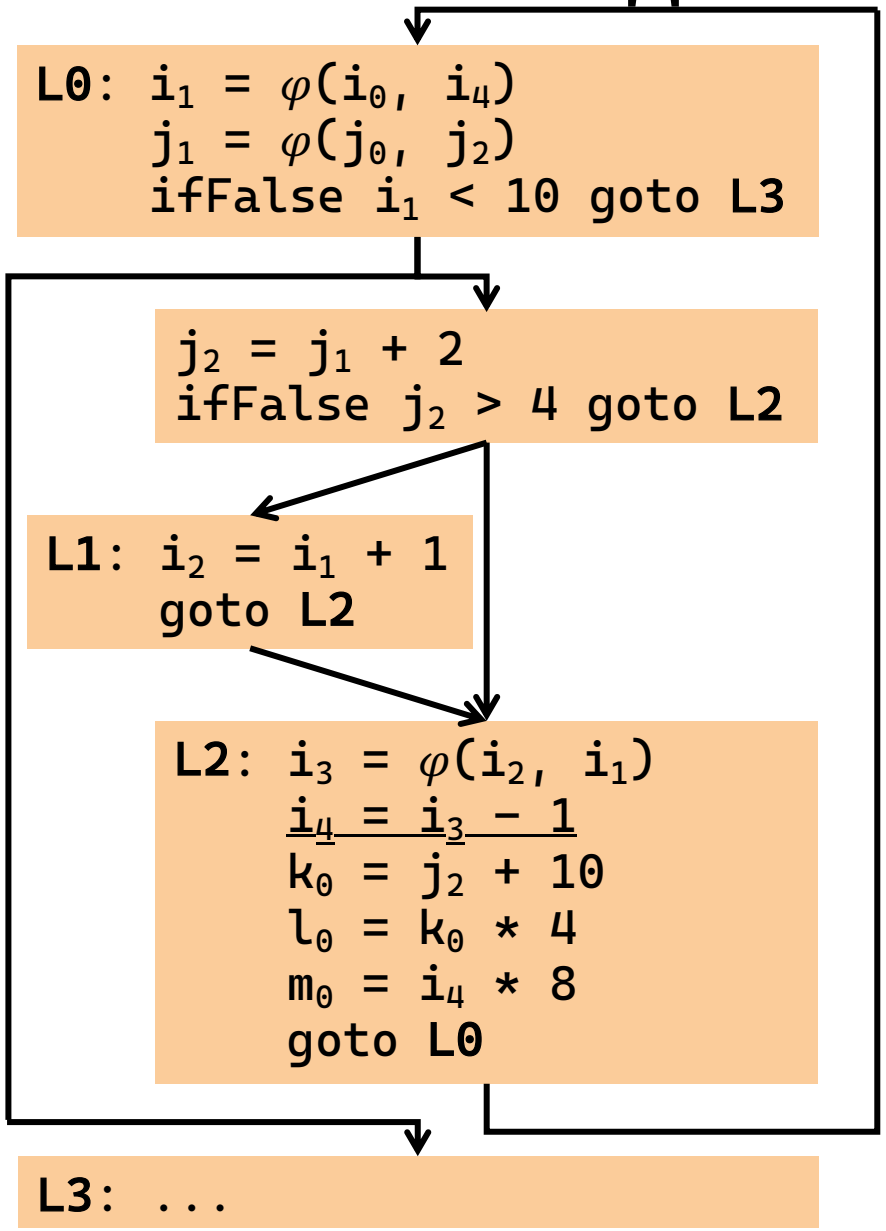


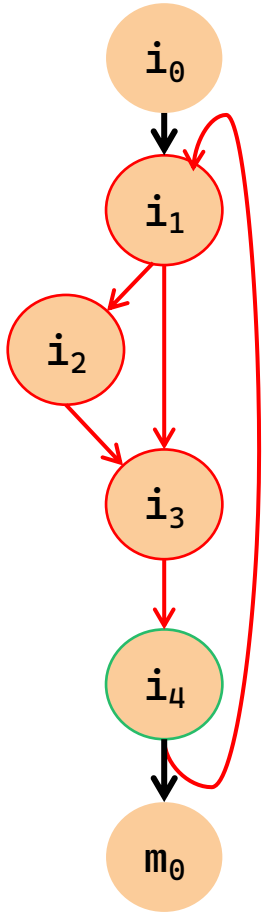
Worklist = {i_4}

n == i_3:
in = m(i_2) ∧ m(i_1)
= ⟨i_1, 1, 1⟩ ∧ ⟨i_1, 1, 0⟩ = ⊥ => in == ⊥
m'(i_3) = ⊥
m'(i_3) != m(i_3) => Worklist += succ(i_3)

M:{
i_1 -> ⟨i_1, 1, 0⟩
i_2 -> ⟨i_1, 1, 1⟩
i_3 -> T
i_4 -> T
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



