16. Распределение и назначение регистров

2.1 Постановка задачи

2.1.1 Модель памяти

- ◊ В промежуточном представлении с каждой переменной связывается ячейка памяти для хранения ее значений.
- ◊ Эта ячейка называются *абстрактной*, так как с ней связывается *символический адрес*, который может указывать на регистры, стек, кучу, статическую память, причем каждому классу памяти соответствует бесконечно много символических адресов
- ◊ В компиляторах часто используется модель памяти «*регистр – регистр*». В этой модели компилятор старается расположить все данные на *символических регистрах* (или *псевдорегистрах*). В отличие от физических регистров, число которых невелико, псевдорегистров бесконечно много. При распределении памяти часть псевдорегистров придется отобразить не на регистры, а в память.

2.1 Постановка задачи

2.1.2 Распределение и назначение регистров

◊ *Распределение регистров*

отображает неограниченное множество имен (псевдорегистров) на конечное множество физических регистров целевой машины, в т.ч. Генерируются загрузки/сохранения из/в память.

Это NP-полная задача.

◊ *Назначение регистров*

отображает множество распределенных имен регистров на физические регистры целевого процессора. Для решения этой задачи известно несколько алгоритмов *полиномиальной сложности.*

◊ Во время назначения регистров предполагается, что распределение регистров уже было выполнено, так что при генерации каждой команды требуется не более n регистров (n – число физических регистров).

2.2 Локальное распределение регистров

2.2.1 Постановка задачи

- Применения регистров:
  - На регистры помещаются **операнды и результаты операций**
    (при выполнении операции необходимо, чтобы ее операнды находились на регистрах, результат получается на регистре).
  - Регистры – **временные переменные**
    (на регистры помещаются промежуточные результаты при вычислении выражений
    если удается, на них размещаются все переменные, использующиеся в пределах только одного базового блока).
  - Регистры используются **для хранения глобальных значений**.
  - Регистры используются для помощи в управлении памятью времени выполнения (например для управления стеком времени выполнения, включая поддержку указателя стека).

2.2 Локальное распределение регистров

2.2.1 Постановка задачи

- Рассмотрим алгоритм, распределяющий только те регистры, которые предназначены для операндов и временных переменных
(остальные регистры зарезервированы).

- Предположения:
  - Базовый блок уже оптимизирован (все «лишние» вычисления удалены).
  - Для каждой операции **OP** существует команда вида

    OP reg, reg, reg

    (операнды и результат – на регистрах)
  - В набор команд входят команды:

    LD reg, mem (загрузка из памяти на регистр)

    ST mem, reg (сохранение значения регистра)

- Необходимо, чтобы генератор кода минимизировал количество операций **LD** и **ST** в целевом коде

2.2 Локальное распределение регистров

2.2.2 Дескрипторы регистров и переменных

◊ *Дескриптор* $DR[r]$ *регистра* r указывает, значение какой переменной содержится на регистре r (на каждом регистре могут храниться значения одного или нескольких имен)

◊ *Дескриптор* $DA[a]$ *переменной* a указывает адрес текущего значения a. Это может быть регистр, адрес памяти, указатель стека

◊ Пусть определена *функция* $getReg\ (I)$, имеющая доступ ко всем дескрипторам регистров и адресов, а также к другим атрибутам объектов, хранящимся в таблице символов,
которая назначает регистры для операндов и результата команды I.

◊ Функция $getReg\ (I)$ позволяет назначать регистры во время выбора команд

2.2 Локальное распределение регистров

2.2.3 Выбор команд для базового блока

◊ Выбор команд для вычислительной трехадресной инструкции $x \leftarrow op, y, z$

1. С помощью функции $getReg()$ выбираются регистры R_x, R_y и R_z для x, y и z.
2. Если $DR[R_y] \neq y$ (y не находится на R_y), генерируется команда **LD R_y, y'**, где $y' = DA[y]$ (местоположение y в памяти).
3. Если $DR[R_z] \neq z$, а $DA[z] = z'$, генерируется команда **LD R_z, z'**.
4. Генерируется команда **OP R_x, R_y, R_z**.

2.2 Локальное распределение регистров

2.2.3 Выбор команд для базового блока

◊ Выбор команды для инструкции копирования $x = y$

Функция $getReg()$ всегда выбирает для x и y одни и те же регистры.

Если $DR[R_y] \neq y$, генерируется команда **LD R_y, y′**.

Если $DR[R_y] = y$, ничего не генерируется

Во всех случаях обновляется $DR[R_y]$: x становится одним из значений, находящихся на R_y.

◊ Генерация команды запоминания значений переменных, остающихся живыми после выхода из блока.

Если переменная x жива на выходе из блока,
и если в конце блока оказывается, что $DA[x] = R$ (а не x),
требуется генерация команды **ST x, R**.

2.2 Локальное распределение регистров

2.2.3 Выбор команд для базового блока

- Правила обновления DR и DA после генерации команды
  - Для команды **`LD R, x`**:
    - изменяем $DR[R]$: в R хранится только x;
    - изменяем $DA[x]$, добавляя ссылку на R
  - Для команды **`ST x, R`**
    - изменяем $DA[x]$, добавляя ссылку на x
  - Для команды: **`ADD` $\mathbf{R_x}$`,` $\mathbf{R_y}$`,` $\mathbf{R_z}$**:
    - изменяем $DR[R_x]$: в R_x хранится x;
    - изменяем $DA[x]$: x – только на R_x
    - удаляем R_x из DA всех переменных, кроме x.
  - Для команды $x = y$
    - если $DA[y] \neq R_y$ добавляем команду $LD\ R_y, y$;
    - изменяем $DA[x]$ так, чтобы он указывал только на R_y

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1  t ← -, a, b
2  u ← -, a, c
3  v ← +, t, u
4  a ← d
5  d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

