6. Форма статического единственного присваивания (*SSA*-форма)

6.1 Форма статического единственного присваивания (SSA-форма)

6.1.1. Постановка задачи

◊ *Форма статического единственного присваивания* (*SSA*) позволяет в каждой точке программы объединить

- информацию об имени переменной
- с информацией о текущем значении этой переменной (или, что то же самое, с информацией о том, какое из определений данной переменной определяет ее текущее значение в данной точке).

6.1 Форма статического единственного присваивания (*SSA-форма*)

6.1.1. Постановка задачи

◊ *Форма статического единственного присваивания* (*SSA*) позволяет в каждой точке программы объединить

- информацию об имени переменной
- с информацией о текущем значении этой переменной (или, что то же самое, с информацией о том, какое из определений данной переменной определяет ее текущее значение в данной точке).

◊ Хотелось бы, чтобы программа в *SSA*-форме удовлетворяла двум условиям:

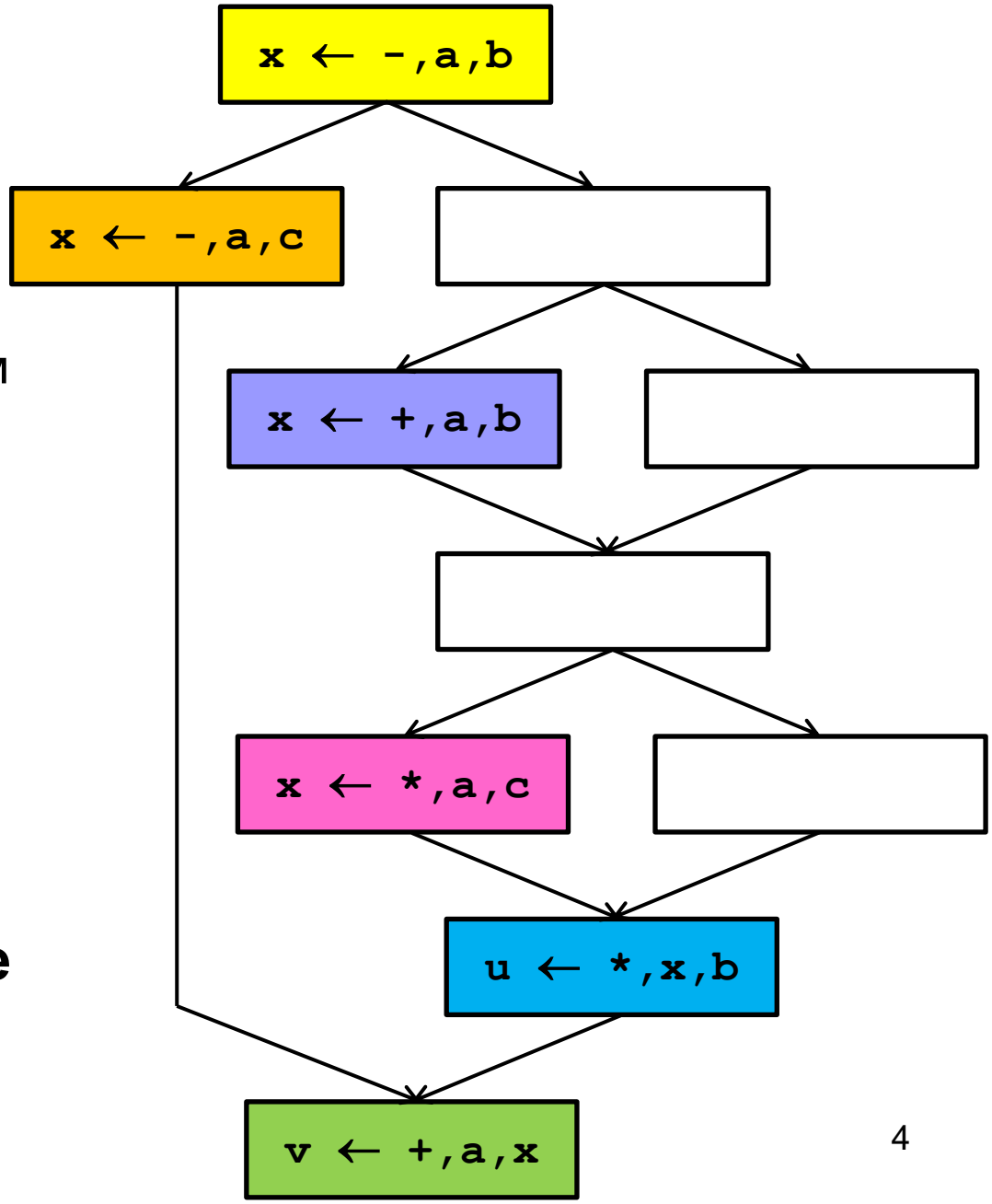
(1) каждое определение переменной имеет индивидуальное имя;

(2) каждое использование переменной ссылается на единственное определение.

6.1 SSA-форма

6.1.1. Постановка задачи

- Использования в синем блоке достигают три определения **x**, в зеленом – четыре определения **x**
- Цель же состоит в том, чтобы **каждого использования достигало только одно определение**
- Введем «функцию» объединения значений или *φ-функцию*



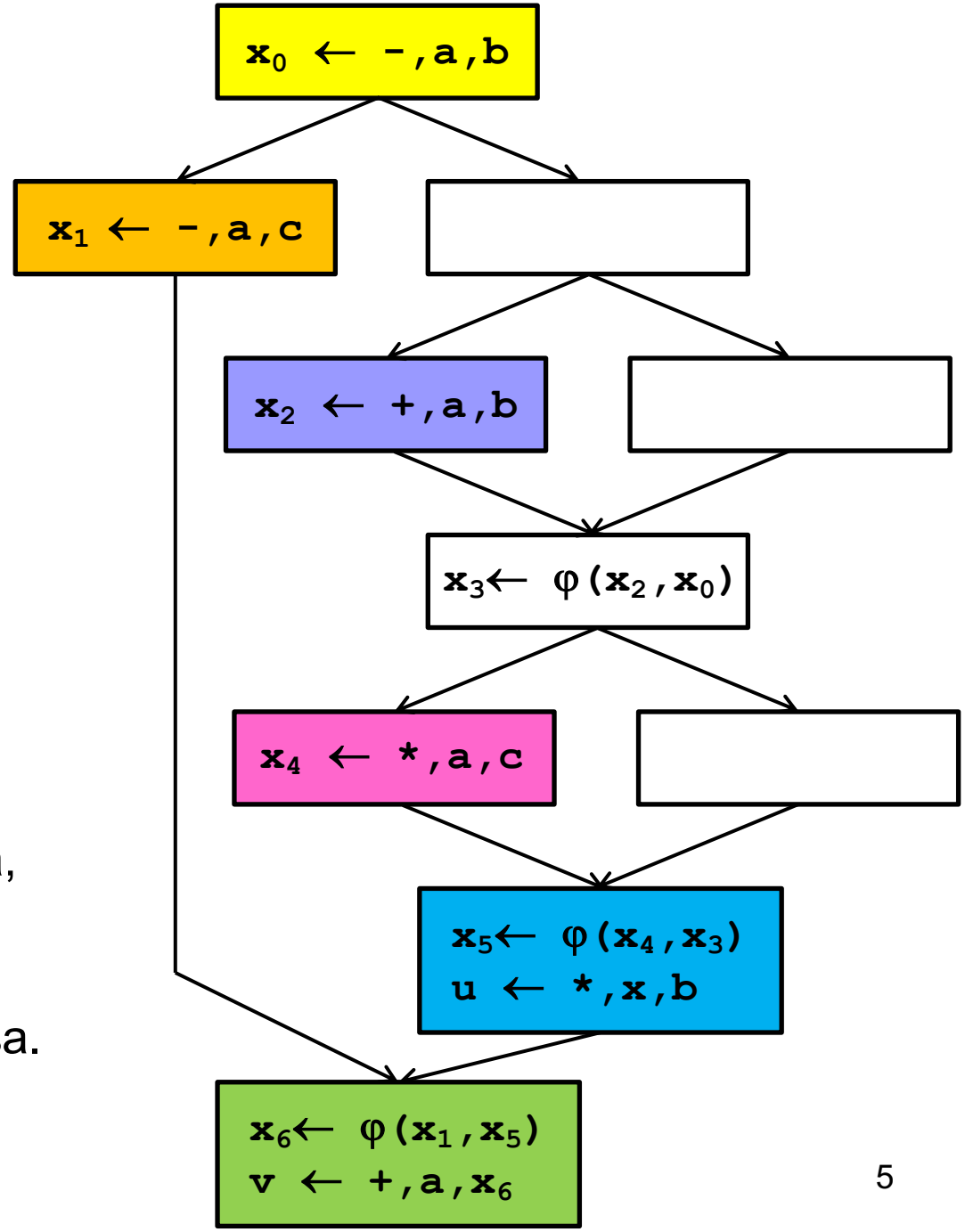
6.1 SSA-форма

6.1.1. Постановка задачи

◊ По определению $x_3 \leftarrow \varphi(x_2, x_0)$ является новым определением переменной x:

значение x_3 равно x_2, когда управление попадает в блок слева,

значение x_3 равно x_0, когда управление попадает в блок справа.



6.1 SSA-форма

6.1.2. Определение φ-функции

- ◊ φ-функция определяет *SSA-имя* для значения своего аргумента, соответствующего ребру, по которому управление входит в блок.
- ◊ **При входе в базовый блок все его φ-функции выполняются одновременно и до любого другого оператора, определяя целевые *SSA*-имена.**