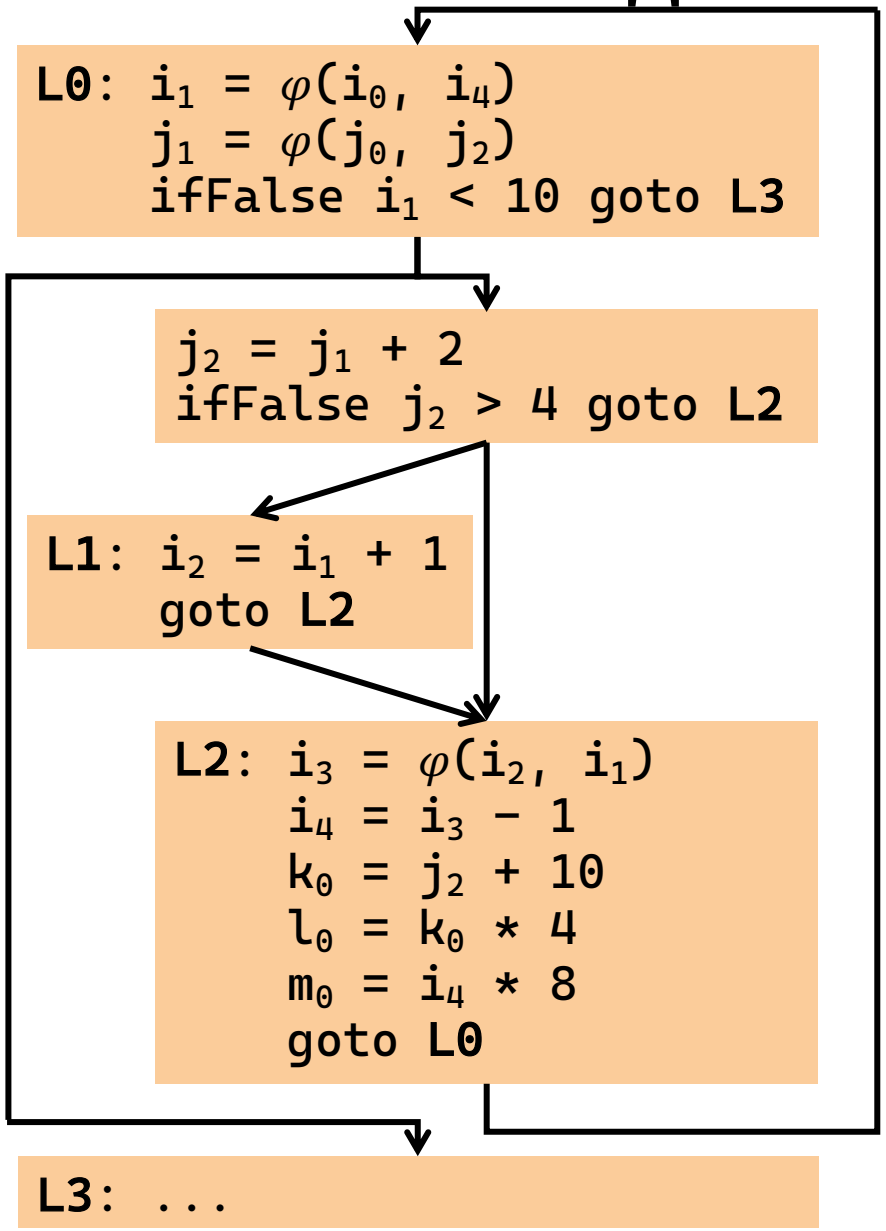


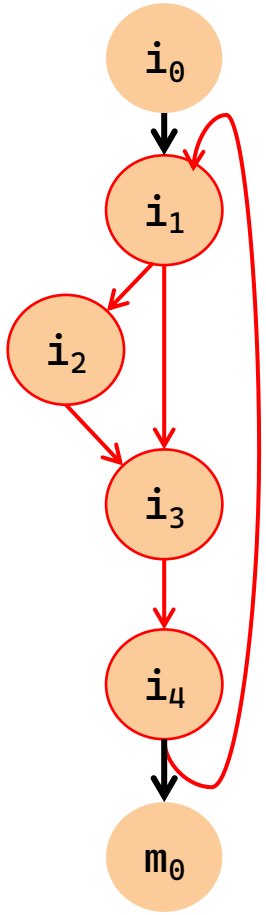
Worklist = {}

n == i_4:
$i_4 = i_3 - 1 \Rightarrow i_4 = \bot - 1 = \bot$
$m'(i_4) = \bot$
$m'(i_4) \neq m(i_4) \Rightarrow$ Worklist += succ(i_4)

M:{
$i_1 \rightarrow \langle i_1, 1, 0 \rangle$
$i_2 \rightarrow \langle i_1, 1, 1 \rangle$
$i_3 \rightarrow \bot$
$i_4 \rightarrow \top$
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