Для инструкции 1: **t = a – b** необходимо сгенерировать три команды:

- загрузка регистра R_a
- загрузка регистра R_b
- вычитание (результат на регистре R_t)

R_1	R_2	R_3

a	b	c	d	t	u	v
a	b	c	d			

LD R1, a

$getReg()$ выдает R_1 для R_a

R_1	R_2	R_3
a		

a	b	c	d	t	u	v
a, R_1	b	c	d			

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1   t ← -, a, b
2   u ← -, a, c
3   v ← +, t, u
4   a ← d
5   d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

Для инструкции **1**: **t = a – b** необходимо сгенерировать три команды:

- загрузка регистра R_a
- загрузка регистра R_b
- вычитание (результат на регистре R_t)

R_1	R_2	R_3
a		

a	b	c	d	t	u	v
a	b	c	d			

```
LD   R1,   a
LD   R2,   b
```

getReg() выдает R_2 для R_b

R_1	R_2	R_3
a	b	

a	b	c	d	t	u	v
a, R_1	b, R_2	c	d			

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1   t ← -, a, b
2   u ← -, a, c
3   v ← +, t, u
4   a ← d
5   d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

Для инструкции **1**: **t = a – b** необходимо сгенерировать три команды:

- загрузка регистра R_a
- загрузка регистра R_b
- вычитание (результат на регистре R_t)

R_1	R_2	R_3
a	b	

a	b	c	d	t	u	v
a, R_1	b, R_2	c	d			

```
LD   R1,   a
LD   R2,   b
SUB  R2,   R1,   R2
```

$getReg()$ выдает R_2 для R_t

R_1	R_2	R_3
a	t	

a	b	c	d	t	u	v
a	b	c	d	R_2		

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1   t ← -, a, b
2   u ← -, a, c
3   v ← +, t, u
4   a ← d
5   d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

Для инструкции **2**: **u = a – c** необходимо сгенерировать две команды:

- загрузка регистра R_c
- вычитание (результат на регистр R_u)

R_1	R_2	R_3
a	t	

a	b	c	d	t	u	v
a, R_1	b	c	d	R_2		

```
LD   R3,   c
SUB  R1,   R1,  R3
```

getReg() выдает R_3 для R_c и R_1 для R_u

R_1	R_2	R_3
u	t	c

a	b	c	d	t	u	v
a	b	c, R_3	d	R_2	R_1	

u помещается на R_1, так как значение a, ранее располагавшееся на R_1, больше внутри блока не используется

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1   t ← -, a, b
2   u ← -, a, c
3   v ← +, t, u
4   a ← d
5   d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

Для инструкции копирования 4: **a = d** необходимо сгенерировать одну команду:

- загрузка регистра R_d

R_1	R_2	R_3
u	**t**	**v**

a	**b**	**c**	**d**	**t**	**u**	**v**
a	**b**	**c**	**d**	R_2	R_1	R_3

LD R2, d

getReg() выдает R_2 для R_d

R_1	R_2	R_3
u	**a, d**	**v**

a	**b**	**c**	**d**	**t**	**u**	**v**
a	**b**	**c**	**d**, R_2		R_1	R_3

В регистре R_2 теперь хранятся и d, и a.

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1   t ← -, a, b
2   u ← -, a, c
3   v ← +, t, u
4   a ← d
5   d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

Для инструкции 5: **d = v + u** необходимо сгенерировать одну команду:

- сложение, результат на регистр R_d

R_1	R_2	R_3
u	a, d	v

a	b	c	d	t	u	v
a	b	c	d, R_2		R_1	R_3

ADD R1, R3, R1

getReg() выдает R_1 для R_d

R_1	R_2	R_3
d	a	v

a	b	c	d	t	u	v
R_2	b	c	R_1			R_3

После этой команды

- **d** хранится только на R_1, но не в ячейке памяти для **d**.
- **a** хранится только на R_2, но не в ячейке памяти для **a**.

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1   t ← -, a, b
2   u ← -, a, c
3   v ← +, t, u
4   a ← d
5   d ← +, v, u
```

t, **u**, **v** – временные переменные, локальные для блока,

a, **b**, **c**, **d** – переменные, живые при выходе из блока.

В заключение необходимо сохранить живые переменные **a** и **d**, значения которых есть только на регистрах

R_1	R_2	R_3
d	a	v

a	b	c	d	t	u	v
R_2	b	c	R_1			R_3

```
ST   a,   R2
ST   d,   R1
```

R_1	R_2	R_3
d	a	v

a	b	c	d	t	u	v
a, R_2	b	c	d, R_1			R_3

2.2 Локальное распределение регистров

2.2.4 Пример

Генерация кода для базового блока:

```
1  t ← -, a, b
2  u ← -, a, c
3  v ← +, t, u
4  a ← d
5  d ← +, v, u
```

t, **u** и **v** – временные переменные, локальные для блока,
a, **b**, **c** и **d** – переменные, живые при выходе из блока.

Сгенерированный код:

```
LD    R1, a
LD    R2, b
SUB   R2, R1, R2
LD    R3, c
SUB   R1, R1, R3
ADD   R3, R2, R1
LD    R2, d
ADD   R1, R3, R1
ST    a, R2
ST    d, R1
```

Код содержит
4 команды **LD**
2 команды **ST**

Все эти команды связаны с множествами $In(B)$ и $Out(B)$ переменных живых при входе в блок B и при выходе из него

6 команд из 10 связаны с обращениями к памяти

2.2 Локальное распределение регистров

2.2.5 Реализация функции *getReg*

R_1	R_2	R_3
d	a	v

a	b	c	d	t	u	v
a, R_2	b	c	d, R_1			R_3
i	i	i	i			
o	o	o	o			
f						

◊ Генерация команды для инструкции I

x ← op, y, z

- ◈ выбор регистров для операндов **y** и **z**
- ◈ выбор регистра для результата **x**

◊ Выбор регистра R_y для операнда **y**

(регистр R_z для операнда **z** выбирается аналогично) .

- ◈ Если $DA[y]$ ссылается на регистр R, то полагаем $R_y = R$
- ◈ Если $DA[y]$ не содержит ссылок на регистры, но имеется регистр R, для которого $D[R]$ не содержит ссылок ни на одну переменную, то полагаем $R_y = R$

2.2 Локальное распределение регистров

2.2.5 Реализация функции *getReg*

◊ Выбор регистра R_y для операнда **y**

- ◈ Если $DA[y]$ не содержит ссылок на регистры и не имеется ни одного регистра R, для которого $DR[R]$ не содержит ссылок ни на одну переменную, то R можно использовать в качестве R_y , если для каждой переменной v, ссылка на которую содержится $DR[R]$, выполняется одно из следующих условий:
  - ♦ $DA[v]$ содержит ссылку не только на R, но и на адрес v,
  - ♦ v представляет собой переменную x, вычисляемую командой I, и x не является одновременно одним из операндов команды I,
  - ♦ переменная v после команды I больше не используется.
- ◈ Если ни одна из перечисленных выше ситуаций не имеет места, то прежде чем использовать R в качестве R_y, необходимо выполнить *сброс регистра*, т.е. команду **ST v, R**

2.2 Локальное распределение регистров

2.2.5 Реализация функции *getReg*

◊ Выбор регистра R_x для результата **x**

- ◈ Если $DR[R]$ ссылается только на x, то полагаем $R_x = R$
  Это можно делать даже тогда, когда x является одним из y или z, так как в одной машинной команде допускается совпадение двух регистров.
- ◈ Если y не используется после команды I и если $DR[R_y]$ ссылается только на y, то R_y может использоваться в роли R_x.
- ◈ Если z не используется после команды I и если $DR[R_z]$ ссылается только на z, то R_z может использоваться в роли R_x.

◊ Генерация команд для инструкции I

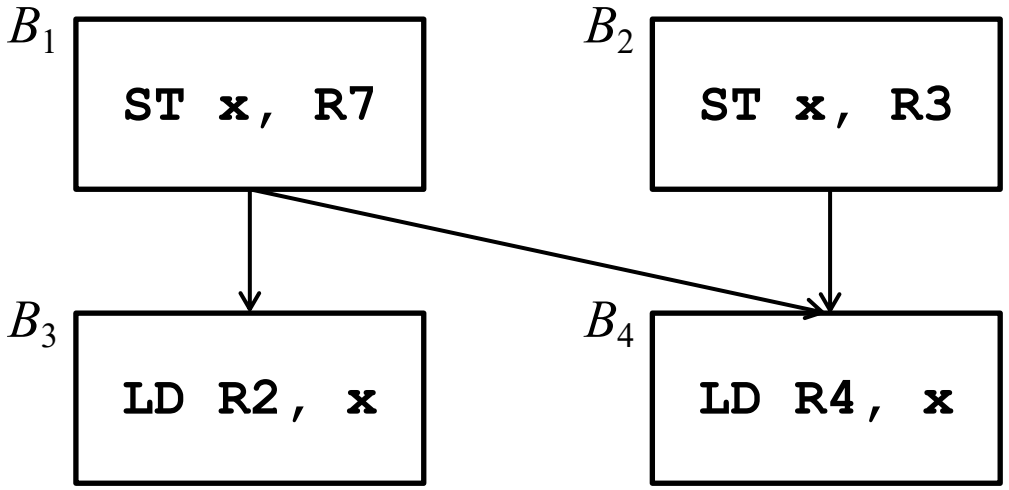
x ← y

Сначала выбирается R_y, как и для операнда инструкции **x ← op, y, z**, после чего полагается $R_x = R_y$.

2.2 Локальное распределение регистров

2.2.6 Ограничения

- В примере на рисунке независимое назначение регистров в базовых блоках привело к тому, что для одной и той же переменной **x** в каждом блоке используются разные регистры
- Если бы $getReg$ блока B_3 знала, что в блоке B_1 значение **x** было получено на **R7**, то выделила бы для **x** регистр **R7**, что позволило бы исключить команду загрузки на регистр в блоке B_3
- Наличие блоков B_2 и B_4 еще больше усложняет проблему, так как возникают различные требования на разных путях



2.3 Глобальное распределение и назначение регистров

2.3.1 Интервалы жизни

◊ *Интервалом жизни* (ИЖ) значения w переменной v называется множество команд программы, начиная с команды, в которой переменная v определяется со значением w, и кончая последней командой, в которой переменная v используется ***с этим значением***.