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
    x = x + 1
    y = y + x
}
```

6.1 SSA-форма

6.1.2. Определение φ -функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

Каждая из φ-функций *объединяет* значения своих аргументов в новое значение, определением которого она является.

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

Каждая из φ-функций *объединяет* значения своих аргументов в новое значение, определением которого она является.

До цикла –	x_0, y_0
На входе в цикл –	x_1, y_1
Внутри цикла –	x_2, y_2
После цикла –	x_3, y_3

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

◊ **Затруднение**. Первый аргумент $\varphi(x_0, x_2)$ определяется в блоке, который предшествует циклу, второй аргумент определяется позже в блоке, содержащем $\varphi(x_0, x_2)$. Следовательно, при первом выполнении $\varphi(x_0, x_2)$ ее второй аргумент еще не определен.

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

◊ **Выход из затруднения**: по определению любая φ-функция (а значит и $\varphi(x_0, x_2)$) **читает только один из своих аргументов**, а именно тот аргумент, который соответствует последнему пройденному ребру ГПУ.

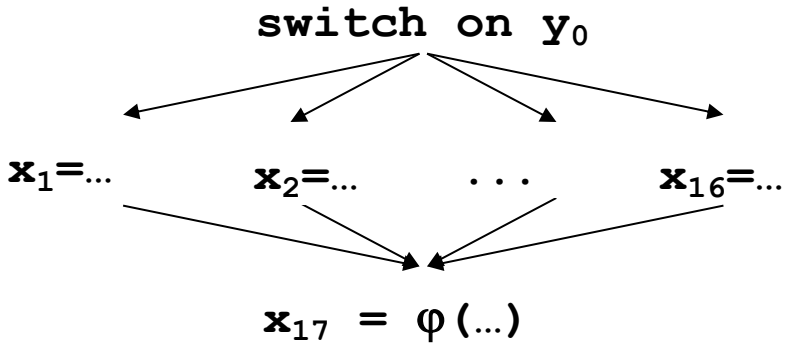
Поэтому φ-функция не прочитает неопределенного значения.

6.1 SSA-форма

6.1.3. Количество аргументов φ -функции

◊ По определению у φ **-функции может быть любое число аргументов**.

Например, на рисунке φ -функция, у которой 16 аргументов



6.2 Построение SSA-формы

6.2.1. Постановка задачи

◊ Для преобразования процедуры в SSA-форму **компилятор** должен:

- вставить в точки сбора необходимые φ -функции для каждой переменной
- переименовать переменные (в том числе и временные) таким образом, чтобы выполнялись следующие два правила:
  (1) каждое определение имеет индивидуальное имя; и
  (2) каждое использование ссылается на единственное определение.

6.2 Построение SSA-формы

6.2.2. Базовый алгоритм построения SSA-формы

◊ **Вход**: программа в промежуточном представлении

◊ **Выход**: промежуточное представление программы в SSA-форме

◊ **Метод**: Выполнить следующие действия:

(1) *Вставить* φ *-функции*:

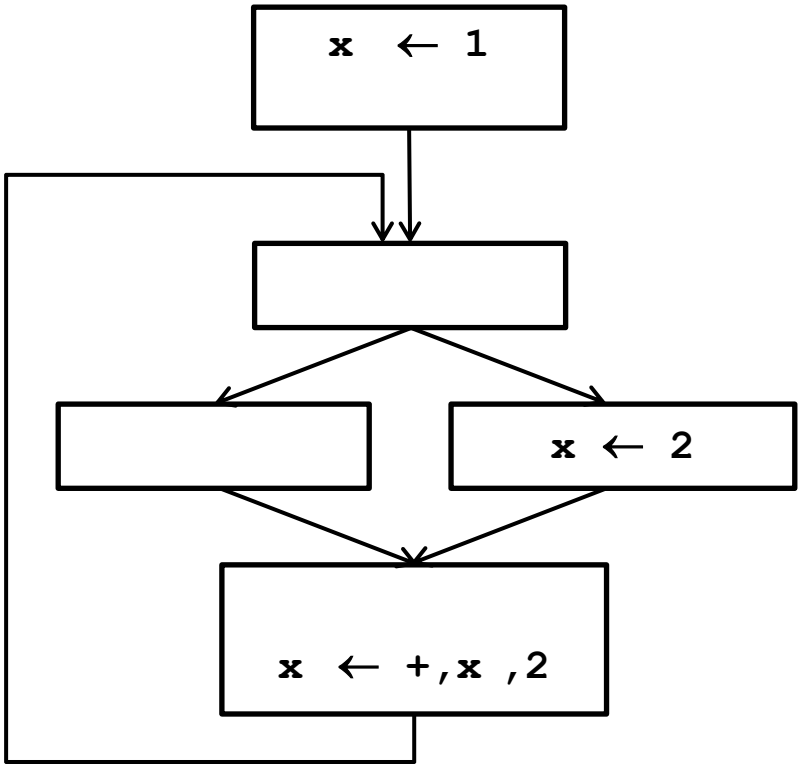
в начало каждого блока B, у которого $|Pred(B)| > 1$ вставить φ -функцию вида $y = \varphi(y, y, \ldots)$ для каждого имени y, которое встречается в функции.

Вставленная φ -функция должна иметь по одному аргументу для каждого $B' \in Pred(B)$:

Порядок вставляемых φ -функций несущественен.

6.2 Построение SSA-формы

6.2.3. Базовый алгоритм построения SSA-формы. Пример



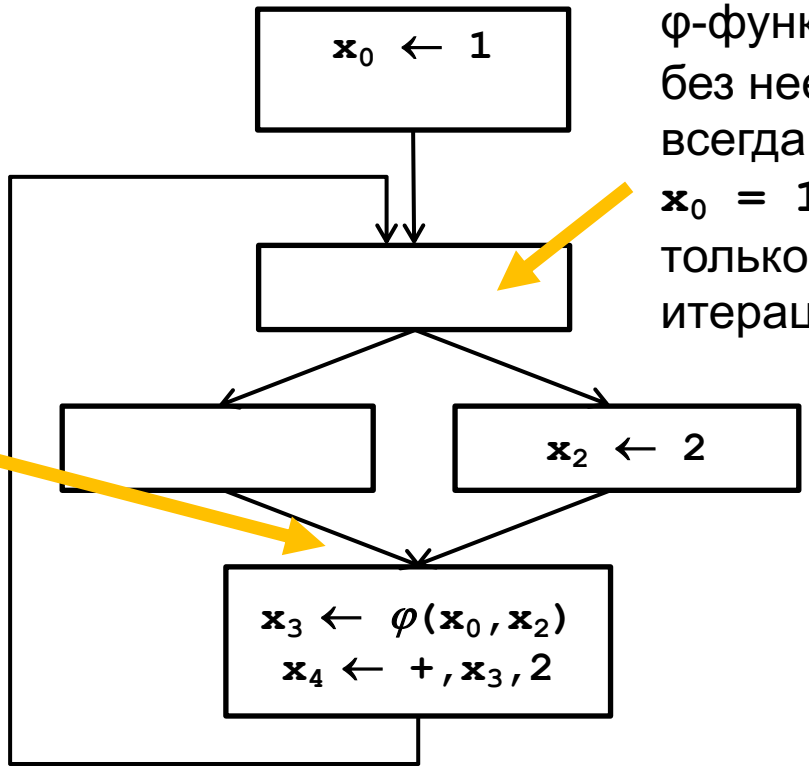
6.2 Построение SSA-формы

6.2.3. Базовый алгоритм построения *SSA*-формы. Пример

(Неправильное решение)

Не хватает еще одной φ-функции от x_0 и x_4: без нее по левой дуге всегда будет приходить $x_0 = 1$, что верно только для 1-й итерации цикла

Без $\varphi(x_0, x_4)$ по левой дуге всегда будет приходить $x_0 = 1$ (что верно только для 1-й итерации цикла). Компилятор может подставить значение $x_0 = 1$ внутрь φ-функции, что приведет к генерации некорректного кода.

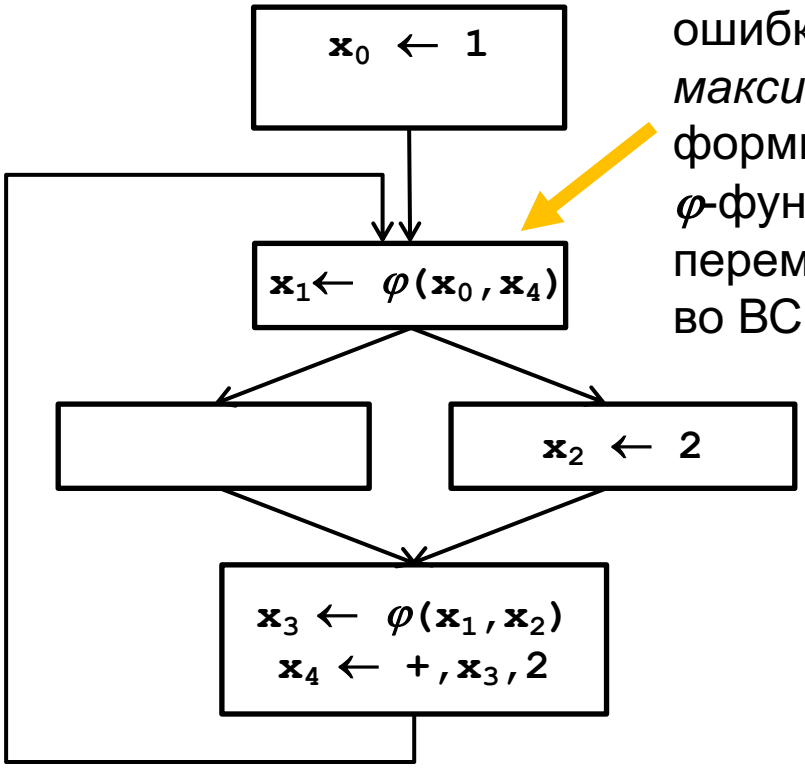


Как можно было избежать этой ошибки: при построении максимальной SSA-формы нужно вставлять φ-функции для ВСЕХ переменных процедуры во ВСЕ точки сбора.

6.2 Построение SSA-формы

6.2.3. Базовый алгоритм построения SSA-формы. Пример

(Правильное решение)



Как избежать такой ошибки: при построении *максимальной* SSA-формы нужно вставлять φ-функции для ВСЕХ переменных процедуры во ВСЕ точки сбора.

6.2 Построение SSA-формы

6.2.2. Базовый алгоритм построения SSA-формы

◊ **Вход**: программа в промежуточном представлении

◊ **Выход**: промежуточное представление программы в SSA-форме

◊ **Метод**: Выполнить следующие действия:

(1) $Вставить\ \varphi\text{-}функции$:

в начало каждого блока B, у которого $|Pred(B)| > 1$ вставить φ-функцию вида $y = \varphi(y, y, \ldots)$ для каждого имени y, которое ~~либо определяется, либо используется в B.~~ **встречается в функции.**

Вставленная φ-функция должна иметь по одному аргументу для каждого $B' \in Pred(B)$:

Порядок вставляемых φ-функций несущественен.

6.2 Построение SSA-формы

6.2.2. Базовый алгоритм построения *SSA*-формы

◊ **Вход**: программа в промежуточном представлении

◊ **Выход**: промежуточное представление программы в *SSA*-форме

◊ **Метод**: Выполнить следующие действия:

(2) *Переименовать переменные*:

(a) вычислить достигающие определения; при этом только одно определение будет достигать любого использования:

это гарантируют вставленные φ-функции, которые тоже являются определениями;

(b) переименовать каждое использование (как основных, так и временных) переменных таким образом, чтобы новое имя соответствовало единственному определению, которое достигает его.

6.2 Построение SSA-формы

6.2.2. Базовый алгоритм построения SSA-формы

◊ **Вход**: программа в промежуточном представлении

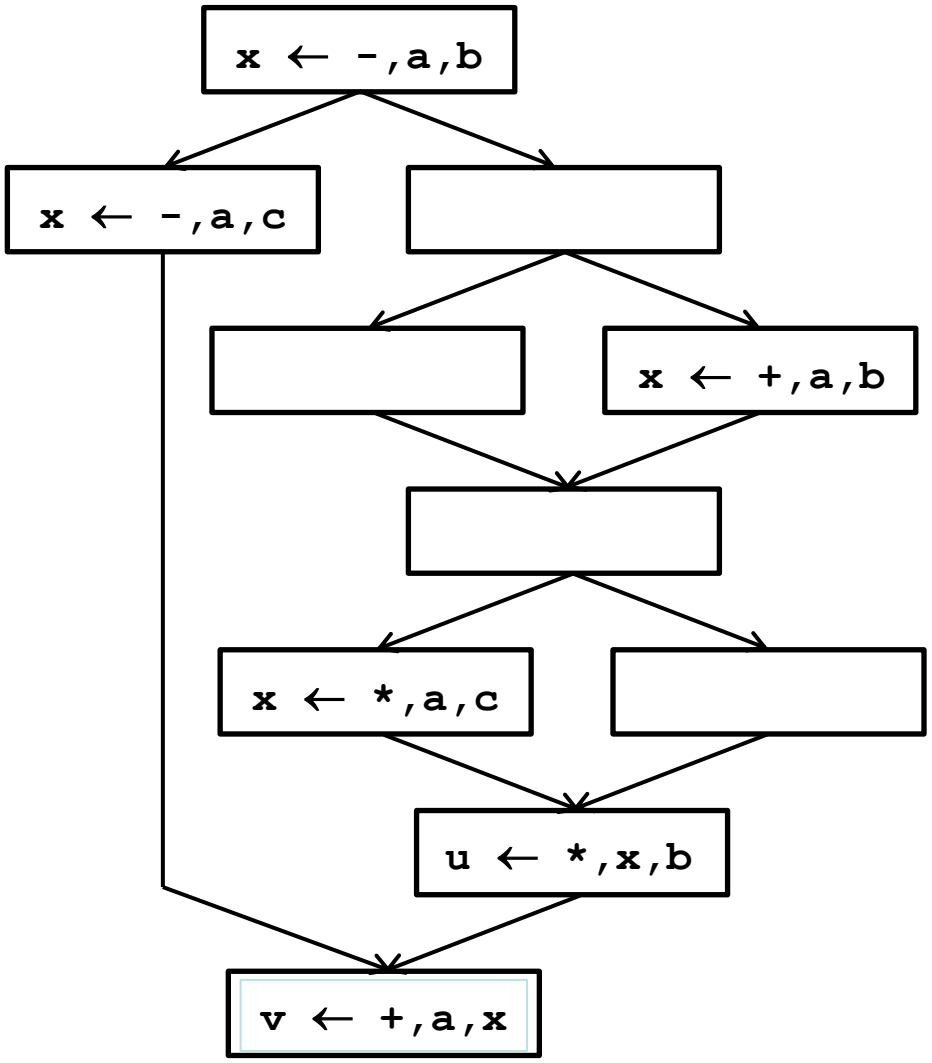
◊ **Выход**: промежуточное представление программы в SSA-форме

◊ **Метод**: Выполнить следующие действия:

(3) *Отсортировать определения*, достигающие каждой φ -функции и для каждой φ -функции обеспечить соответствие имен ее аргументов путям, по которым определения этих аргументов достигают блока, содержащего указанную φ -функцию.

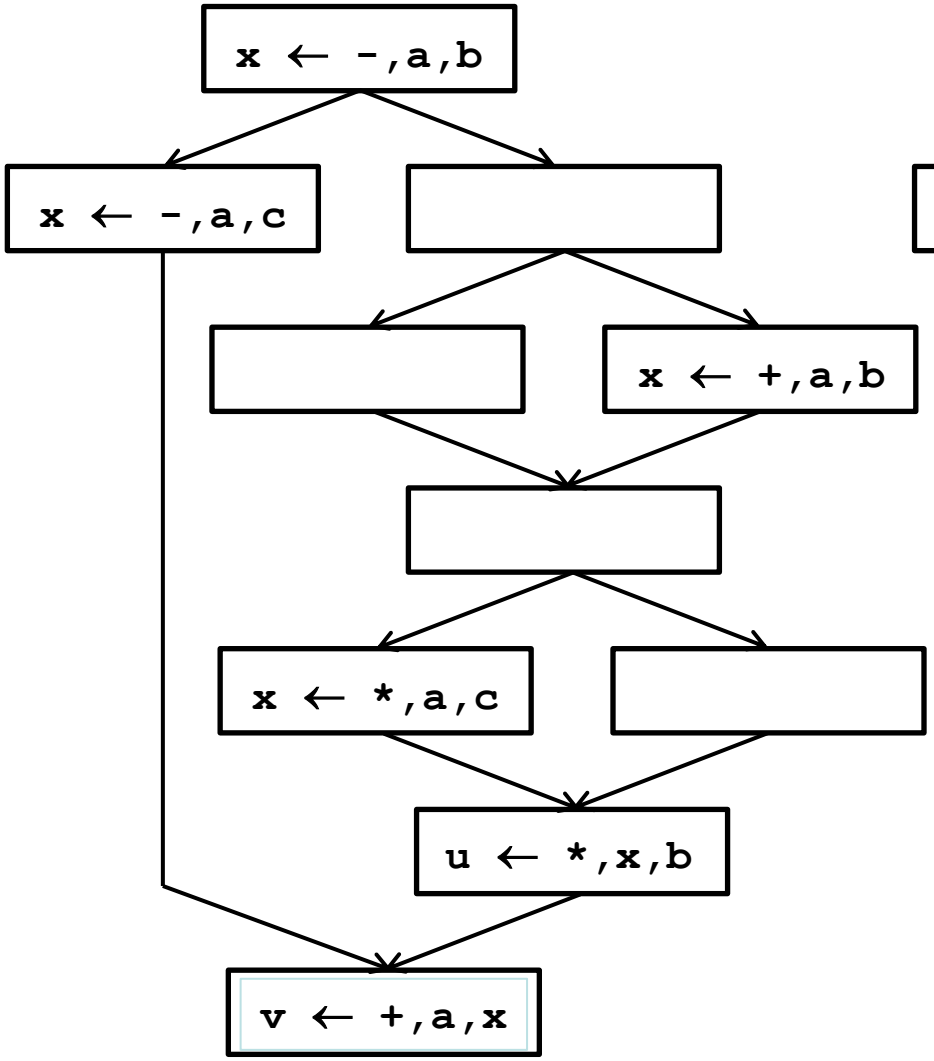
6.2 Построение SSA-формы

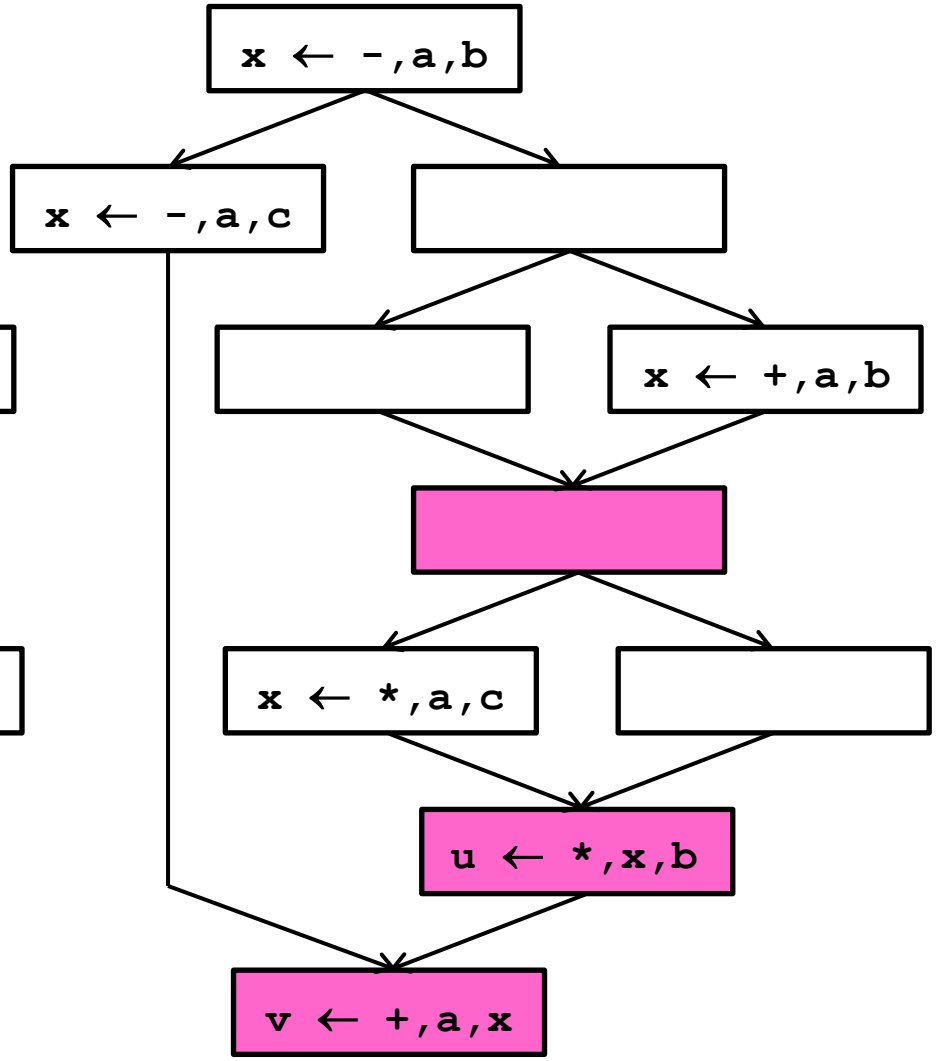
6.2.3. Базовый алгоритм построения *SSA*-формы. Пример



6.2 Построение SSA-формы

6.2.3. Базовый алгоритм построения *SSA*-формы. Пример





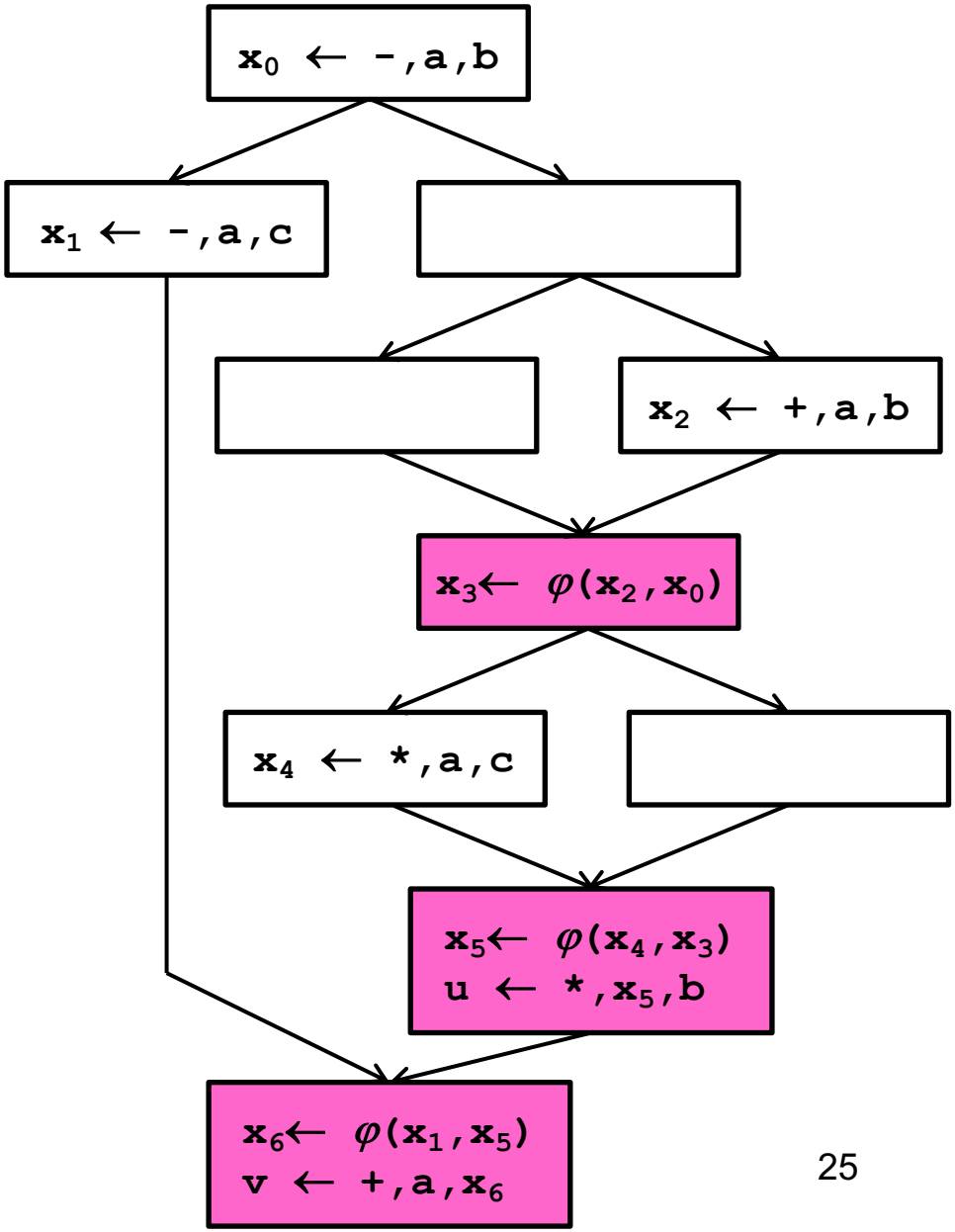
6.2 Построение SSA-формы

6.2.3. Базовый алгоритм построения SSA-формы. Пример

- ◊ Алгоритм вставил ϕ-функцию после каждой точки сбора ГПУ.
- ◊ После преобразования процедуры в SSA-форму

  (1) внутри процедуры каждое определение создает уникальное имя;

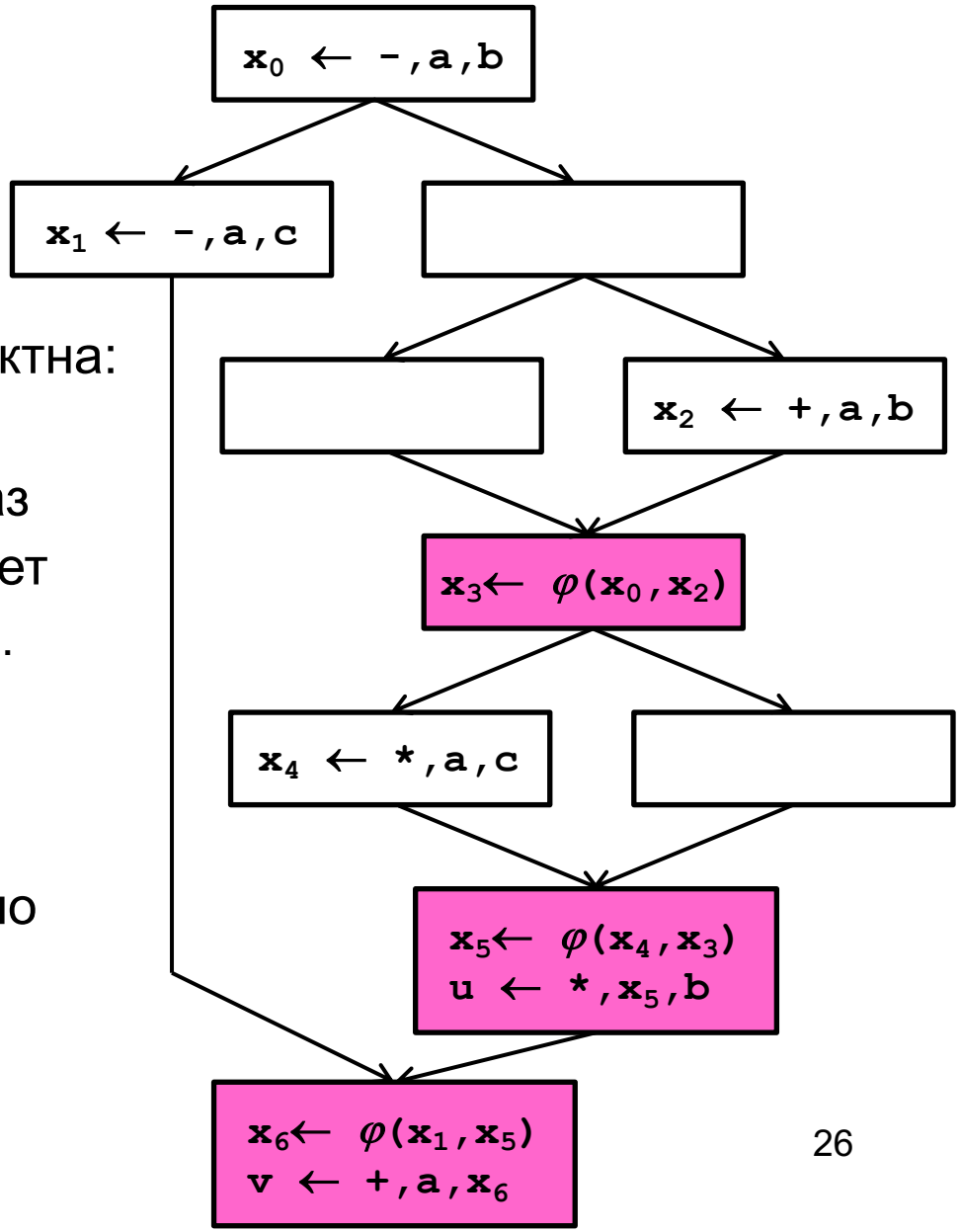
  (2) каждое использование обращается к единственному определению.



6.2 Построение SSA-формы

6.2.3. Базовый алгоритм построения *SSA*-формы. Пример

- Построенная *SSA-форма* корректна:
  - Каждая переменная определяется только один раз
  - Каждое обращение использует имя отдельного определения.
- Построенная *SSA-форма* называется *максимальной SSA-формой,* так как она обычно содержит **намного больше** φ-функций, чем необходимо.



6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи

◊ По определению *частично усеченная SSA-форма* содержит меньше φ-функций, чем максимальная (но, к сожалению, как следует и из ее названия она **не всегда** содержит минимально возможное количество φ-функций).

6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи

◊ По определению *частично усеченная SSA-форма* содержит меньше φ-функций, чем максимальная (но, к сожалению, как следует и из ее названия она содержит не минимально возможное количество φ-функций).

◊ Чтобы не всегда вставлять φ-функции необходимо **для каждой точки сбора** уметь выяснять, какие переменные нуждаются в φ-функциях.

6.3 Построение частично усеченной SSA-формы

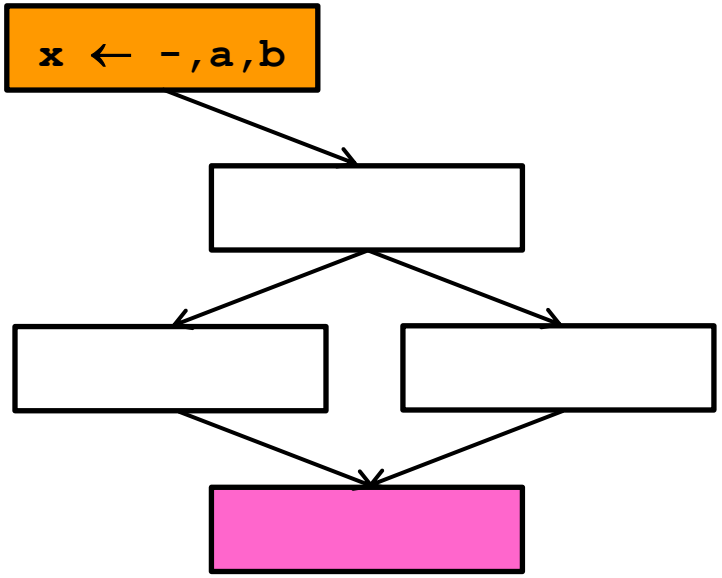
6.3.1. Постановка задачи