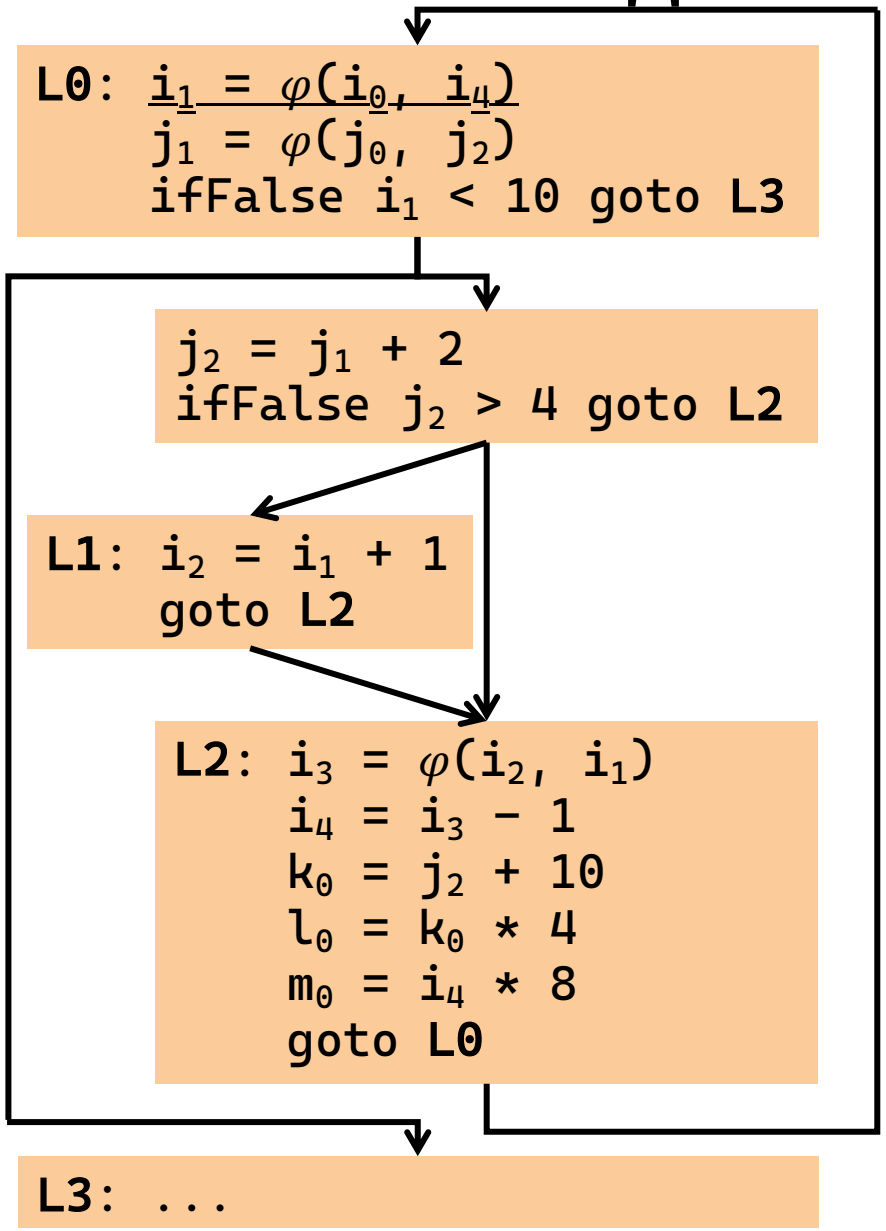


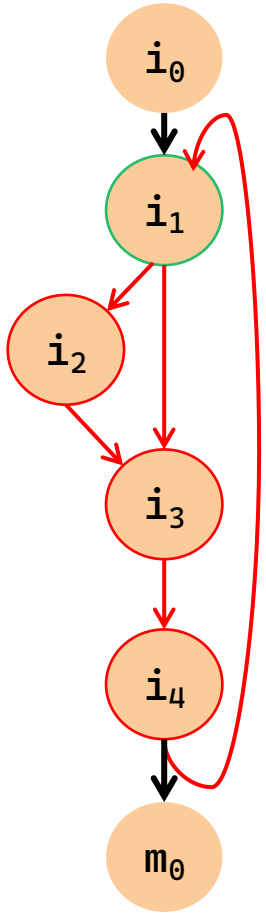


Worklist = {i_1}

M:{
i_1 -> ⟨i_1, 1, 0⟩
i_2 -> ⟨i_1, 1, 1⟩
i_3 -> ⊥
i_4 -> ⊥
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример





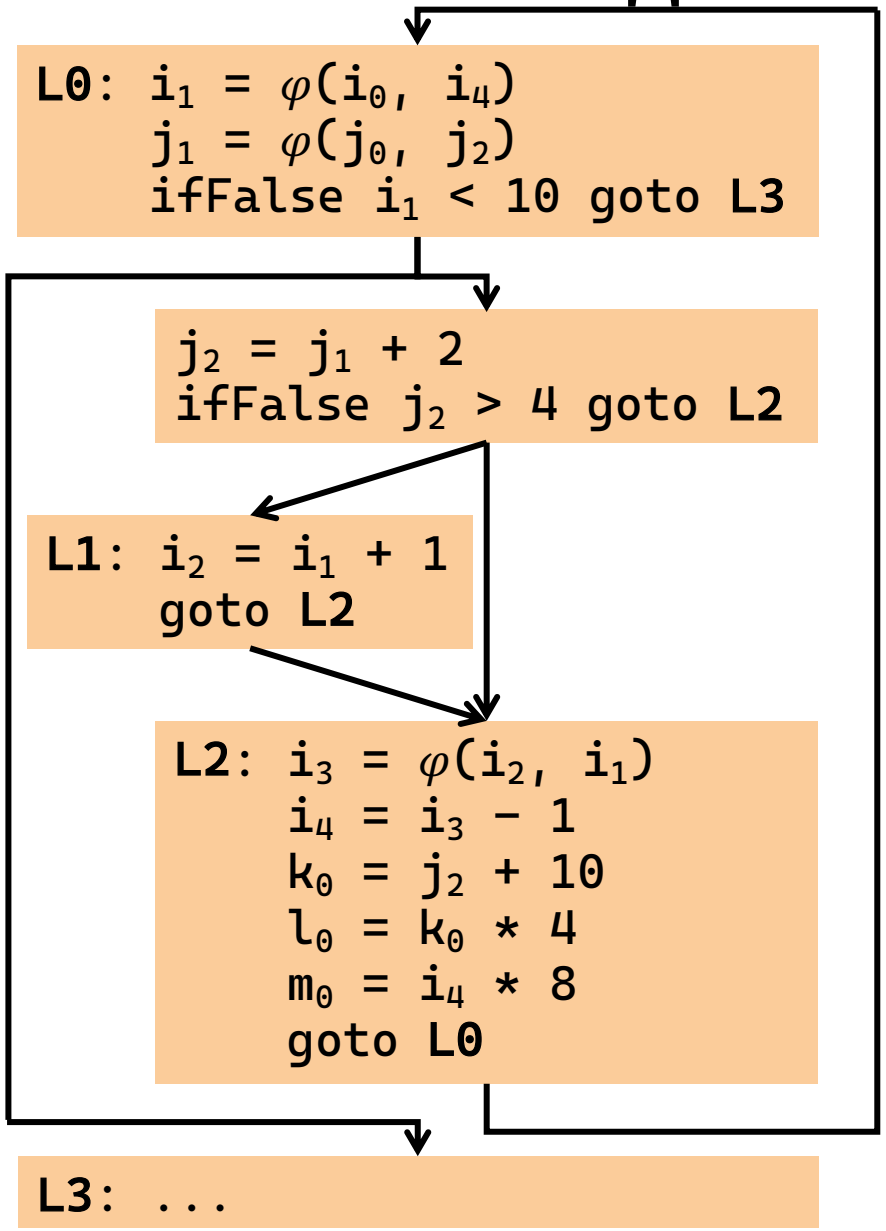
Worklist = {}

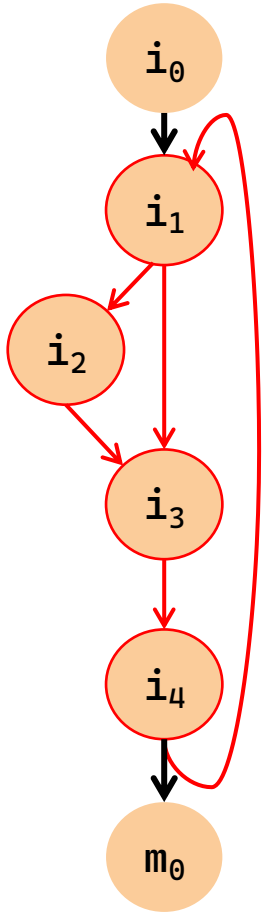
n == i_1:
in = m(i_0) ∧ m(i_4) = ⟨i_1, 1, 0⟩ ∧ ⊥
= ⊥ => in == ⊥

m'(i_1) = ⊥ // основная индуктивная переменная
// => убиваем всё семейство
// индуктивных переменных i_1
m'(i_1) = m'(i_2) = ⊥
m'(i_1) != m(i_1) => Worklist += succ(i_1)

M:{
i_1 -> ⟨i_1, 1, 0⟩
i_2 -> ⟨i_1, 1, 1⟩
i_3 -> ⊥
i_4 -> ⊥
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