```
 0    LD    R0    ...              |   R0   [0,10]
 1    LD    R1    R0(0)            |   R1   [1,6]
 2    LD    R2    2                |   R1   [6,7]
 3    LD    R3    R0(@x)           |   R1   [7,8]
 4    LD    R4    R0(@y)           |   R1   [8,9]
 5    LD    R5    R0(@z)           |   R1   [9,10]
 6    MUL   R1    R1     R2        |   R2   [2,6]
 7    MUL   R1    R1     R3        |   R3   [3,7]
 8    MUL   R1    R1     R4        |   R4   [4,8]
 9    MUL   R1    R1     R5        |   R5   [5,9]
10    ST    v     R0(0)            |
```

2.3 Глобальное распределение и назначение регистров

2.3.2 Построение интервалов жизни

◊ **Построение множеств переменных, живых на выходе из каждого блока.** Это задача анализа потока данных. Методом итераций решается система уравнений

$$LiveOut[B] = \bigcup_{s \in Succ(B)} (use_S \cup (LiveOut[S] - VarKill_S)) \quad (1)$$

где $LiveOut(B)$ – множество переменных, живых на выходе из B,
$LiveIn(B)$ – множество переменных, живых на входе в B,
$use(B)$ – множество переменных блока B, которые используются в B до их переопределения в B,
$VarKill_B$ – множество переменных блока B, которые переопределяются в B (оно обозначалось как def_B).

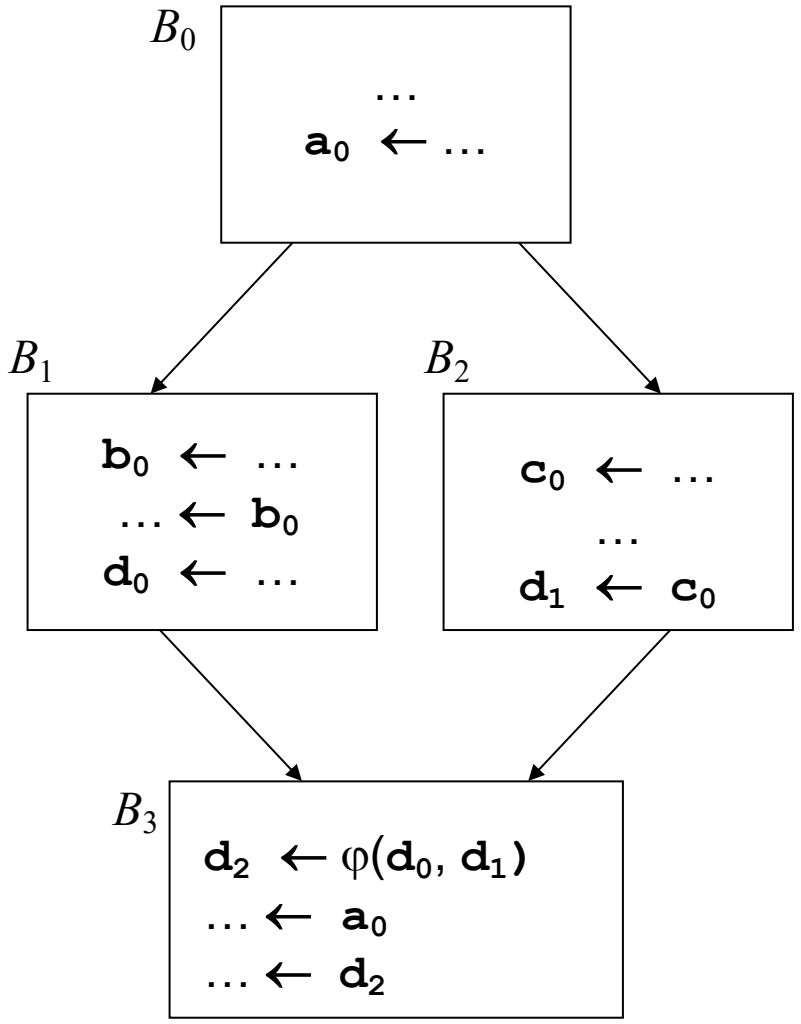
◊ Решив методом итераций систему (1),
получим $LiveOut(B)$ для всех B.

2.3 Глобальное распределение и назначение регистров
2.3.2 Построение интервалов жизни

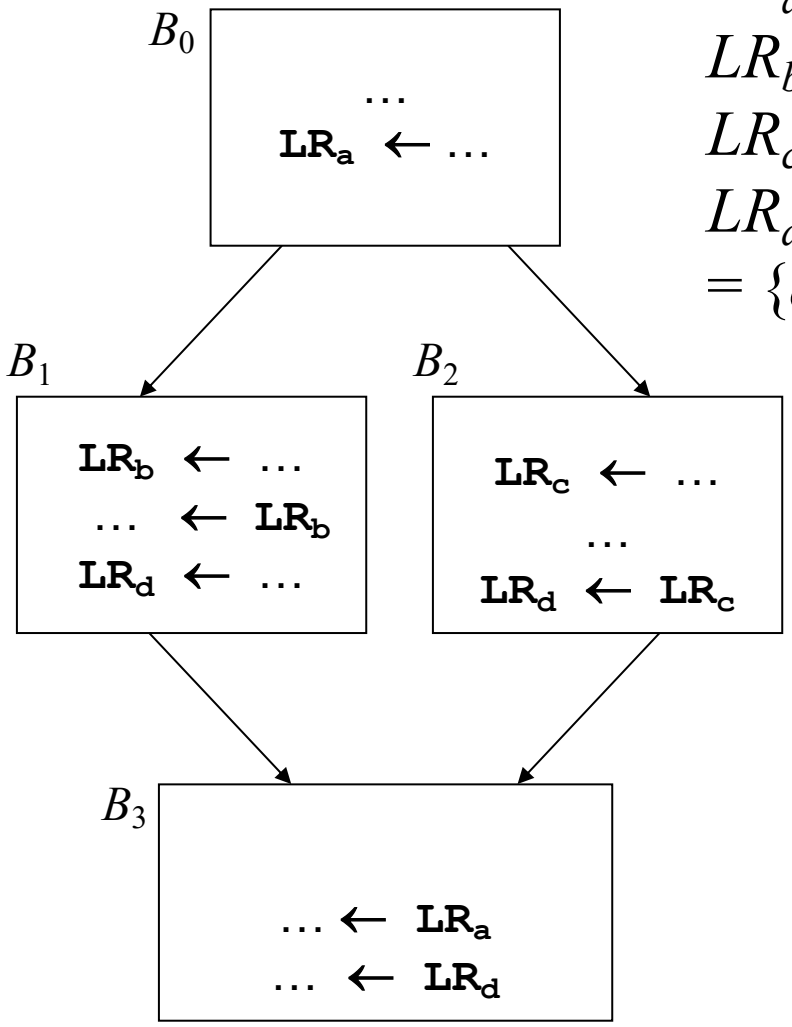
- ◊ **Исследование отношений между различными определениями и использованиями**. Нужно построить множества всех определений, достигающих одного и того же использования (*UD-цепочки*), и всех использований, которые достигаются одним и тем же определением (*DU-цепочки*).
- ◊ Такое построение удобно проводить в *SSA*-форме, так как в ней каждое имя определяется только один раз,
  каждое использование ссылается на единственное имя,
  а объединение имен обеспечивается с помощью φ-функций.
  Для программы в *SSA*-форме требуемая группировка имен достигается за один просмотр.
- ◊ ***Алгоритм объединения имен*** анализирует каждую φ-функцию и строит объединение множеств, связанных с каждым ее параметром, и множества, связанного с определяемой ею переменной. ***Это объединение и представляет ИЖ***.
- ◊ **После обработки всех φ-функций строится отображение *SSA*-имен на имена интервалов жизни.**

2.3 Глобальное распределение и назначение регистров

2.3.3 Построение интервалов жизни. Пример



Фрагмент кода
в *SSA*-представлении



Тот же фрагмент, переписанный в
терминах интервалов жизни

$$LR_a = \{a_0\}$$
$$LR_b = \{b_0\}$$
$$LR_c = \{c_0\}$$
$$LR_d = $$
$$= \{d_0, d_1, d_2\}$$

2.3 Глобальное распределение и назначение регистров

2.3.4 Оценка стоимости сброса

- Стоимость сброса складывается из следующих трех компонент:
  - стоимость вычисления адресов при сбросе
  - стоимость операций доступа к памяти
  - оценка частоты выполнения
- Для хранения сброшенных значений во фрейме процедуры выделяется специальная область.
  Это позволяет свести к минимуму стоимость вычисления адресов при сбросе, исключив использование косвенной адресации и дополнительных регистров для вычисления адреса сбрасываемого значения.
- Интервал жизни, который содержит только команды загрузки регистра и его сохранения может иметь
  отрицательную стоимость сброса