◊ По определению *частично усеченная SSA-форма* содержит меньше φ-функций, чем максимальная (но, к сожалению, как следует и из ее названия она содержит не минимально возможное количество φ-функций).

◊ Чтобы не всегда вставлять φ-функции необходимо
для каждой точки сбора уметь выяснять, какие переменные нуждаются в φ-функциях.
Или
для каждого определения переменной уметь находить множество всех точек сбора, которые нуждаются в φ-функциях для значения, порожденного этим определением.

6.3 Построение частично усеченной SSA-формы

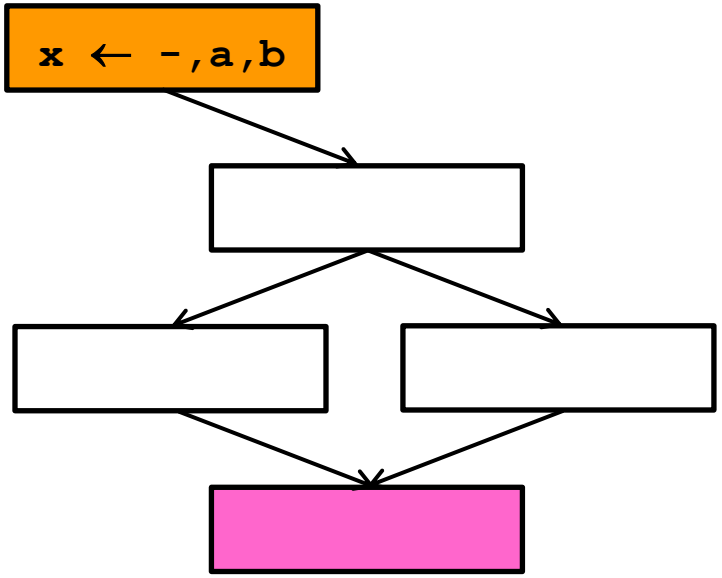
6.3.1. Постановка задачи



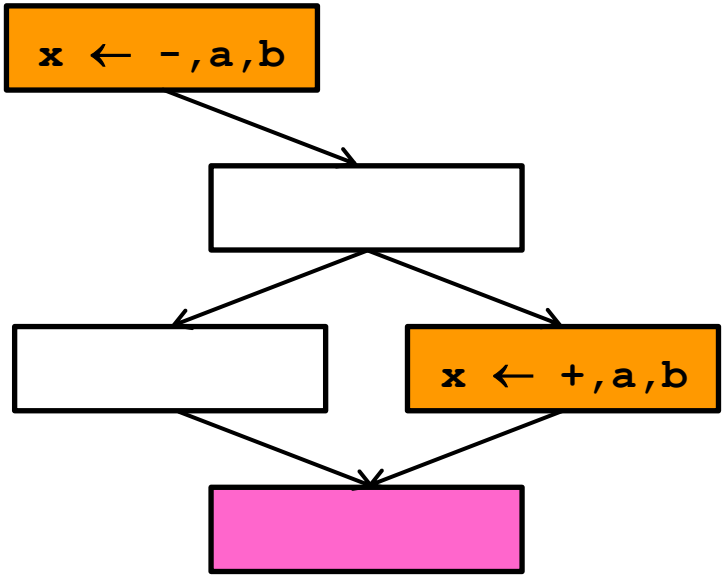
В розовом блоке φ-функция для **x** не нужна, так как и слева, и справа приходит одно и то же значение **x**

6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи



В розовом блоке φ -функция для **x не нужна**, так как и слева, и справа приходит одно и то же значение **x**



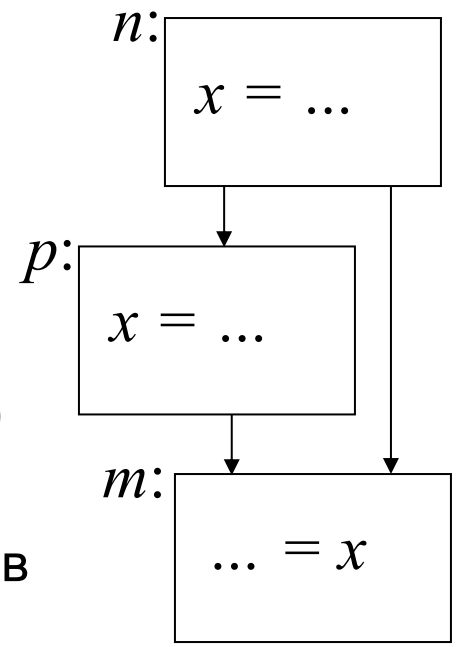
В розовом блоке **нужна** φ -функция для **x** так как справа появился блок, в котором вычисляется еще одно значение **x**

6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи

◊ Пусть $Dom(m)$ – множество доминаторов узла m.

- Если $n \in Dom(m)$, определение x_0 в вершине n не требует φ-функции в узле m, так как каждый путь, который достигает m проходит через n.
- Если $n \in Dom(m)$, единственным вариантом, при котором определение x_0 не достигнет узла m, является вклинивание еще одного определения x в некотором узле $p \notin Dom(m)$, расположенном между n и m.



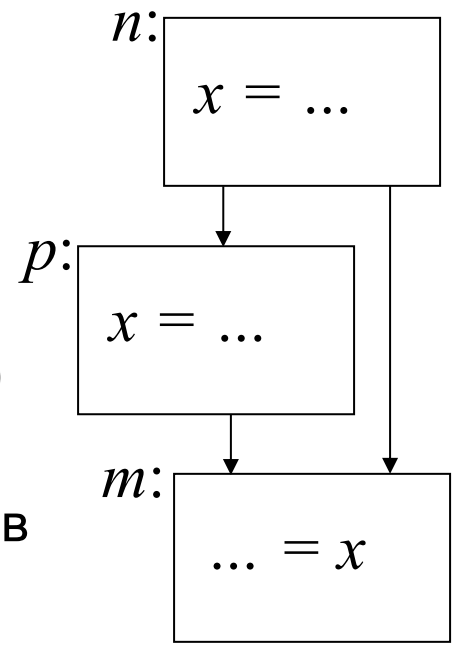
При этом φ-функцию будет требовать не определение x в узле n, а его переопределение в узле p.

6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи

◊ Пусть $Dom(m)$ – множество доминаторов узла m.

Нетрудно заметить, что блок с меткой m – граница доминирования блока с меткой p.



◈ Если $n \in Dom(m)$, единственным вариантом, при котором определение x_0 не достигнет узла m, является вклинивание еще одного определения x в некотором узле $p \notin Dom(m)$, расположенном между n и m.

При этом φ -функцию будет требовать не определение x в узле n, а его переопределение в узле p.

6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи

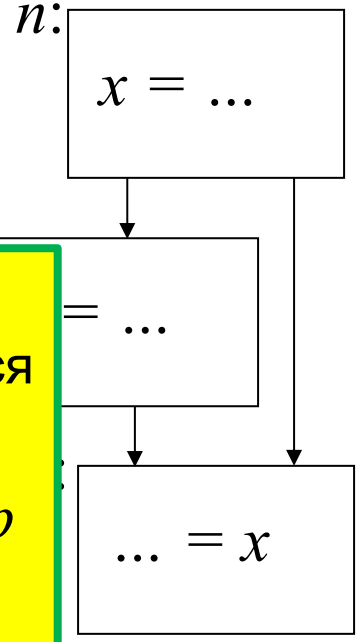
◊ Пусть $Dom(m)$ – множество доминаторов узла m.

◈ Е

в

узле m, так как каждый путь, который достигает m проходит через n.

Нетрудно заметить, что блок с меткой m – граница доминирования блока с меткой p.



Напоминание.

Границей доминирования $DF(n)$ узла n называется множество узлов m, удовлетворяющих условиям:

(1) n является доминатором предшественника p узла m: $p \in Pred(m)$ & $n \in Dom(p)$

(2) n не является строгим доминатором m

$$n \notin (Dom\,(m) - \{m\})$$

не определение x в узле n, а его переопределение в узле p.

6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ-функций

◊ Базовый алгоритм помещал по φ -функции для **каждой** переменной в начало **каждой** вершины сбора.

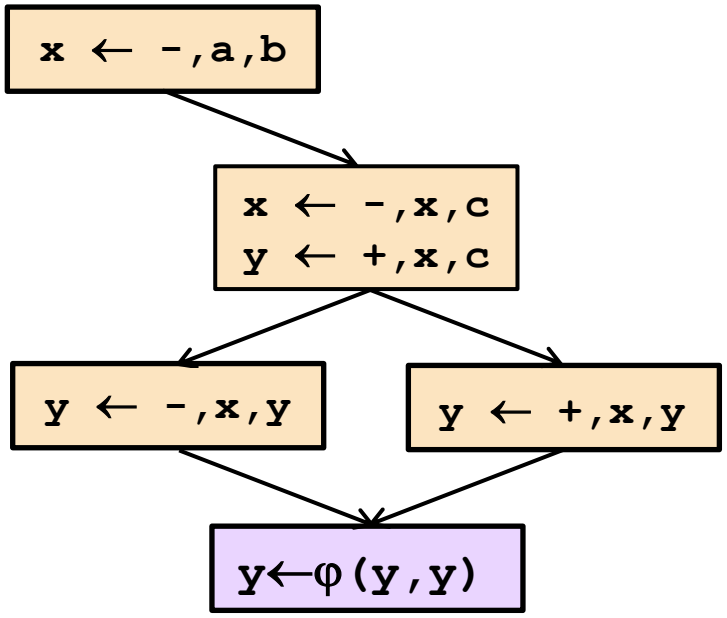
◊ Границы доминирования позволяют более точно определить, какие именно φ -функции необходимы в данной вершине.

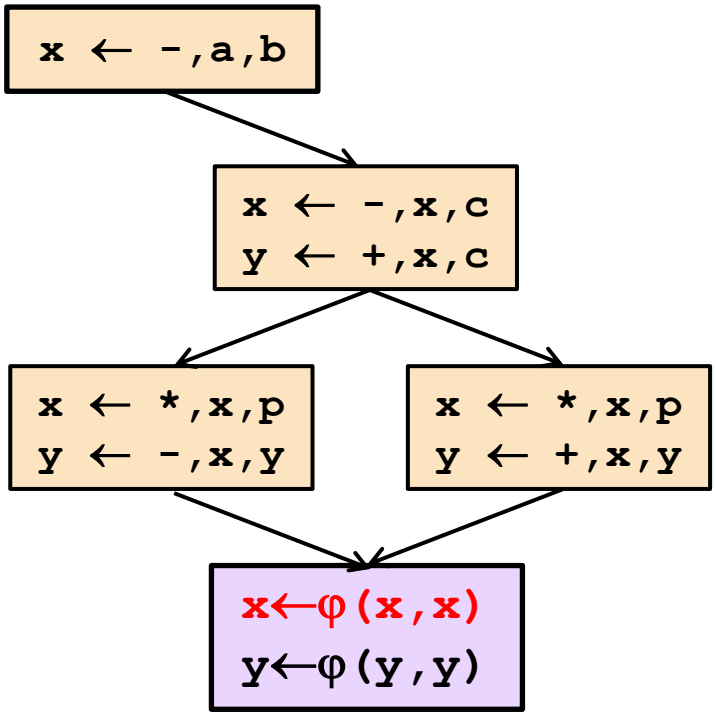
Правило: Определение переменной x в базовом блоке B требует соответствующей φ -функции в каждой вершине из $DF(B)$.

При этом вставленная φ -функция становится новым определением x, так что могут потребоваться новые φ -функции.

6.3 Построение частично усеченной SSA-формы

6.3.1. Постановка задачи. Пример





Какие φ-функции нужны в лиловом блоке?

6.3 Построение частично усеченной SSA-формы

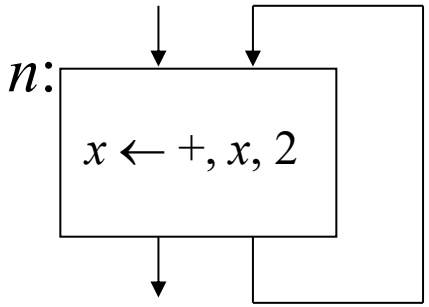
6.3.2. Размещение φ-функций

◊ Базовый алгоритм помещал по φ -функции для **каждой** переменной в начало **каждой** вершины сбора.

◊ Границы доминирования позволяют более точно определить, какие именно φ -функции необходимы в данной вершине.

Правило: Определение переменной x в базовом блоке B требует соответствующей φ -функции в каждой вершине из $DF(B)$.
При этом вставленная φ -функция становится новым определением x, так что могут потребоваться новые φ -функции.

◊ **Пример**. $DF(n) = n$



6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ-функций

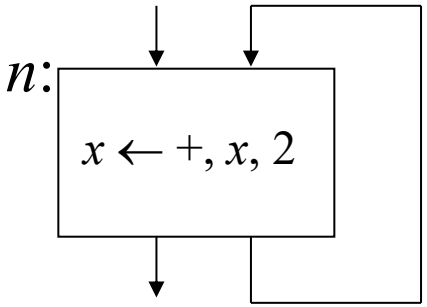
◊ Базовый алгоритм помещал по φ -функции для **каждой** пер

◊ Гра ть, как

Пр требует соо .

При этом вставленная φ -функция становится новым определением x, так что могут потребоваться новые φ -функции.

◊ **Пример**. $DF(n) = n$



Границей доминирования $DF(n)$ узла n называется множество узлов m, удовлетворяющих условиям:

(1) n является доминатором предшественника p узла m: $p \in Pred(m)$ & $n \in Dom(p)$

(2) n не является строгим доминатором m $n \notin (Dom\,(m) - \{m\})$

6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ-функций

◊ Переменная, которая используется только в одном блоке, не может иметь *живой* φ -функции. Поэтому φ -функции нужно вставлять только для *глобальных имен*.

◊ Поэтому перед размещением φ-функций вычисляется множество глобальных имен $Globals$. При вычислении $Globals$ используются результаты анализа живых переменных.

◊ Алгоритм вычисления множества $Globals$ попутно вычисляет для каждого базового блока B множество def_B и для каждой переменной $x \in Globals$ – множество $Blocks(x)$ базовых блоков B, в которых $x \in def_B$.

6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ -функций

Алгоритм построения множеств *Globals* и *Blocks*(*x*)

◊ **Вход**: ГПУ

◊ **Выход**: множество $Globals$,
множества $Blocks(x)$ для каждой переменной x
множества def_B для каждого базового блока B.

◊ **Метод**: Выполнить следующие действия:

6.3 Построение частично усеченной SSA-формы

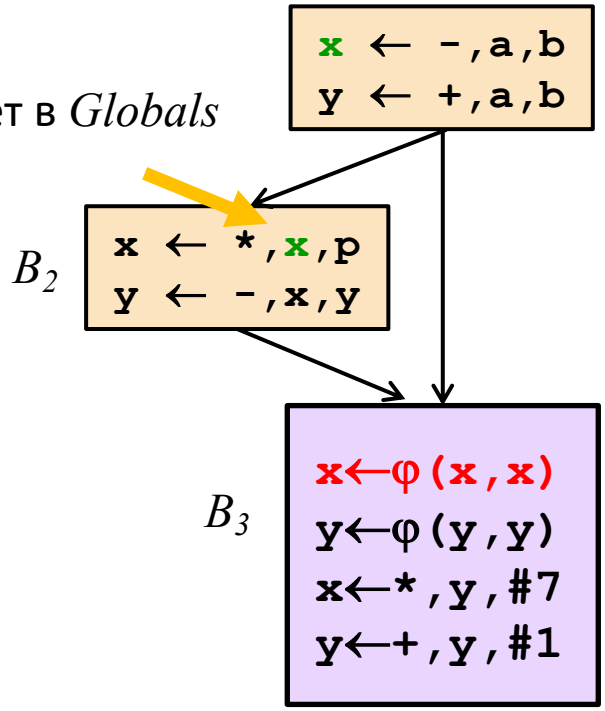
6.3.2. Размещение φ-функций

Алгоритм построения множеств *Globals* и *Blocks*(*x*)

$Globals = \varnothing$;
for each *variable* x **do** $Blocks(x) = \varnothing$;
for each *block* B **do {**
    $def_B = \varnothing$;
    for each *instruction* $i \in B$ **do {**
        // пусть команда i имеет вид: $x \leftarrow op, y, z$
        if $y \notin def_B$ **then** $Globals = Globals \cup \{y\}$;
        if $z \notin def_B$ **then** $Globals = Globals \cup \{z\}$;
        $def_B = def_B \cup \{x\}$;
        $Blocks(x) = Blocks(x) \cup \{B\}$
    }
}

Частично-усеченная и усеченная SSA-формы

x используется в блоке до определения, т.е. x попадет в *Globals*



Переменная, которая используется только в одном блоке, не может иметь *живой* φ-функции.

Усеченная форма: x, определяемая в блоке B_2, не жива за пределами блока, поэтому φ-функция (в B_3) для нее не требуется.
Но для того, чтобы это установить, нужно выполнить анализ живых переменных.

Частично-усеченная форма: Компромиссный вариант – не требует анализа живых переменных, φ-функции вставляются только для *глобальных имен*, т.е. имен, используемых в нескольких базовых блоках.
Но это не гарантирует, что все аргументы φ-функций живы, и может приводить к появлению избыточных φ-функций (например, в B_3)

6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ-функций

Алгоритм размещения φ-функций

- **Вход**: исходный ГПУ
- **Выход**: преобразованный ГПУ
- **Метод**: Выполнить следующие действия