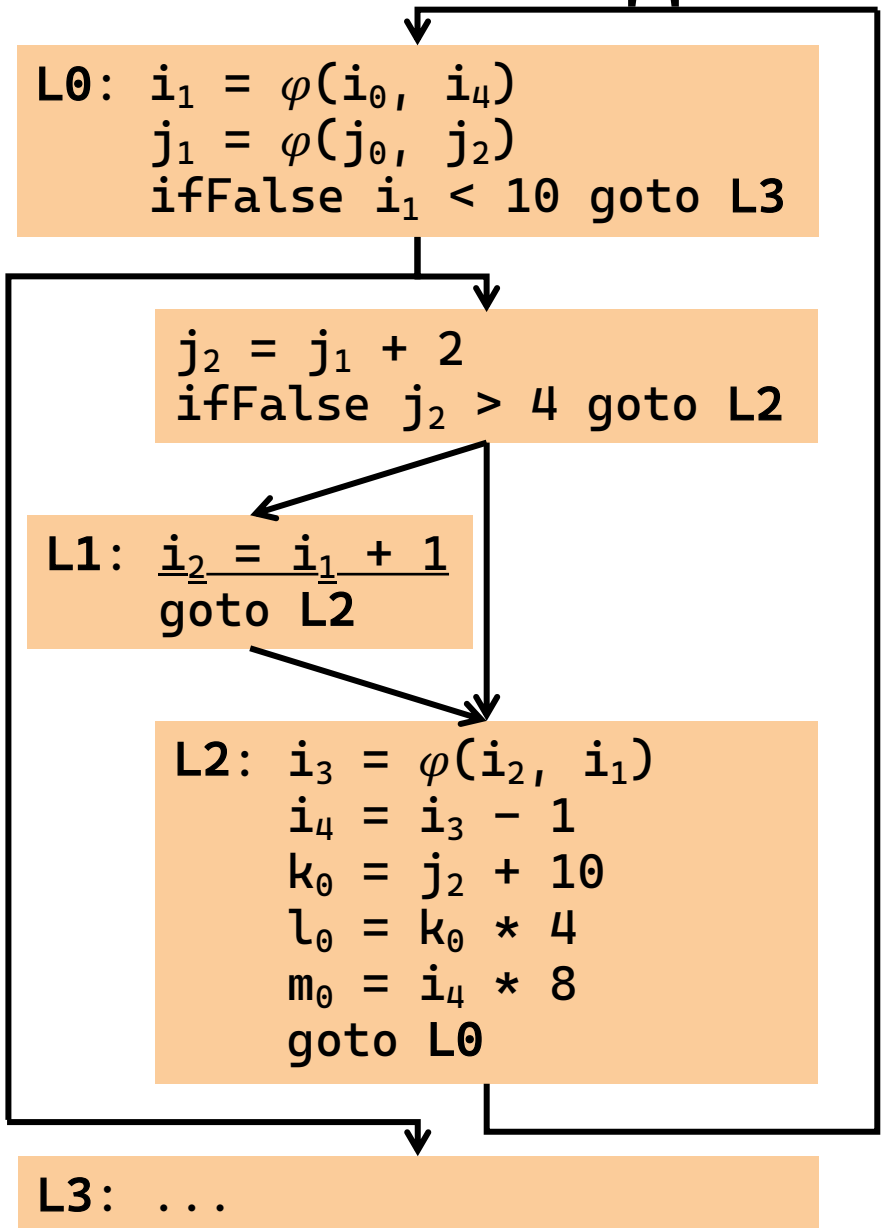


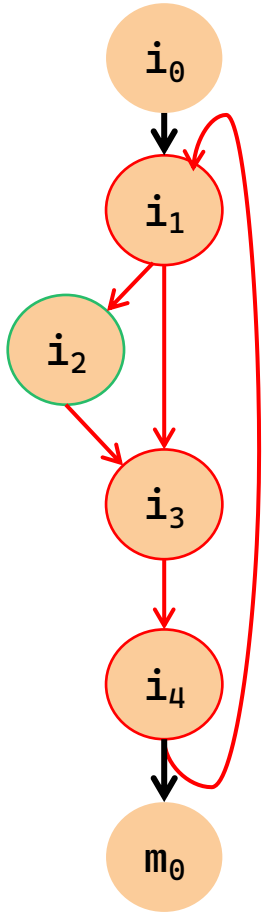


Worklist = {i_2, i_3}

M:{
i_1 -> ⊥
i_2 -> ⊥
i_3 -> ⊥
i_4 -> ⊥
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример



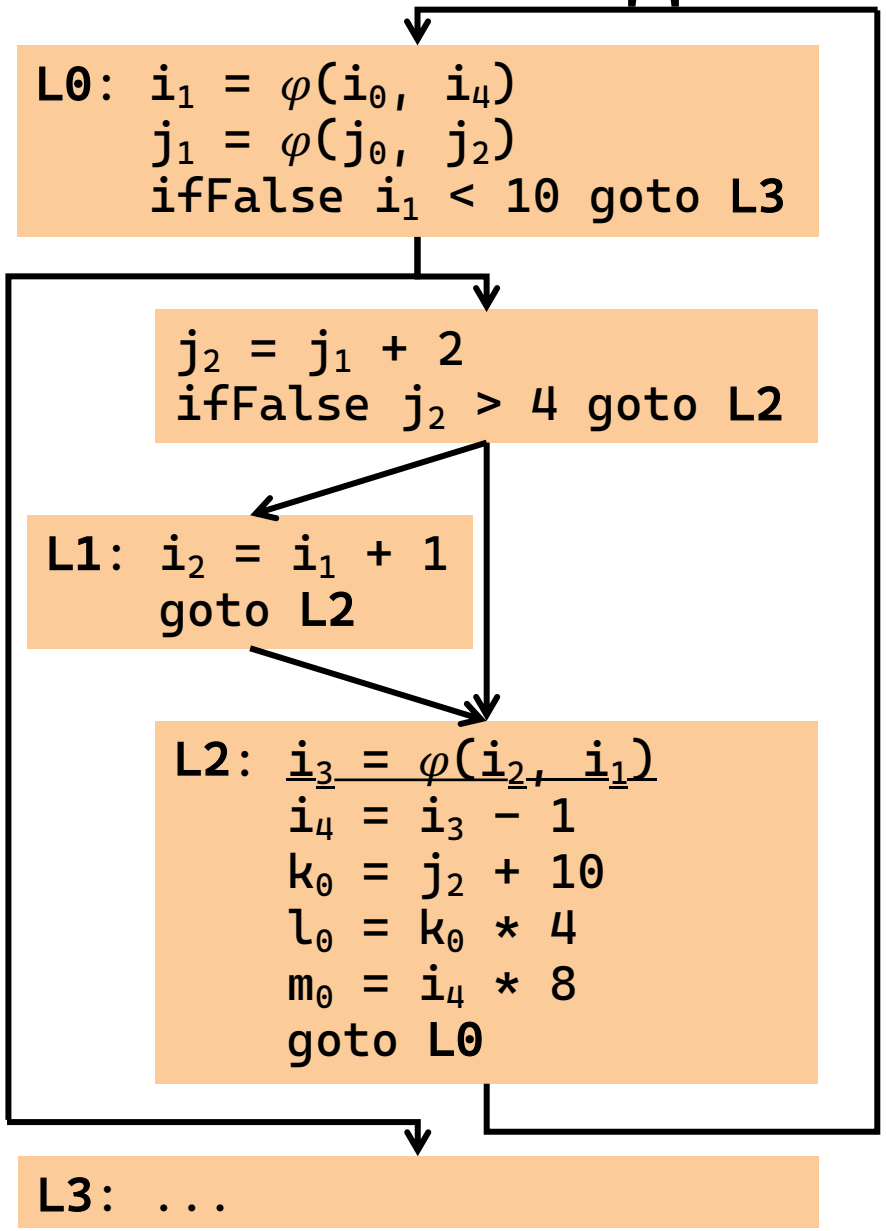


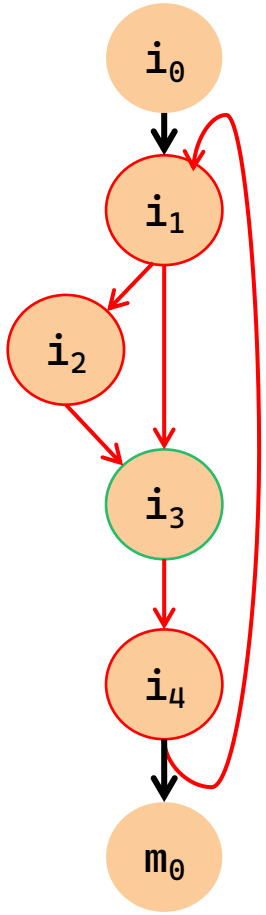
Worklist = {i_3}

n == i_2:
m'(i_2) = ⊥
m'(i_2) == m(i_2) => **continue;**

M:{
i_1 -> ⊥
i_2 -> ⊥
i_3 -> ⊥
i_4 -> ⊥
}

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

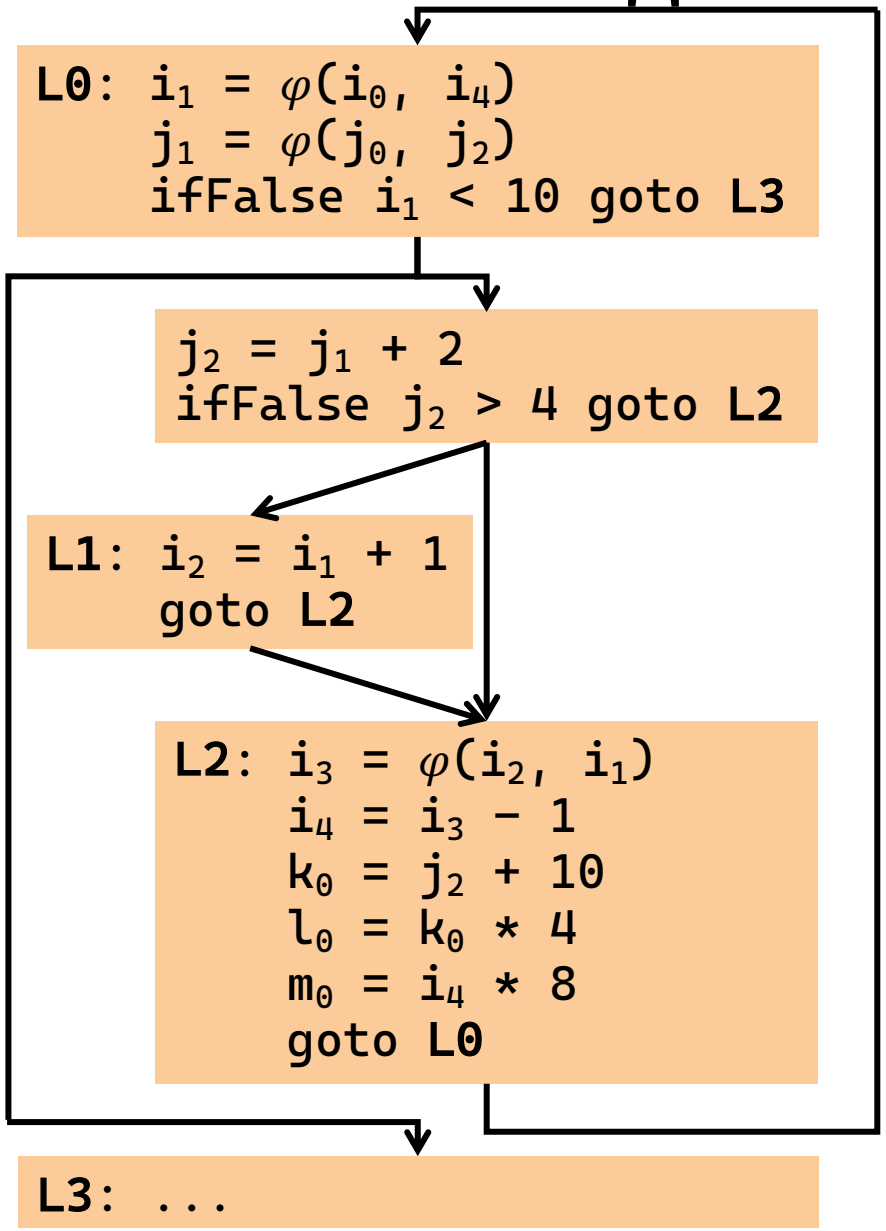


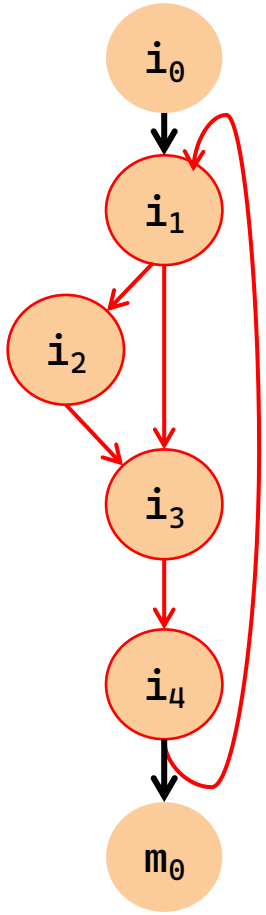



```
Worklist = {}

n == i3:
m'(i3) = ⊥
m'(i3) == m(i3) => continue;
```

```
M:{
i1 -> ⊥
i2 -> ⊥
i3 -> ⊥
i4 -> ⊥
}
```