  ```
  LD R1, [mem]; ST [mem], R1; use R1
  ```

  – в случае, когда обе команды обращаются к одному и тому же адресу памяти. Такие ситуации могут возникнуть вследствие ранее выполненных оптимизаций (например, исключения избыточных вычислений).

2.3 Глобальное распределение и назначение регистров

2.3.4 Оценка стоимости сброса

- Чтобы учитывать частоту выполнения базовых блоков в графе потока управления, каждый базовый блок снабжается ***аннотацией***, содержащей оценку стоимости его выполнения (профилирование).
- Для получения грубой оценки стоимости частоты выполнения используется ***простая эвристика***:
  делается допущение, что каждый цикл выполняется 10 раз; тогда стоимость каждой загрузки внутри одного цикла оценивается как 10, внутри гнезда из двух циклов – как 100 и т.д.; стоимость каждой ветви непредсказуемого **`if-then-else`** оценивается как 0.5.

  На практике из этой эвристики, в частности, следует, что ***сброс выгоднее выполнять в более внешнем цикле***.
- Аннотации могут вычисляться заранее (и тогда потребуется дополнительный просмотр программы), либо во время первого обращения.

2.3 Глобальное распределение и назначение регистров

2.3.5 Конфликтные ситуации и граф конфликтов

◊ При распределении регистров моделируется состязание за место на регистрах целевой машины.
Рассмотрим два различных интервала жизни LR_i и LR_j. Если в программе существуют команды, во время которых и LR_i, и LR_j актуальны (т.е. «живы» одновременно), то они не могут занимать один и тот же регистр.
В таком случае говорят, что LR_i и LR_j находятся в конфликте.

◊ **Определение**. Интервалы жизни LR_i и LR_j *находятся в конфликте* если один из них актуален при определении другого и они имеют различные значения.