```
for each name x ∈ Globals do {
    WorkList = Blocks(x);
    for each block B ∈ WorkList do {
        for each block D ∈ DF(B) do {
            вставить φ-функцию для x в D;
            WorkList = WorkList ∪ {D};
        };
        WorkList = WorkList – {B};
    }
}
```

Чтобы учесть новое определение x, реализуемое вставленной φ-функцией

6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ-функций

◊ **Замечание 1**. Для каждого блока $B \in WorkList$ алгоритм вставляет φ -функцию в начало каждого блока $D \in DF(B)$.

Порядок φ -функций роли не играет.

После вставления φ -функции для x в блок D алгоритм добавляет D в $WorkList$, чтобы в дальнейшем учесть новое определение x в блоке D.

6.3 Построение частично усеченной SSA-формы

6.3.2. Размещение φ-функций

◊ **Замечание 2**. Для улучшения эффективности алгоритма необходимо избегать следующих двух видов дублирования:

(1) помещение какого-либо блока в $WorkList$ более одного раза для каждого глобального имени; для этого можно вести список уже обработанных блоков (например, не удалять B из $WorkList$, а помечать его как уже обработанный).

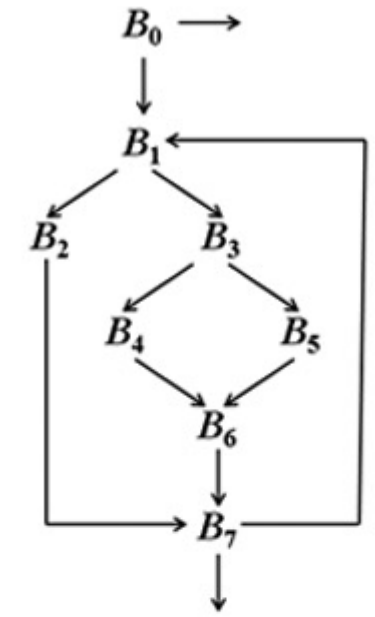
(2) рассматриваемый блок может входить в состав границ доминирования нескольких блоков, входящих в $WorkList$; чтобы избежать вставления дублирующих ϕ-функций для переменной x, можно использовать список блоков, которые уже содержат ϕ-функции для x, что быстрее, чем проверять каждый раз какие ϕ-функции включены.

6.3 Построение частично усеченной SSA-формы

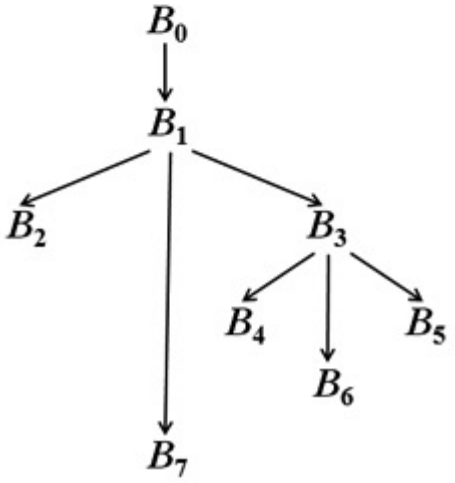
6.3.3. Размещение φ-функций. Пример.

B_0	i ← 1	B_5	c ←
B_1	a ← b ←	B_3	a ← d ←
B_2	b ← c ← d ←	B_7	y ← +, a, b z ← +, c, d i ← +, i, 1
B_6	b ←	B_4	d ←

Граф потока управления



Дерево доминаторов



6.3 Построение частично усеченной SSA-формы

6.3.3. Размещение φ-функций. Пример.

◊ Сначала вычисляются множества *Globals* и *Blocks(x)*:

$Globals = \{a, b, c, d, i\}$.

Множества $Blocks(x)$ представлены в таблице:

a	b	c	d	i
$\{B_1, B_3\}$	$\{B_1, B_2, B_6\}$	$\{B_2, B_5\}$	$\{B_2, B_3, B_4\}$	$\{B_0, B_7\}$

◊ Алгоритму размещения φ-функций потребуются также уже вычисленные границы доминирования

n	0	1	2	3	4	5	6	7
$DF(B_n)$	$\varnothing$	$\{B_1\}$	$\{B_7\}$	$\{B_7\}$	$\{B_6\}$	$\{B_6\}$	$\{B_7\}$	$\{B_1\}$

6.3 Построение частично усеченной SSA-формы

6.3.3. Размещение φ-функций. Пример.

◊ Применяем алгоритм размещения φ-функций.

Берем первую переменную из $Globals = \{a, b, c, d, i\}$, т.е . a

$Blocks(a) = \{B_1, B_3\}$, так что алгоритм должен вставить

φ-функцию для a в каждый блок из границ доминирования

$DF(B_1) = \{B_1\}$ и $DF(B_3) = \{B_7\}$.

Так, вставив φ-функцию для a в B_7, получим новое определение a – φ-функцию, вставленную в B_7. Следовательно, необходимо включить B_7 в $WorkList$ и вставить φ-функцию для a в каждый блок из $DF(B_7) = \{B_1\}$. Но B_1 уже имеется в $WorkList$, так что, $DF(B_7)$ не добавляется к $WorkList$.

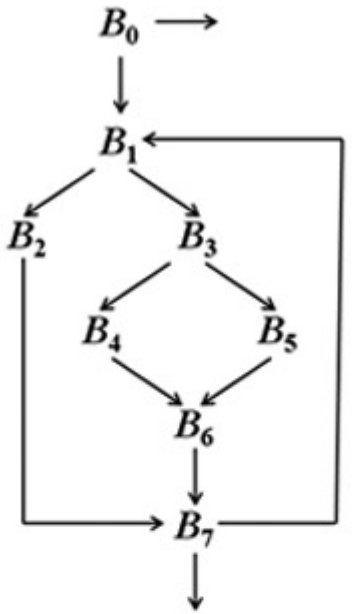
x	a	b	c	d	i
Insert $\varphi(x)$	$\{B_7, B_1\}$	$\{B_7, B_1\}$	$\{B_7, B_1, B_6\}$	$\{B_7, B_1, B_6\}$	$\{B_1\}$

6.3 Построение частично усеченной SSA-формы

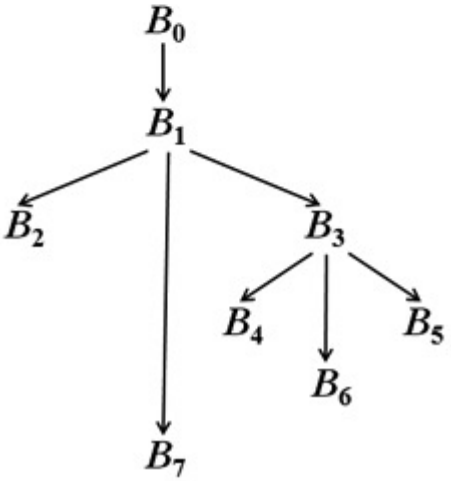
6.3.3. Размещение φ-функций. Пример.

B_0	i ← 1	B_5	c ←
B_1	a ← φ(a,a) b ← φ(b,b) c ← φ(c,c) d ← φ(d,d) i ← φ(i,i) a ← b ←	B_7	a ← φ(a,a) b ← φ(b,b) c ← φ(c,c) d ← φ(d,d) y ← +, a, b z ← +, c, d i ← +, i, 1
B_2	b ← c ← d ←	B_3	a ← d ←
B_4	d ←	B_6	c ← φ(c,c) d ← φ(d,d) b ←

Граф потока управления



Дерево доминаторов



6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Алгоритм

◊ **Вход**: программа с размещенными φ -функциями

◊ **Выход**: программа, в которой каждой переменной сопоставлено ее *SSA-имя*.

◊ **Метод**: Сначала (в основном алгоритме) инициализируются стеки и счетчики, после чего из корня дерева доминаторов n_0 вызывается рекурсивная функция *Rename*.

Rename обрабатывает блок, рекурсивно вызывая его последователей по дереву доминаторов.

Закончив обрабатывать очередной блок, *Rename* выталкивает из стеков все имена, помещенные в них во время обработки блока.

Функция *NewName*, манипулируя со счетчиками и стеками, в случае необходимости создает новые имена.

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Алгоритм

- Основной алгоритм:

```
for each i ∈ Globals do {
        counter[i] = 0;
        stack[i] = ∅;
};
Rename(n0);
```

- Функция **NewName**:

```
NewName(n) { // n – имя переменной
        i = counter[n];
        counter[n] += 1;
        Push ni onto stack[n];
              return ni
}
```

6.3 Построение частично усеченной SSA-формы
6.3.7. Переименование переменных.

Функция $Rename(B)$:

```
for each φ-function ∈ B: x = φ(…) do
     rename x as NewName(x);
for each instruction ∈ B: x ← op, y, z do{
     rewrite y as top(stack[y]);
     rewrite z as top(stack[z]);
     rewrite x as NewName(x);
for each successor of B in the flowgraph do
     fill in φ-function parameters;
for each successor S of B in the
     dominator tree do Rename(S)
for each φ-function ∈ B: x = φ(…)do
          Pop(stack[x]);
for each instruction ∈ B: x ← op, y, z do
          Pop(stack[x]);
```

6.3 Построение частично усеченной SSA-

6.3.7. Переименование переменных.

```
NewName(n) {
  i = counter[n];
  counter[n] += 1;
    Push n_i onto
stack[n];
  return n_i}
```

Функция $Rename(B)$:

```
for each φ-function ∈ B: x = φ
     rename x as NewName(x);
for each instruction ∈ B: x ← op, y, z do{
     rewrite y as top(stack[y]);
     rewrite z as top(stack[z]);
     rewrite x as NewName(x);
for each successor of B in the flowgraph do
     fill in φ-function parameters;
for each successor S of B in the
     dominator tree do Rename(S)
for each φ-function ∈ B: x = φ(…)do
          Pop(stack[x]);
for each instruction ∈ B: x ← op, y, z do
          Pop(stack[x]);
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Применим алгоритм переименования к рассматриваемому примеру в предположении, что на входе в блок B_0 определены имена a_0, b_0, c_0, d_0.

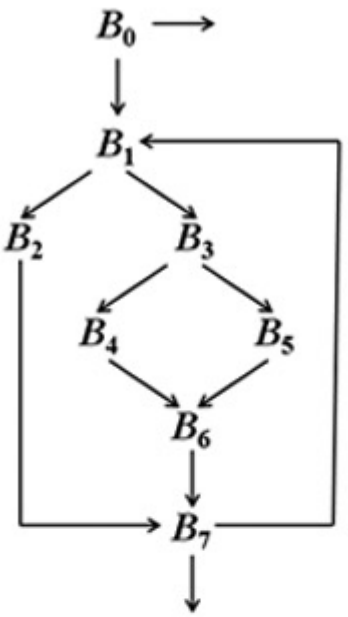
B_0 `i ← 1`	B_5 `c ←`

B_1 `a ← φ(a,a)`
`b ← φ(b,b)`
`c ← φ(c,c)`
`d ← φ(d,d)`
`i ← φ(i,i)`
`a ←`
`b ←`
	B_7 `a ← φ(a,a)`
	`b ← φ(b,b)`
	`c ← φ(c,c)`
	`d ← φ(d,d)`
	`y ← +, a, b`
	`z ← +, c, d`
	`i ← +, i, 1`

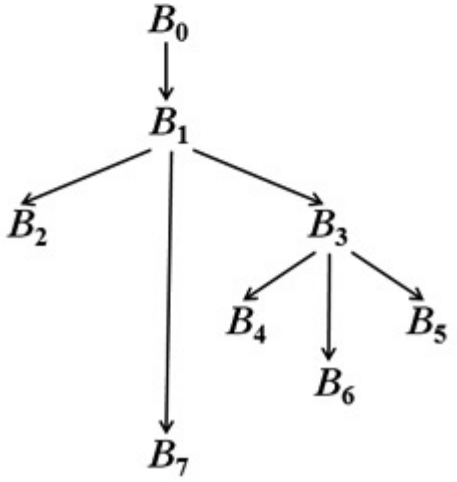
B_2 `b ←`
`c ←`
`d ←`
	B_3 `a ←`
	`d ←`

B_4 `d ←`	B_6 `c ← φ(c,c)`
	`d ← φ(d,d)`
	`b ←`

Граф потока управления



Дерево доминаторов



6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

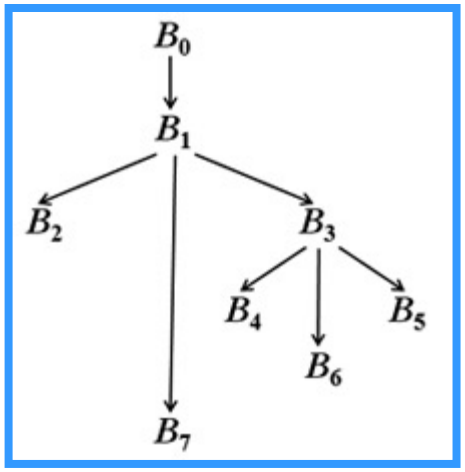
◊ Корнем обрабатываемой части дерева доминаторов является блок B_0. Поэтому «***Основной алгоритм***», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $\boldsymbol{Rename(B_0)}$.

◊ Порядок обработки базовых блоков определяется деревом доминаторов

- ◈ первым обрабатывается блок $\boldsymbol{B_0}$.

```
B0    i ← 1
```

- ◈ во время обработки $\boldsymbol{B_0}$ будет вызван $\boldsymbol{Rename(B_1)}$,
- ◈ во время обработки $\boldsymbol{B_1}$ будут вызваны $\boldsymbol{Rename(B_2)}$, $\boldsymbol{Rename(B_7)}$ и $\boldsymbol{Rename(B_3)}$,
- ◈ во время обработки $\boldsymbol{B_3}$ будет вызван $\boldsymbol{Rename(B_4)}$, $\boldsymbol{Rename(B_6)}$, $\boldsymbol{Rename(B_5)}$



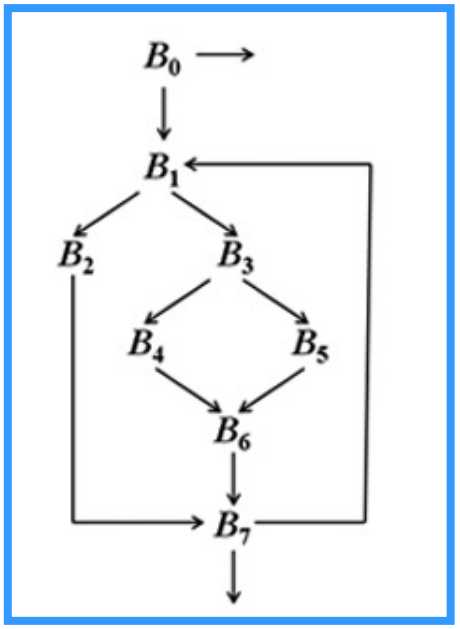
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

◊ Корнем обрабатываемой части дерева доминаторов является блок B_0. Поэтому «***Основной алгоритм***», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $\boldsymbol{Rename(B_0)}$.