Индуктивные переменные.
Обнаружение индуктивных переменных в SSA-форме. Пример

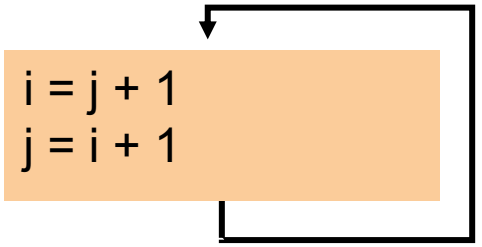




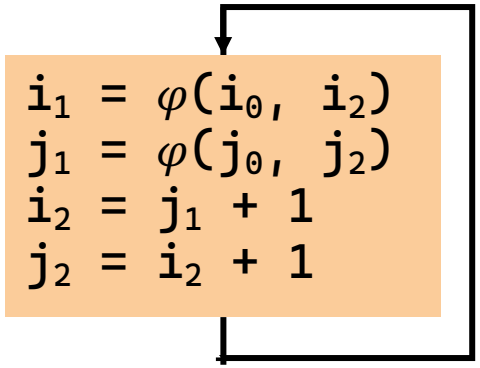
Worklist = {}

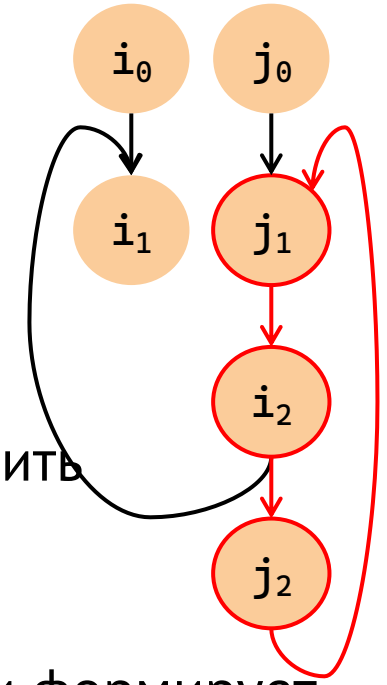
M:{
i_1 -> ⊥
i_2 -> ⊥
i_3 -> ⊥
i_4 -> ⊥
}

Индуктивные переменные. Обнаружение индуктивных переменных в SSA-форме. Пример 1



- Отсутствуют *основные индуктивные переменные*
- Чтобы обнаружить индуктивные переменные в вышеприведенном коде не в SSA-форме потребуется трансформировать программу (Выполнить подстановку **j** вместо **i** в выражении **j = i + 1**)





- SSA-форма позволяет определить все индуктивные переменные в вышеприведенном примере
- Компонента сильной связности формирует семейство индуктивных переменных
- j_1 основная индуктивная переменная
- i_2, j_2 производные индуктивные переменные
- i_1 производная индуктивная переменная семейства j_1

Индуктивные переменные. 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

Индуктивные переменные. 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 **⟨i,3,p⟩**, принадлежащая семейству основной индуктивной переменной **i**
- Как вычислять **j**, используя сложение вместо умножения?

Индуктивные переменные. Снижение сложности операций

- Как вычислять **j**, используя сложение вместо умножения?
- Сделать это очень просто:
- для производной индуктивной переменной **j** из семейства *основной индуктивной переменной* **i: ⟨i,1,0⟩** с инкрементом **h**, **m(j) = ⟨i,c,d⟩** 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**

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

Индуктивные переменные. Снижение сложности операций

- Применив алгоритм к рассматриваемому циклу, получим

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

=>

```
i = 1;
s = p + 3*i; // Предзаголовок
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

Индуктивные переменные. Снижение сложности операций

- Применив алгоритм к рассматриваемому циклу, получим

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

=>

```
i = 1;
s = p + 3*i; // Предзаголовок
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
- Алгоритм снижения стоимости позволил заменить в теле цикла умножение **`j = p + 3 * i`** на сложение **`s = s + 6`**

Индуктивные переменные.
Исключение избыточных индуктивных переменных

- **Во-первых**, переменные **j** и **s** – это, по существу, одна переменная, так как **j** это копия **s**
- **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; // Предзаголовок
while (i < a.length) {
  j = s;
  a[j] = a[j] + 1;
  i = i + 2;
  s = s + 6;
}
```

=>

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

Индуктивные переменные.
Исключение избыточных индуктивных переменных

- **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 **⟨`i, 3, p`⟩** семейства **`i`**,
  получим **`t = p + 3 * a.length`**
- После исключения **`i`** цикл примет окончательный вид:

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

=>

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

- Предзаголовок можно не оптимизировать: он выполняется всего один раз

Другие оптимизации циклов. Раскрутка циклов

- **Простой пример**. Выполним раскрутку 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 раскрутки 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];
  }
}
```

Другие оптимизации циклов. Раскрутка циклов

- **Простой пример**. Выполним раскрутку 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 раскрутки внутреннего цикла на 4:

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

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

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

```
}
```

Duff's device

```
do {
      *to = *from++;
} while (--count > 0);
```

```
n = count / 8;
    do {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
    } while (--n > 0);
```

```
    n = (count + 7) / 8;

    switch (count % 8) {
        case 0: *to = *from++;
        case 7: *to = *from++;
        case 6: *to = *from++;
        case 5: *to = *from++;
        case 4: *to = *from++;
        case 3: *to = *from++;
        case 2: *to = *from++;
        case 1: *to = *from++;
    }
    while (--n > 0) {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
    }
```

```
n = (count + 7) / 8;

switch (count % 8) {
case 0: do {  *to = *from++;
case 7:       *to = *from++;
case 6:       *to = *from++;
case 5:       *to = *from++;
case 4:       *to = *from++;
case 3:       *to = *from++;
case 2:       *to = *from++;
case 1:       *to = *from++;
        } while (--n > 0);
}
```

Другие оптимизации циклов. Раскрутка циклов

- **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];
}
```

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

Другие оптимизации циклов. 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];
}
```

Другие оптимизации циклов. 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
- 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];
}
```

Другие оптимизации циклов. 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
- После второго 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];
  for(i = 1; i < ni; i++)
    y[i] += x[j + 3] * m[i][j + 3];
}
```