◊ Граф, узлы которого соответствуют отдельным интервалам жизни, а дуги соединяют интервалы жизни, находящиеся в конфликте, называется *графом конфликтов* (ГК). Этот граф не является направленным, так как отношение нахождения в конфликте симметрично.

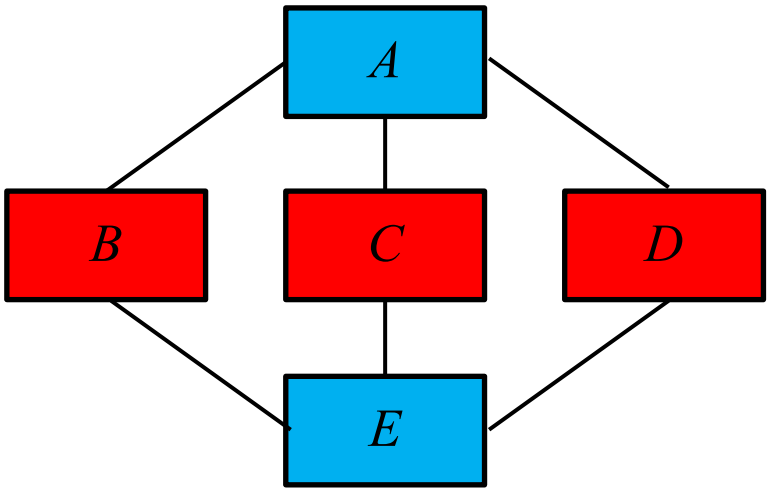
◊ Таким образом, если два узла ГК являются смежными (соединены дугой), то им должны соответствовать различные регистры.

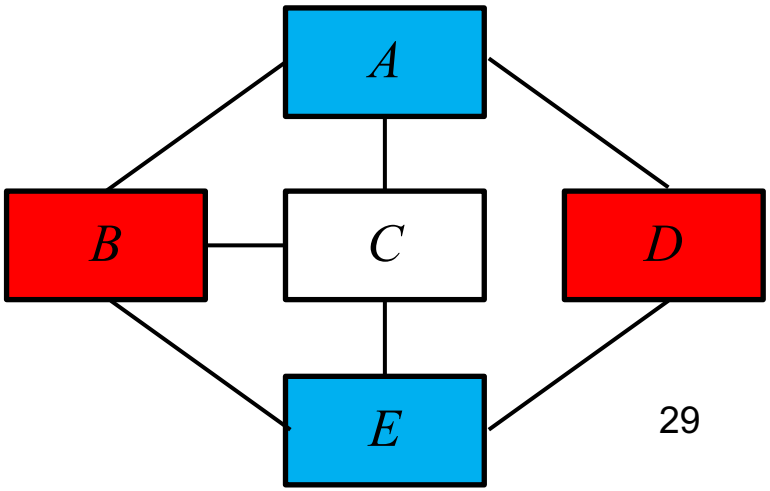
2.3 Глобальное распределение и назначение регистров

2.3.6 Раскраска графа

- ◊ Раскраска произвольного графа G состоит в присвоении каждому узлу G определенного цвета таким образом, чтобы любым двум смежным узлам G не были сопоставлены одинаковые цвета.
- ◊ Раскраска, использующая n цветов называется n-раскраской, а наименьшее из таких n называется *хроматическим числом* графа.

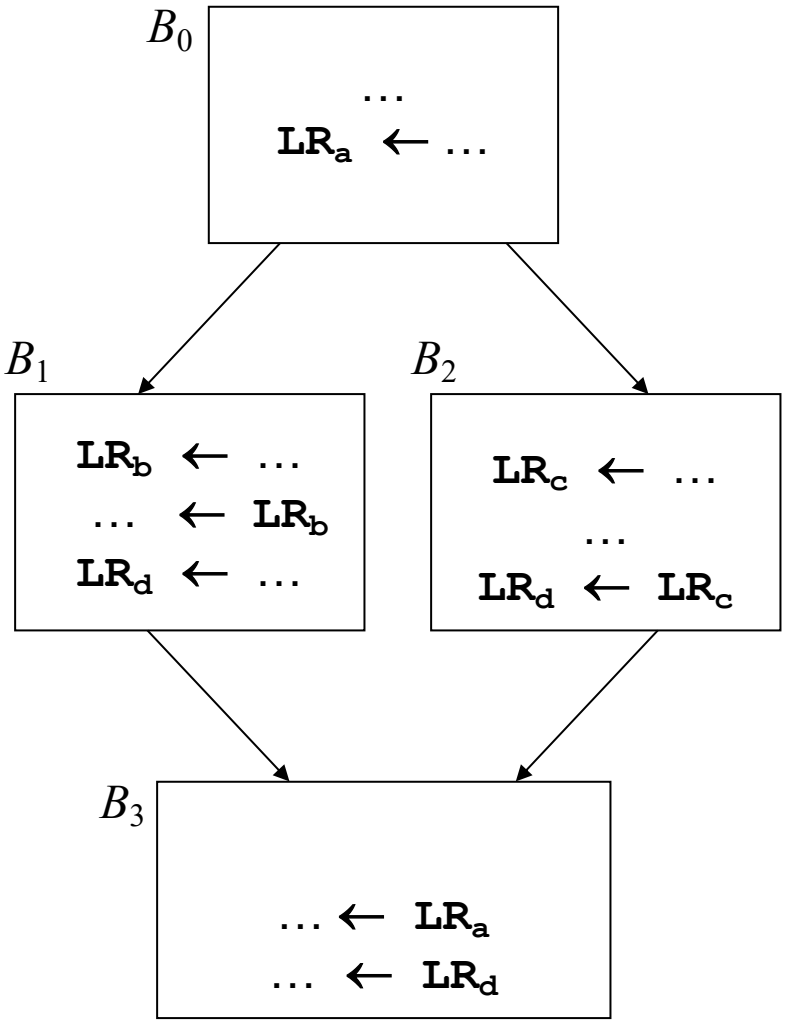
  На рисунке внизу хроматическое число левого графа равно 2, а правого графа – 3.
- ◊ Проблема нахождения хроматического числа графа и проблема выяснения, допускает ли граф n-раскраску NP-полны.





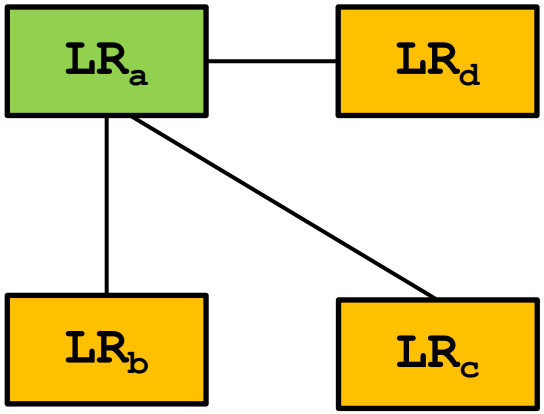
2.3 Глобальное распределение и назначение регистров

2.3.7 Раскраска графа конфликтов



Фрагмент кода с именами интервалов жизни

- Если у целевого компьютера n регистров, то проблема их распределения для программы сводится к проблеме построения n-раскраски ГК
- В частности, для данного примера ГК допускает 2-раскраску, следовательно, достаточно 2 регистров

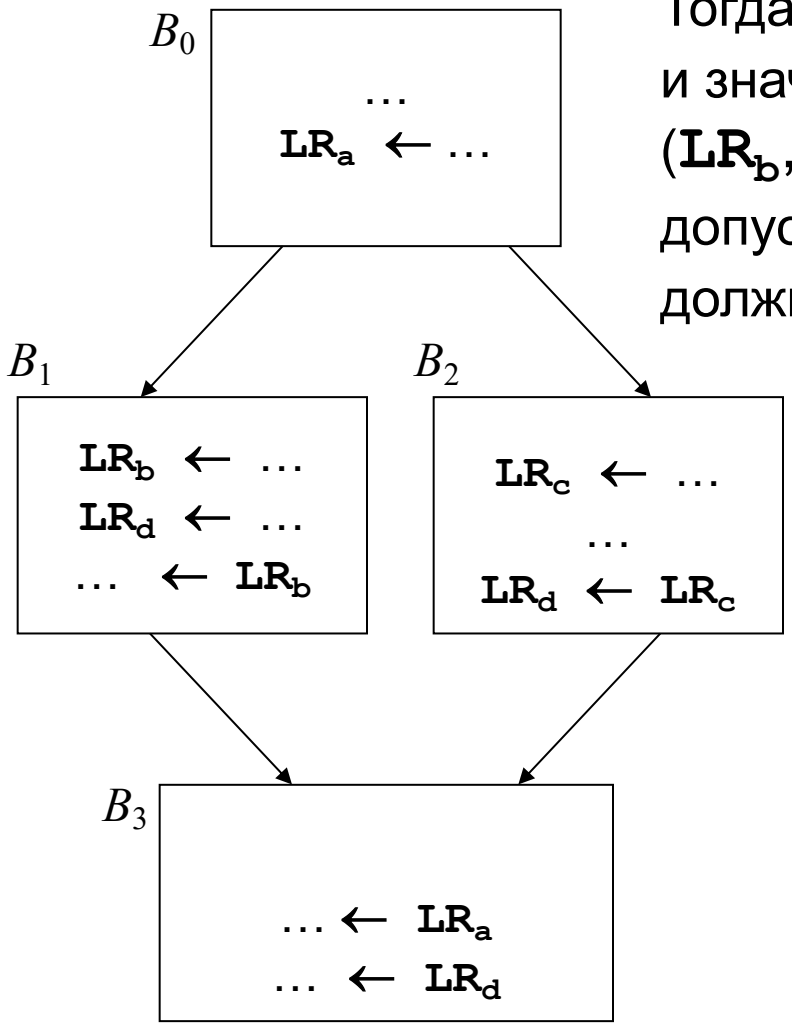


Граф конфликтов

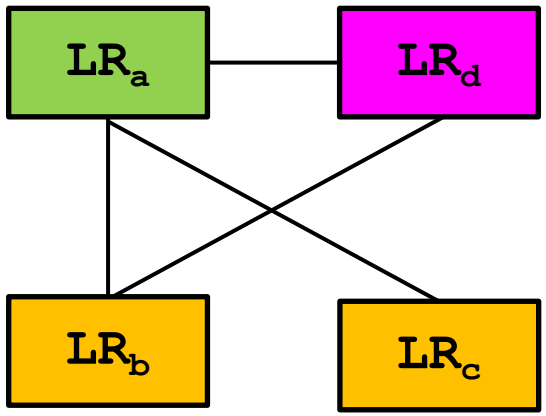
2.3 Глобальное распределение и назначение регистров

2.3.7 Раскраска графа конфликтов

Если оптимизирующая фаза компилятора переставила две последние команды блока B_1