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки (↓)	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{c}_0$	$\mathtt{d}_0$	

B_0: `i ← 1;`



◊ $\boldsymbol{Rename(B_0)}$:

- Вызов $NewName$ **(i)** → возвращает имя $\mathtt{i}_0$
  - Замена `i ← 1;` на $\mathtt{i}_0$ `← 1;`
  - Занесение $\mathtt{i}_0$ в стек для i
  - Увеличение счетчика для i на 1
- Заполнение параметров φ-функций в $\boldsymbol{B_1}$
- Вызов $\boldsymbol{Rename(B_1)}$

B_0: $\mathtt{i}_0$ `← 1;`

6.3 Построение частично усеченной SSA-формы

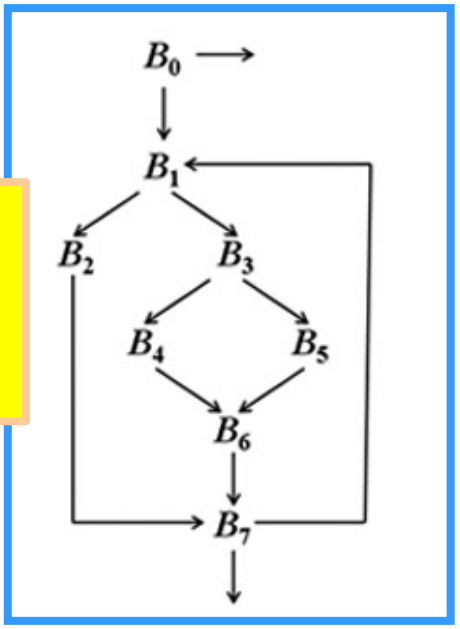
6.3.4. Переименование переменных

◊ Корнем обрабатываемой части дерева доминаторов является блок B_0. Поэтому «***Основной алгоритм***», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $\boldsymbol{Rename(B_0)}$.

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки (↓)	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{c}_0$	$\mathtt{d}_0$	

B_0: `i ← 1;`

Счетчики для переменных a, b, c, d имеют значение 1, а не 0, так как считается, что этим переменным были присвоены значения до входа в блок B_0



Занесение $\mathtt{i}_0$ в стек для i

Увеличение счетчика для i на 1

◈ Заполнение параметров φ-функций в $\boldsymbol{B_1}$

◈ Вызов $\boldsymbol{Rename(B_1)}$

B_0: `i`$_0$ `← 1;`

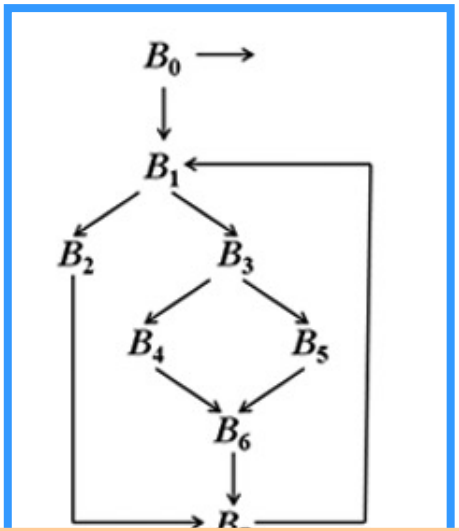
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

◊ Корнем обрабатываемой части дерева доминаторов является блок B_0. Поэтому «***Основной алгоритм***», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $\boldsymbol{Rename(B_0)}$.

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки (↓)	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{c}_0$	$\mathtt{d}_0$	

B_0: `i ← 1;`



◊ $\boldsymbol{Rename(B_0)}$:

- Вызов $NewName$ **(i)** → возвращает имя $\mathtt{i}_0$
  - Замена `i ← 1;` на $\mathtt{i}_0$ `← 1;`
  - Занесение $\mathtt{i}_0$ в стек для i
  - Увеличение счетчика для i на 1
- Заполнение параметров φ-функций в $\boldsymbol{B_1}$
- Вызов $\boldsymbol{Rename(B_1)}$

$B_1 = Succ(B_0)$ по ГПУ

B_0: $\mathtt{i}_0$ `← 1;`

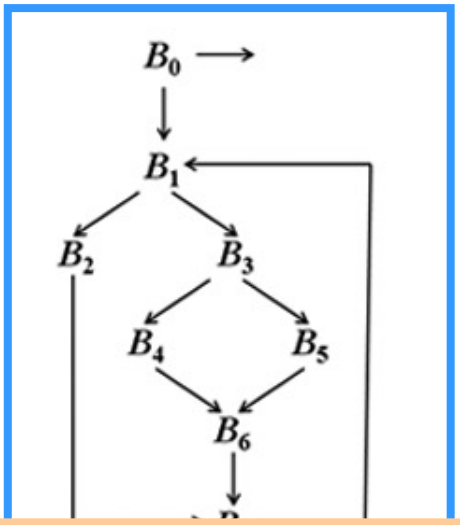
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

◊ Корнем обрабатываемой части дерева доминаторов является блок B_0. Поэтому «***Основной алгоритм***», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов ***Rename*(B_0)**.

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки (↓)	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{c}_0$	$\mathtt{d}_0$	

B_0: `i ← 1;`



◊ ***Rename*(B_0)**:

- Вызов *NewName* **(i)** → возвращает имя $\mathtt{i}_0$
  - Замена `i ← 1;` на $\mathtt{i}_0$ `← 1;`
  - Занесение $\mathtt{i}_0$ в стек для i
  - Увеличение счетчика для i на 1
- Заполнение параметров φ-функций в B_1
- Вызов ***Rename*(B_1)**

$B_1 = Succ(B_0)$ по ГПУ

$B_1 = Succ(B_0)$ по дереву доминаторов

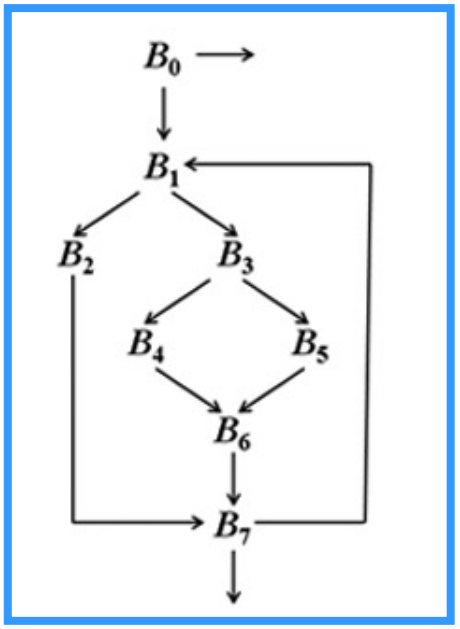
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

◊ Корнем обрабатываемой части дерева доминаторов является блок B_0. Поэтому «***Основной алгоритм***», обнулив счетчики и опустошив стеки для переменных из множества $Globals = \{a, b, c, d, i\}$, сделает вызов $\boldsymbol{Rename(B_0)}$.

Вход в B_0	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки (↓)	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{c}_0$	$\mathtt{d}_0$	

B_0: `i ← 1;`



◊ $\boldsymbol{Rename(B_0)}$:

- Вызов *NewName* **(i)** → возвращает имя $\mathtt{i}_0$
  - Замена `i ← 1;` на $\mathtt{i}_0$ `← 1;`
  - Занесение $\mathtt{i}_0$ в стек для i
  - Увеличение счетчика для i на 1
- Заполнение параметров φ-функций в $\boldsymbol{B_1}$
- Вызов $\boldsymbol{Rename(B_1)}$

B_0: $\mathtt{i}_0$ `← 1;`

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_1	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки (↓)	a_0	b_0	c_0	d_0	i_0

◊ Работа ***Rename***(B_1):

- Вызов *NewName* **(a)** → a_1
- Вызов *NewName* **(b)** → b_1
- Вызов *NewName* **(c)** → c_1
- Вызов *NewName* **(d)** → d_1
- Вызов *NewName* **(i)** → i_1
- Вызов *NewName* **(a)** → a_2
- Вызов *NewName* **(c)** → c_2
- Заполнение параметров φ-функций в B_2 отсутствует, так как в B_2 нет φ-функций
- Вызов ***Rename***(B_2)

B_1:

```
a ← φ(a0, a);
b ← φ(b0, b);
c ← φ(c0, c);
d ← φ(d0, d);
i ← φ(i0, i);
a ← ...;
c ← ...;
(a < c);
```

↓ ***Rename***(B_1)

B_1:

```
a1 ← φ(a0, a);
b1 ← φ(b0, b);
c1 ← φ c0, c);
d1 ← φ(d0, d);
i1 ← φ(i0, i);
a2 ← ...;
c2 ← ...;
(a2 < c2);
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_1	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки (↓)	a_0	b_0	c_0	d_0	i_0

B_1:
```
a ← φ(a0, a);
b ← φ(b0, b);
c ← φ(c0, c);
d ← φ(d0, d);
i ← φ(i0, i);
```

Номера левых параметров φ-функций соответствуют номерам переменных, определяемым в блоке B_0

- Вызов *NewName* (**b**) → $\mathbf{b_1}$
- Вызов *NewName* (**c**) → $\mathbf{c_1}$
- Вызов *NewName* (**d**) → $\mathbf{d_1}$
- Вызов *NewName* (**i**) → $\mathbf{i_1}$
- Вызов *NewName* (**a**) → $\mathbf{a_2}$

Номера присвоены функцией *NewName*

- Заполнение параметров φ-функций в B_2 отсутствует, так как в B_2 нет φ-функций
- Вызов ***Rename***(B_2)

Rename(B_1)

B_1:
```
a1 ← φ(a0, a);
b1 ← φ(b0, b);
c1 ← φ c0, c);
d1 ← φ(d0, d);
i1 ← φ(i0, i);
a2 ← ...;
c2 ← ...;
(a2 < c2);
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_1	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки (↓)	a_0	b_0	c_0	d_0	i_0

B_1:
```
a ← φ(a0, a);
b ← φ(b0, b);
c ← φ(c0, c);
d ← φ(d0, d);
i ← φ(i0, i);
a ← ...;
```

◊ Работа ***Rename*(B_1)**:

- Вызов *NewName* (a) → a_1
- Вызов *NewName* (b) → b_1
- Вызов *NewName* (c) → c_1
- Вызов *NewName* (d) → d_1
- Вызов *NewName* (i) → i_1
- Вызов *NewName* (a) → a_2

```
NewName(n) {
  i = counter[n];
  counter[n] += 1;
  Push ni onto stack[n];
  return ni}
```

***me*(B_1)**

Номера присвоены функцией *NewName*

- Заполнение параметров φ-функций в B_2 отсутствует, так как в B_2 нет φ-функций
- Вызов ***Rename*(B_2)**

B_1:
```
a1 ← φ(a0, a);
b1 ← φ(b0, b);
c1 ← φ c0, c);
d1 ← φ(d0, d);
i1 ← φ(i0, i);
a2 ← ...;
c2 ← ...;
(a2 < c2);
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_2

	a	b	c	d	i
Счетчики	3	2	3	2	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

B_2 до чистки

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	
			c_3		

```
NewName(n) {
  i = counter[n];
  counter[n] += 1;
  Push n_i onto stack[n];
  return n_i}
```

B_2:
```
b ← ...;
c ← ...;
d ← ...;
```

↓ ***Rename*(B_2)**

B_2:
```
b2 ← ...;
c3 ← ...;
d2 ← ...;
```

Работа ***Rename*(B_2)**:

- Вызов *NewName* **(b)** → b_2
- Вызов *NewName* **(c)** → c_3
- Вызов *NewName* **(d)** → d_2
- Заполнение параметров φ-функций в B_7
- Чистка стеков
- Возврат в ***Rename*(B_1)**

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_2

	a	b	c	d	i
Счетчики	3	2	3	2	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	

B_2 до чистки

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	
			c_3		

B_2:
```
b ← . . . ;
c ← . . . ;
d ← . . . ;
```

↓ ***Rename*(B_2)**

B_2:
```
b2 ← . . . ;
c3 ← . . . ;
d2 ← . . . ;
```

Работа ***Rename*(B_2)**:

- Вызов *NewName* (**b**) → b_2
- Вызов *NewName* (**c**) → c_3
- Вызов *NewName* (**d**) → d_2
- Заполнение параметров φ-функций в B_7
- Чистка стеков
- Возврат в ***Rename*(B_1)**

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

B_2 до чистки

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
		b_2	c_2	d_2	

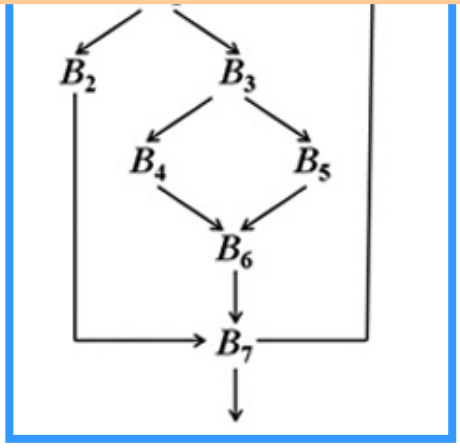
Работа ***Rename*(B_2)**:

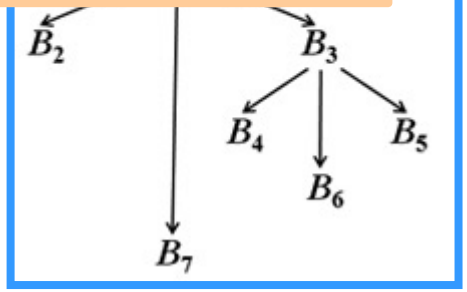
- Вызов *NewName* **(b)** → b_2
- Вызов *NewName* **(c)** → c_3
- Вызов *NewName* **(d)** → d_2
- Заполнение параметров φ-функций в B_7
- **Чистка стеков**
- Возврат в ***Rename*(B_1)**

Как только имя переменной, определенное в B_i было использовано во всех блоках из $Succ(B_i)$, оно выбрасывается из стека, так как больше не понадобится (**Чистка стеков**)

чистки

Счетчики	3	3	4	3	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
			c_2		



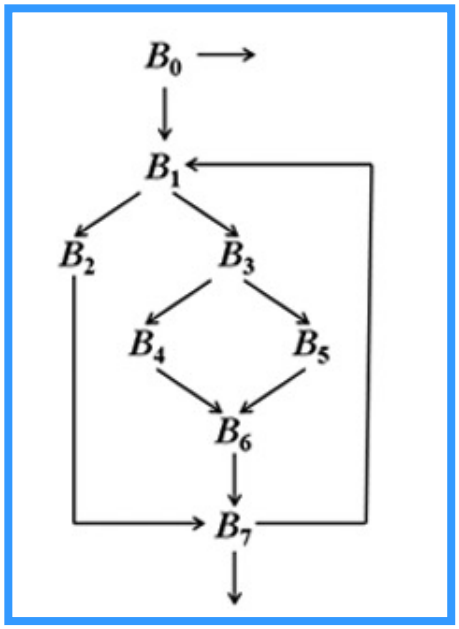


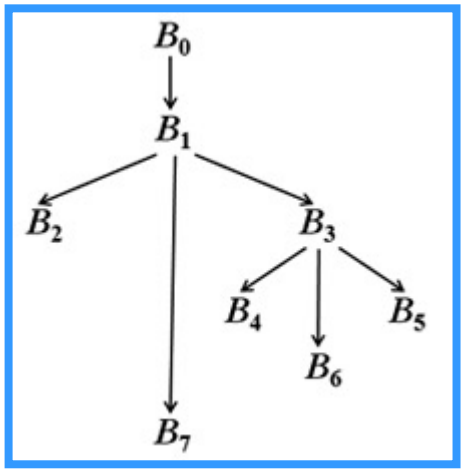
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

После B_2

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
			c_2		





- Работа $\boldsymbol{Rename(B_1)}$:
  - Заполнение параметров φ-функций $\boldsymbol{B_7}$ проведено в $\boldsymbol{B_2}$
  - Вызов $\boldsymbol{Rename(B_7)}$

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_7	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
			c_2		

B_7:
```
a ← φ(a2, a);
b ← φ(b2, b);
c ← φ(c3, c);
d ← φ(d2, d);
x ← a + b;
z ← c + d;
i ← i + 1;
(i ≤ 100)…;
```

◊ Работа ***Rename***(B_7):

- Вызов *NewName* **(a)** → a_3;
- Вызов *NewName* **(b)** → b_3;
- Вызов *NewName* **(c)** → c_4;
- Вызов *NewName* **(d)** → d_3;
- Вызов *NewName* **(i)** → i_2;
- Переименование операндов;
- **x** и **z** не входят в *Globals* и не переименовываются
-

Rename(B_7)

B_7:
```
a3 ← φ(a2, a);
b3 ← φ(b2, b);
c4 ← φ(c3, c);
d3 ← φ(d2, d);
x ← a3 + b3;
z ← c4 + d3;
i2 ← i1 + 1;
(i2 ≤ 100)…;
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