Другие оптимизации циклов. 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
- 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];
}
```

Другие оптимизации циклов. 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
- Последний цикл можно преобразовать*:

```
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];
}
```

*сокращены обращения к памяти

Другие оптимизации циклов. Слияние циклов

```
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];
}
```

```
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] = 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];
}
```

Другие оптимизации циклов. Слияние циклов

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

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

Рассмотренное оптимизирующее преобразование называется *раскруткой с последующим сжатием* (*Unroll and Jam*)

```
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 = 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];
}
```

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

Другие оптимизации циклов. Раскрутка циклов. 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 C-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).

Применение индуктивных переменных. Раскрутка цикла (Loop Unrolling)

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

```
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];
  }
}
```

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

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*

Применение индуктивных переменных. Размыкание цикла (Loop Unswitching)

```
for (i = 0; i < 1000; i++) {
  x[i] += y[i];
  if (w)
    y[i] = 0;
}
```

```
if (w) {
  for (i = 0; i < 1000; i++) {
    x[i] += y[i];
    y[i] = 0;
  }
} else {
  for (i = 0; i < 1000; i++)
    x[i] += y[i];
}
```

Размыкание цикла (loop unswitching) выносит условия за пределы цикла и с последующим дублированием тела цикла с помещением соответствующих вариантов в соответствующие ветви условия

Размыкание цикла позволяет улучшить производительность за счет последующей векторизации цикла

Размыкание цикла может быть выполнено совместно с размоткой цикла, а результатом размотки, в свою очередь, являются несколько операций в итерации, производимые над последовательными участками памяти, которые можно заменить одной векторной (если архитектура поддерживает векторные инструкции)

Размыкание + размотка цикла позволяет более эффективно выполнить цикл параллельно

Применение индуктивных переменных. Расщепление тела цикла и слияние циклов (Loop Fission and Loop Fusion)

```
for (i = 0; i < 100; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

Расщепление тела цикла разбивает цикл на несколько циклов, каждый из которых имеет те же индексные границы, однако содержит только часть тела исходного цикла

Расщепление позволяет устранить зависимости по данным в цикле и может использоваться для подготовке к векторизации цикла, для перестановок циклов или повышения вероятности попадания попадания в кеш процессора

Расщепление тела цикла оправдано если количество вычислений в цикле становится чрезмерным (вопросы производительности) или если цикл содержит условные операторы. В этом случае бывает полезно разделить циклы на два:
один с условным оператором и один без него

```
for (i = 0; i < 100; i++)
  a[i] = 1;
for (i = 0; i < 100; i++)
  b[i] = 2;
```

Слияние циклов (loop fusion) объединяет несколько циклов,
смежных в дереве циклов, в один

Преобразование возможно, если циклы имеют одинаковое количество итераций и не зависят друг от друга по данным

Слияние циклов может повысить локальность данных,
что повышает эффективность работы кэша

Слияние циклов не всегда сокращает время исполнения программы.
В некоторых случаях может оказаться более выгодным исполнить два цикла вместо одного объединённого, так как, например, локальность данных в таком случае может оказаться выше

В некоторых случаях может выполняться расщепление цикла
с последующим слиянием, чтобы перестроить цикл другим способом
для повышения производительности

Применение индуктивных переменных. Расщепление тела цикла и слияние циклов (Loop Fission and Loop Fusion)

```
for (i = 0; i < 100; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

Расщепление тела цикла разбивает цикл на несколько циклов, каждый из которых имеет те же индексные границы, однако содержит только часть тела исходного цикла

Слияние циклов (loop fusion) объединяет несколько циклов, смежных в дереве циклов, в один

Слияние корректно, когда каждое определение и каждое использование в результирующем (слитом) цикле имеют такие же значения, что и в исходных (сливаемых) циклах

```
for (i = 0; i < 100; i++)
  a[i] = 1;
for (i = 0; i < 100; i++)
  b[i] = 2;
```

Применение индуктивных переменных. Расщепление цикла (Loop splitting). Разгрузка цикла (Loop peeling)

До:

```
int p = 10;
for (int i = 0; i < 10; ++i) {
    y[i] = x[i] + x[p];
    p = i;
}
```

После:

```
y[0] = x[0] + x[10];
for (int i = 1; i < 10; ++i)
  y[i] = x[i] + x[i-1];
```

Разгрузка цикла (или *отслаивание*) – частный случай расщепления цикла, в этом случае из основного цикла выносятся одна или несколько первых (или последних) итераций. Таким образом, цикл разбивается на несколько циклов, с различными диапазонами счётчика. Это позволяет упростить тело цикла в каждом из получившихся циклов в соответствии с исходными условиями на счетчик цикла.

Применение индуктивных переменных. Расщепление цикла (Loop splitting). Разгрузка цикла (Loop peeling)

До:

```
for (int i = 0; i < N; ++i) {
    if (i < 2) {
      // Block A
    }
   // Block B
}
```

После:

```
for (int i = 0; i < min(2, N); ++i) {
  // Block A
  // Block B
}

for (int i = 2; i < N; ++i) {
  // Block B
}
```

Разгрузка цикла (или *отслаивание*) – частный случай расщепления цикла, в этом случае из основного цикла выносятся одна или несколько первых (или последних) итераций. Таким образом, цикл разбивается на несколько циклов, с различными диапазонами счётчика. Это позволяет упростить тело цикла в каждом из получившихся циклов в соответствии с исходными условиями на счетчик цикла.

Заключительные замечания. 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
- 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