B_7 до чистки

	a	b	c	d	i
Счетчики	4	4	5	4	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	i_2
			c_4		

B_7:

```
a3 ← φ(a2, a);
b3 ← φ(b2, b);
c4 ← φ(c3, c);
d3 ← φ(d2, d);
x  ← a3 + b3;
z  ← c4 + d3;
i2 ← i1 + 1;
(i2 ≤ 100)…;
```

◊ Работа ***Rename*(B_7)**:

- ◊ Заполнение параметров φ-функций в B_1
- ◊ **Чистка стеков**
- ◊ Возврат в ***Rename*(B_1);**

B_1:

```
a1 ← φ(a0, a3);
b1 ← φ(b0, b3);
c1 ← φ(c0, c4);
d1 ← φ(d0, d3);
i1 ← φ(i0, i2);
a2 ← ...;
c2 ← ...;
(a2 < c2);
```

Значения a_3, b_3, c_4, d_3 и i_2 уже определены и (по определению *SSA*) больше определяться не будут, поэтому они удаляются из стеков

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

B_7 до чистки

	a	b	c	d	i
Счетчики	4	4	5	4	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	i_2
			c_4		

B_7:

```
a3 ← φ(a2, a);
b3 ← φ(b2, b);
c4 ← φ(c3, c);
d3 ← φ(d2, d);
x  ← a3 + b3;
z  ← c4 + d3;
i2 ← i1 + 1;
(i2 ≤ 100)…;
```

◊ Работа ***Rename*(B_7)**:

- ◊ Заполнение параметров φ-функций в B_1
- ◊ **Чистка стеков**
- ◊ Возврат в ***Rename*(B_1);**

$B_1 \in Succ(B_7)$

B_1:

```
a1 ← φ(a0, a3);
b1 ← φ(b0, b3);
c1 ← φ(c0, c4);
d1 ← φ(d0, d3);
i1 ← φ(i0, i2);
a2 ← ...;
c2 ← ...;
(a2 < c2);
```

Значения a_3, b_3, c_4, d_3 и i_2 уже определены и (по определению *SSA*) больше определяться не будут, поэтому они удаляются из стеков

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

B_7 после чистки

	a	b	c	d	i
Счетчики	4	4	5	4	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

B_7:

```
a3 ← φ(a2, a);
b3 ← φ(b2, b);
c4 ← φ(c3, c);
d3 ← φ(d2, d);
x  ← a3 + b3;
z  ← c4 + d3;
i2 ← i1 + 1;
(i2 ≤ 100)…;
```

- ◊ Работа ***Rename*(B_7)**:
  - ◈ Возврат в ***Rename*(B_1);**
- ◊ Работа ***Rename*(B_1)**:
  - ◈ Заполнение параметров φ-функций в B_3 не производится, так как в B_3 нет φ-функций
  - ◈ Вызов ***Rename*(B_3);**

B_1:

```
a1 ← φ(a0, a3);
b1 ← φ(b0, b3);
c1 ← φ(c0, c4);
d1 ← φ(d0, d3);
i1 ← φ(i0, i2);
a2 ← ...;
c2 ← ...;
(a2 < c2);
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_3

Счетчики

Стеки (↓)

a	b	c	d	i
4	4	5	4	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2		

B_3:
```
a ← ...;
d ← ...;
(a ≤ d)...;
```

↓ ***Rename*(B_3)**

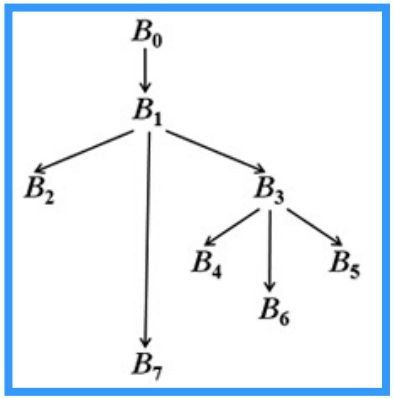
Вход в B_4

Счетчики

Стеки (↓)

a	b	c	d	i
4	4	5	4	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_4	
a_4				

B_3:
```
a4 ← ...;
d4 ← ...;
(a4 ≤ d4)...;
```



Работа ***Rename*(B_3)**:

- Вызов *NewName* **(a)** → a_4;
- Вызов *NewName* **(d)** → d_4;
- Переименование операндов;
- **Чистка стеков**
- Вызов ***Rename*(B_4)**;

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_4

	a	b	c	d	i
Счетчики	5	4	5	5	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	

B_4: `d ← ...;`

↓ ***Rename*(B_4)**

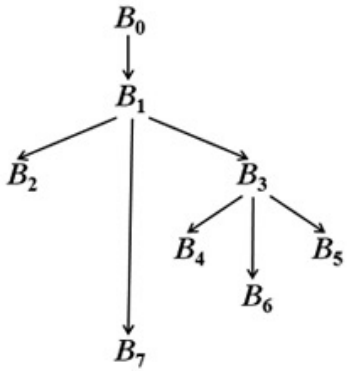
B_4: `d5 ← ...;`

Работа ***Rename*(B_4)**:

- Вызов *NewName* **(d)** → d_5 ;
- Переименование операндов ;
- Заполнение параметров φ-функций в B_6
- **Чистка стеков**
- Вызов ***Rename*(B_6)** ;

B_4 до чистки

	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
				d_5	



B_6:
```
c5 ← φ(c2, c);
d6 ← φ(d4, d5);
b4 ← ...;
```

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_6

	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

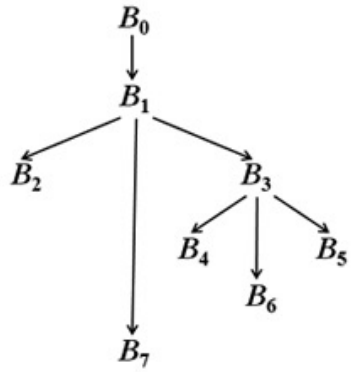
B_6:
```
c ← φ(c, c);
d ← φ(d, d);
b ← ...;
```

↓ ***Rename*(B_6)**

B_6:
```
c5 ← φ(c2, c);
d6 ← φ(d4, d5);
b4 ← ...;
```

B_6 до чистки

	a	b	c	d	i
Счетчики	5	5	6	7	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_4	c_2	d_4	
	a_4		c_5	d_6	



Работа ***Rename*(B_6)**:

- Вызов *NewName* (**c**) → c_5;
- Вызов *NewName* (**d**) → d_6;
- Вызов *NewName* (**b**) → b_4;
- Переименование операндов;
- Заполнение параметров φ-функций в B_7
- **Чистка стеков**
- Возврат в ***Rename*(B_3)**;

6.3 Построение частично усеченной SSA-формы

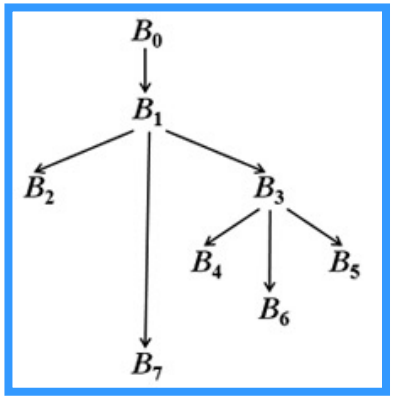
6.3.4. Переименование переменных

Вход в B_5

	a	b	c	d	i
Счетчики	5	4	5	5	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

B_5 до чистки

	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_5	
	a_4		c_5		



B_5: `c ← ...;`

↓ ***Rename*(B_5)**

B_5: `c5 ← ...;`

Работа ***Rename*(B_5)**:

- Вызов *NewName* **(d)** → d_5;
- Переименование операндов;
- Заполнение параметров φ-функций в B_6
- **Чистка стеков**
- Возврат в ***Rename*(B_3)**;

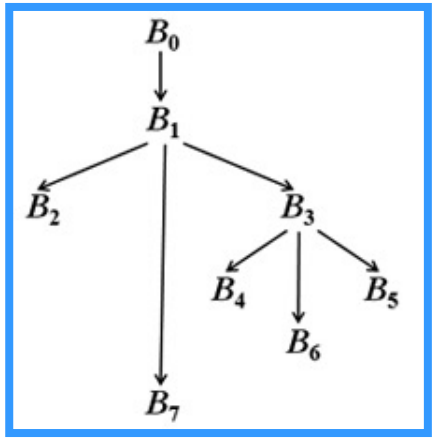
B_6:
```
c6 ← φ(c2, c5);
d6 ← φ(d4, d5);
b4 ← ...;
```

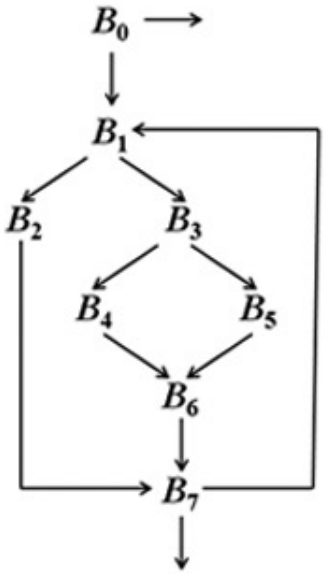
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Вход в B_3

	a	b	c	d	i
Счетчики	5	5	6	7	3
Стеки (↓)	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_4	c_2	d_4	
	a_4		c_5	d_6	





Работа ***Rename***(B_3):

- **Чистка стеков**
- Возврат в ***Rename***(B_1);

Работа ***Rename***(B_1):

- Возврат в ***Rename***(B_0);

6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Блок B_6 (последний)

$Rename(B_6)$ по рекурсии из блока B_1.

1) заменяет имена, определяемые φ-функциями, создавая новые имена a_4, b_4, c_6 и d_6.
2) заменяет использования глобальных имен в двух присваиваниях, не меняя определяемых ими имен, так как ни y, ни z не являются глобальными.
3) последнем присваивании заменяет используемое имя на i_1, а затем создает новое имя i_2 для определения переменной i.
4) заменяет второй параметр каждой φ-функции в B_1 (единственный последователь по ГПУ) на соответствующее текущее имя (a_4, b_4, c_6, d_6 и i_2).

Потом $Rename$ освобождает стеки, возвращается в B_1, в B_0 и прекращает работу.

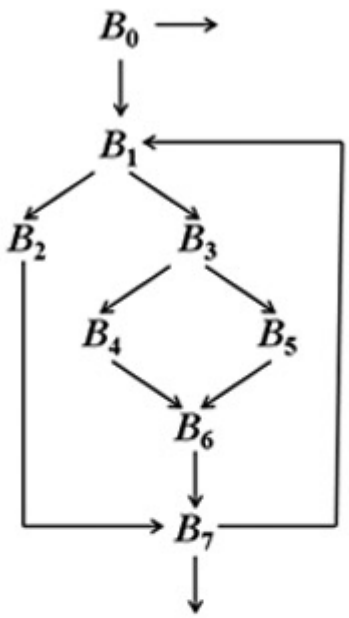
6.3 Построение частично усеченной SSA-формы

6.3.4. Переименование переменных

Блок B_0

Когда в конце обхода дерева доминаторов управление снова попадает в B_0 $Rename$ выталкивает значение из стека для i и происходит выход из $Rename$.

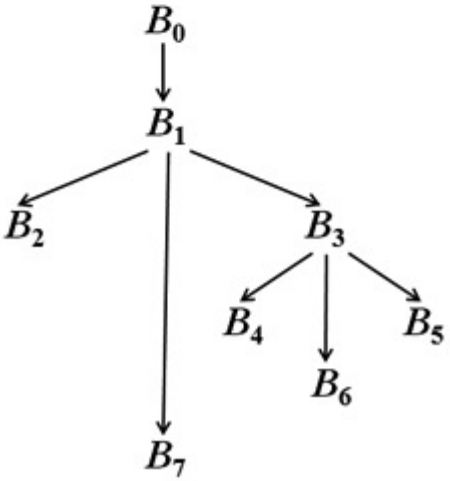
Граф потока управления



Состояние счетчиков и стеков в момент, когда $Rename(B_0)$ уже кончила обрабатывать блок B_0, но еще не удалила имена, связанные с B_0 из соответствующих стеков

Глобальные переменные	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки	a_0	b_0	c_0	d_0	i_0

Дерево доминаторов

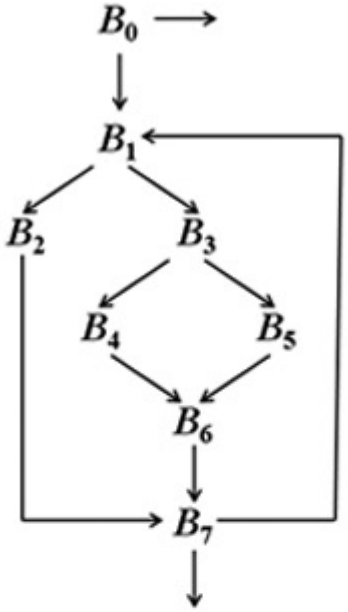


6.3 Построение частично усеченной SSA-формы
6.3.4. Переименование переменных

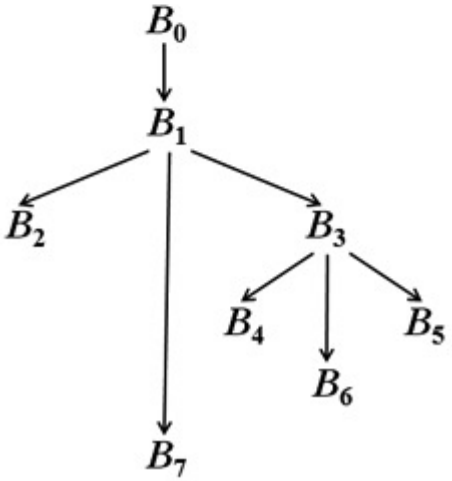
После окончания работы алгоритма получим

B_0	$i_0 \leftarrow 1$		
B_1	$a_1 \leftarrow \varphi(a_0, a_4)$ $b_1 \leftarrow \varphi(b_0, b_4)$ $c_1 \leftarrow \varphi(c_0, c_6)$ $d_1 \leftarrow \varphi(d_0, d_6)$ $i_1 \leftarrow \varphi(i_0, i_2)$ $a_2 \leftarrow$ $b_2 \leftarrow$	B_7	$a_4 \leftarrow \varphi(a_2, a_3)$ $b_4 \leftarrow \varphi(b_2, b_3)$ $c_6 \leftarrow \varphi(c_3, c_5)$ $d_6 \leftarrow \varphi(d_2, d_5)$ $y \leftarrow +, a_4, b_4$ $z \leftarrow +, c_6, d_6$ $i_2 \leftarrow +, i_1, 1$
B_2	$b_2 \leftarrow$ $c_3 \leftarrow$ $d_2 \leftarrow$	B_3	$a_3 \leftarrow$ $d_3 \leftarrow$
B_4	$d_4 \leftarrow$	B_6	$c_5 \leftarrow \varphi(c_2, c_4)$ $d_5 \leftarrow \varphi(d_4, d_3)$ $b_3 \leftarrow$
B_5	$c_4 \leftarrow$		

Граф потока управления



Дерево доминаторов



6.4 Восстановление кода из SSA-формы

6.4.1. Постановка задачи

- ◊ Поскольку процессоры не выполняют φ-функций, компилятор должен перевести программу в SSA-форме в выполняемый код.
- ◊ Рассмотрение примеров наводит на мысль, что **компилятору достаточно попросту опустить индексы имен и удалить φ-функции**.
  Такой подход был бы правильным, если бы при построении SSA-формы использовался простейший алгоритм.
- ◊ Однако, если переименования выполняются рассмотренным алгоритмом, простой процесс переименования **может привести к некорректному коду**.

6.4 Восстановление кода из SSA-формы

6.4.1. Постановка задачи

◊ **Пример**. Внизу слева показан базовый блок из четырех команд и его оптимизация с помощью ЛНЗ над исходными именами.
Справа показан тот же самый пример с использованием SSA-имен.

Вследствие того, что исходное имя a имеет разные SSA-имена a_0 и a_1, удалось оптимизировать последнее присваивание.

Заметим однако, что если у имен опустить индексы, получится некорректный код: c будет присвоено значение 17.

Исходные имена		***SSA*-имена**	
a ← +, x, y	a ← +, x, y	a_0 ← +, x_0, y_0	a_0 ← +, x_0, y_0
b ← +, x, y	b ← a	b_0 ← +, x_0, y_0	b_0 ← a_0
a ← 17	a ← 17	a_0 ← 17	a_1 ← 17
c ← +, x, y	c ← +, x, y	c_0 ← +, x_0, y_0	c_0 ← a_0

6.4 Восстановление кода из SSA-формы

6.4.2. Замена φ-функций группами инструкций копирования

◊ Можно оставить SSA-имена неизменными, заменив каждую φ-функцию группой команд копирования (по одной для каждого входного ребра), предоставив разбираться с именами оптимизирующему преобразованию «Распространение копий».

В рассматриваемом примере при удалении φ-функций будет:

```
B7  a4 ← φ(a2, a3)
    b4 ← φ(b2, b3)
    c6 ← φ(c3, c5)
    d6 ← φ(d2, d5)
    y  ← +, a4, b4
    z  ← +, c6, d6
    i2 ← +, i1, 1
```

```
B6  c5 ← φ(c2, c4)
    d5 ← φ(d4, d3)
    b3 ←
```

```
B2  b2 ←
    c3 ←
    d2 ←
```

```
B7  y  ← +, a4, b4
    z  ← +, c6, d6
    i2 ← +, i1, 1
```

```
B6  b3 ←
    a4 ← a2
    b4 ← b2
    c6 ← c3
    d6 ← d2
```

```
B2  b2 ←
    c3 ←
    d2 ←
    a4 ← a3
    b4 ← b3
    c6 ← c5
    d6 ← d5
```

6.4 Восстановление кода из SSA-формы

6.4.2. Замена φ-функций группами инструкций копирования

```
B7   a4 ← φ(a2, a3)
     b4 ← φ(b2, b3)
     c6 ← φ(c3, c5)
     d6 ← φ(d2, d5)
     y  ← +, a4, b4
     z  ← +, c6, d6
     i2 ← +, i1, 1
```

```
B6   c5 ← φ(c2, c4)
     d5 ← φ(d4, d3)
     b3 ←
```

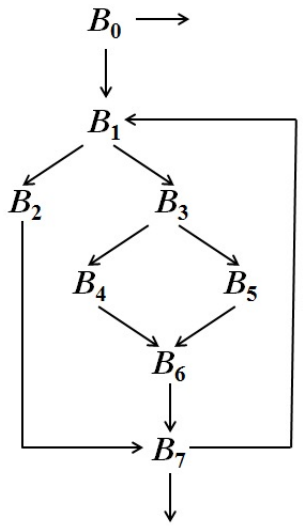
```
B2   b2 ←
     c3 ←
     d2 ←
```

```
B7   y  ← +, a4, b4
     z  ← +, c6, d6
     i2 ← +, i1, 1
```

```
B6   b3 ←
     a4 ← a2
     b4 ← b2
     c6 ← c3
     d6 ← d2
```

```
B2   b2 ←
     c3 ←
     d2 ←
     a4 ← a3
     b4 ← b3
     c6 ← c5
     d6 ← d5
```

- ◊ Удаление φ-функций из блока B_6 породит инструкции копирования в блоках B_4 и B_8.
- ◊ Удаление φ-функций из блока B_1 должно породить инструкции копирования в блоках B_0 и B_7. Но этого делать нельзя!



6.4 Восстановление кода из SSA-формы

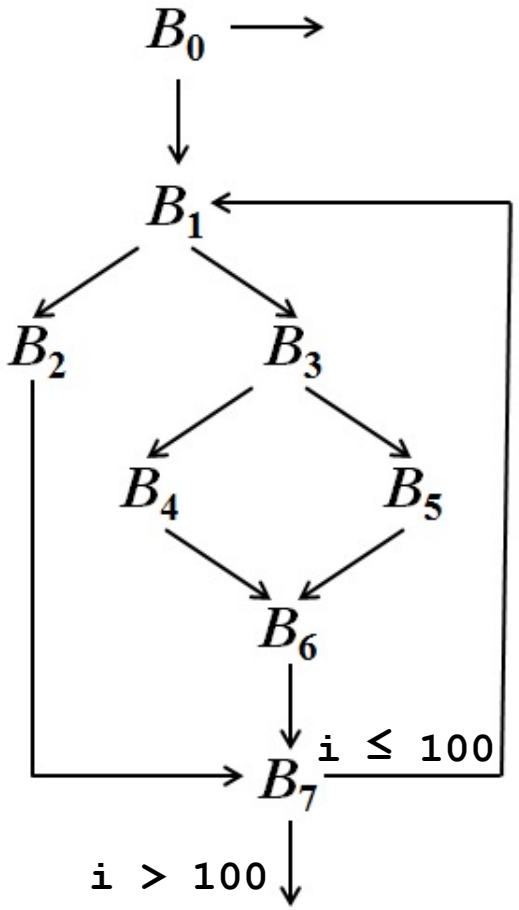
6.4.2. Замена φ-функций группами инструкций копирования

◊ Удаление φ-функций из блока B_1 должно породить инструкции копирования в блоках B_0 и B_7.

Но этого делать нельзя! Почему?

◊ У блоков B_0 и B_7 по два последующих блока (вторые блоки не попали на схему, так как они находятся вне анализируемого цикла).

Если вставить копирования в блоки B_0 и B_7, они будут выполняться не только на ребрах, внутри рассматриваемого цикла, но и на ребрах выводящих из цикла.



6.4 Восстановление кода из SSA-формы

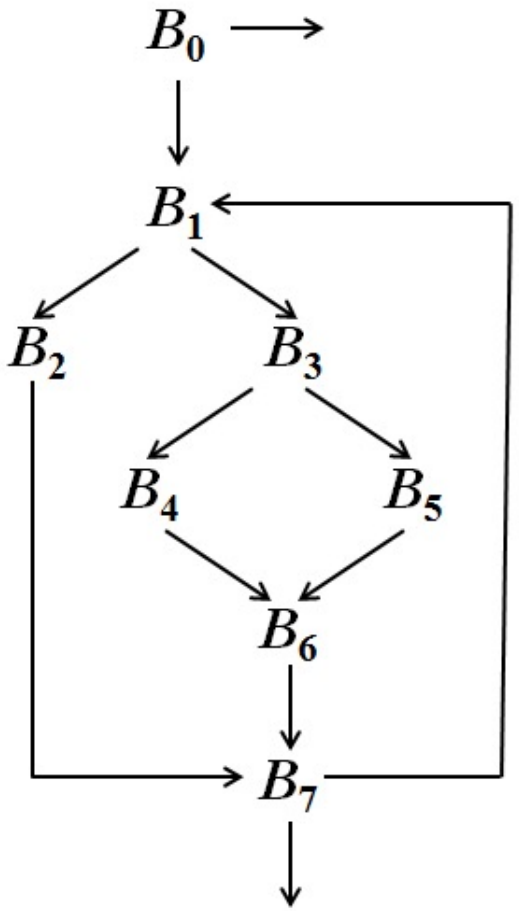
6.4.2. Замена φ-функций группами инструкций копирования

◊ По определению ребро, выходящее из вершины с несколькими потомками, и входящее в вершину с несколькими предками, называется *критическим*.

Ребра ($B_0\ B_1$) и ($B_7\ B_1$) – критические.

◊ Критические ребра обычно исключают, разрывая их и вставляя в разрыв дополнительные вершины.

◊ Разорвем критические ребра и добавим два новых блока B_8 и B_9, поместив в них требуемые инструкции копирования.



6.4 Восстановление кода из SSA-формы

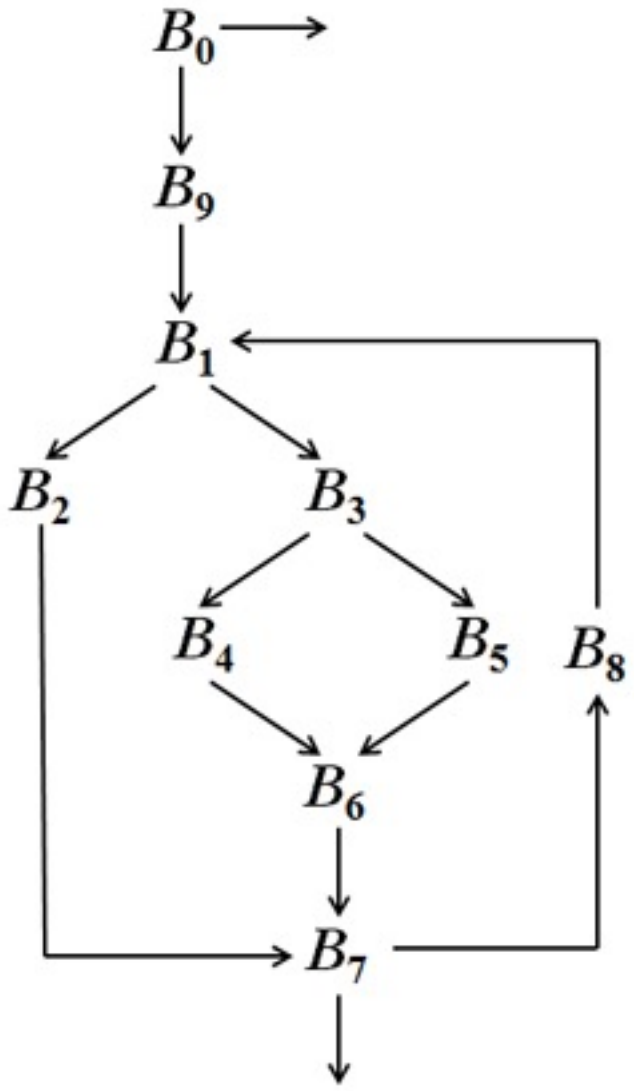
6.4.2. Замена φ-функций группами инструкций копирования

◊ По определению ребро, выходящее из вершины с несколькими потомками, и входящее в вершину с несколькими предками, называется *критическим*.

Ребра ($B_0\ B_1$) и ($B_7\ B_1$) – критические.

◊ Критические ребра обычно исключают, разрывая их а вставляя в разрыв дополнительные вершины.

◊ Разорвем критические ребра и добавим два новых блока B_8 и B_9, поместив в них требуемые инструкции копирования.



6.4 Восстановление кода из SSA-формы

6.4.2. Замена φ-функций группами инструкций копирования