Тогда $\mathbf{LR_b}$ будет живым при определении $\mathbf{LR_d}$, и значит в ГК необходимо добавить дугу ($\mathbf{LR_b}$, $\mathbf{LR_d}$) в результате чего ГК уже не будет допускать 2-раскраску, так как ($\mathbf{LR_a}$,$\mathbf{LR_b}$ и $\mathbf{LR_d}$ должны быть разного цвета)



Фрагмент кода с именами интервалов жизни



Граф конфликтов

Теперь у распределителя регистров две возможности:
1: Использовать третий регистр
2: Перед определением регистра для $\mathbf{LR_d}$ сбросить $\mathbf{LR_a}$ или $\mathbf{LR_b}$

2.3 Глобальное распределение и назначение регистров

2.3.8 Построение графа конфликтов (ГК)

- ◊ После построения глобальных интервалов жизни и множеств $LiveOut$ для каждого базового блока можно за один просмотр программы от $Exit$ к $Entry$ построить ГК.
- ◊ Внутри каждого базового блока алгоритм поддерживает множество $LiveNow$ переменных, живых в текущей точке блока. На выходе блока B полагают $LiveNow = LiveOut(B)$. Затем просматривают каждую команду ($op\ LR_a, LR_b, LR_c$) в направлении от конца блока к его началу, и узел ГК, соответствующий переменной LR_a, определяемой в этой команде, соединяют дугами со всеми узлами, соответствующими переменным из $LiveNow$ (это как раз переменные, живые во время определения другой переменной).
- ◊ Для повышения эффективности ГК задается левой (нижней) половиной своей матрицы смежности.
- ◊ Операции копирования $\mathbf{LR_i} \leftarrow \mathbf{LR_j}$ не порождают конкуренции между LR_i и LR_j, так как эти значения **могут** занимать один и тот же регистр.

2.3 Глобальное распределение и назначение регистров

2.3.8 Построение графа конфликтов

◊ Более точно алгоритм можно выразить следующим образом:

```
for each LR_i do
      create a node n_i ∈ N;
for each basic block b do{
      LiveNow = LiveOut(b)
      for I_n, I_n-1, …, I_1 in b do {
            // I_k имеет вид op_k LR_a, LR_b, LR_c
            for each LR_i ∈ LiveNow {
                  add (LR_a, LR_i) to E
                  remove LR_a from LiveNow
                  add LR_b to LiveNow
                  add LR_c to LiveNow
            }
      }
}
```

2.3 Глобальное распределение и назначение регистров

2.3.9 Слияние интервалов жизни

◊ Рассмотрим команду копирования $LR_i = LR_j$. Если ИЖ LR_i и LR_j не находятся в конфликте (не являются смежными узлами ГК), то команду копирования можно исключить и все ссылки на LR_i заменить ссылками на LR_j.

В результате ИЖ LR_i и LR_j как бы *сольются*.

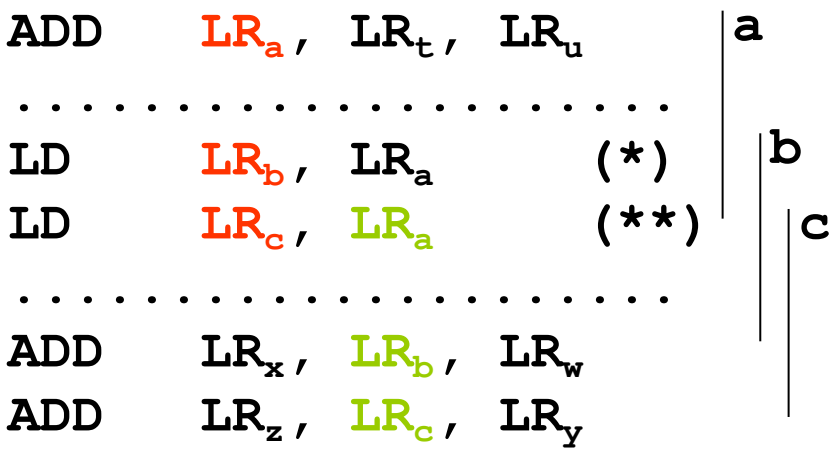
◊ *Слияние (coalescing – не путать со сливом/сбросом – spill!)* ИЖ приносит следующие выгоды:

- Исключается команда копирования (код становится меньше и тем самым потенциально быстрее)
- Снижается степень каждого узла (ИЖ), который был в конфликте либо с LR_i, либо с LR_j.
- Множество ИЖ сокращается (в литературе приводится пример, когда в результате слияния ИЖ удалось исключить свыше 30% всех ИЖ).

2.3 Глобальное распределение и назначение регистров

2.3.9 Слияние интервалов жизни

◊ **Пример.** Рассмотрим фрагмент программы:



Отрезки справа от кода отмечают ИЖ $\mathbf{LR_a}$ $\mathbf{LR_b}$ $\mathbf{LR_c}$

(красным показано определение переменной, зеленым – ее последнее использование).

Несмотря на то, что ИЖ $\mathbf{LR_a}$ пересекается и с $\mathbf{LR_b}$, и с $\mathbf{LR_c}$, он не находится в конфликте ни с тем, ни с другим, так как источник и приёмник в командах копирования (* $\mathbf{LR_a}$ и $\mathbf{LR_b}$) и (** $\mathbf{LR_c}$ и $\mathbf{LR_a}$) не находятся в конфликте.

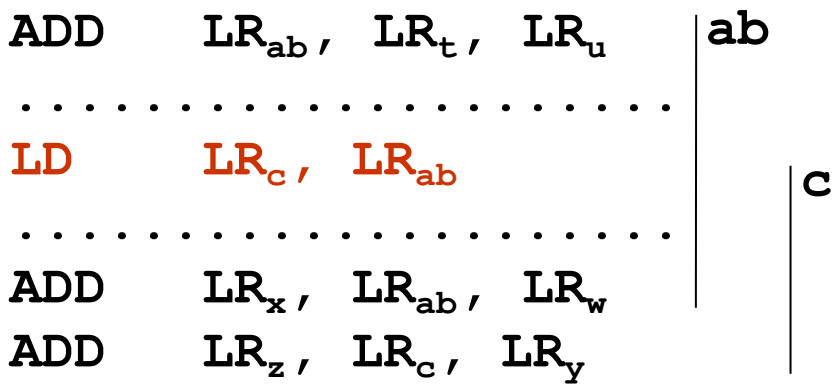
$\mathbf{LR_b}$ и $\mathbf{LR_c}$ находятся в конфликте, так как $\mathbf{LR_b}$ жив при определении $\mathbf{LR_c}$

ИЖ из обеих операций копирования являются кандидатами на слияние.

2.3 Глобальное распределение и назначение регистров

2.3.9 Слияние интервалов жизни

◊ После слияния LR_a и LR_b:



◊ После слияния ИЖ LR_a и LR_b получаем новый ИЖ LR_{ab}.

◊ Теперь ИЖ LR_c получается из LR_{ab} командой копирования, и следовательно LR_c и LR_{ab} не находятся в конфликте, так что слияние LR_a и LR_b в LR_{ab} понизило степень LR_c.
Слияние ИЖ может или уменьшить степени смежных узлов (ИЖ), или оставить их без изменения.

◊ Команда копирования **LD** LR_c, LR_{ab} позволяет продолжить процесс слияния ИЖ, заменив LR_{ab} на LR_{abc}.

2.3 Глобальное распределение и назначение регистров

2.3.10 Эвристики раскраски графа конфликтов

- После того как ГК построен, необходимо решить две задачи:
  - Для построенного ГК необходимо найти n-раскраску (n – число регистров целевой машины)
  - Необходимо разработать алгоритм обработки ситуации, когда при необходимости раскраски очередного интервала жизни (узла ГК) выясняется, что все n цветов исчерпаны
- Поскольку проблема n-раскраски графа NP-полна, применяются быстрые эвристические алгоритмы. При этом нет гарантии, что n-раскраска будет построена
- При исчерпании регистров применяются либо *слив (spill)*, либо *расщепление (split)* ИЖ (узлов ГК).
  В обоих случаях исходный ГК преобразуется к новому ГК, который может допускать n-раскраску.

2.3 Глобальное распределение и назначение регистров

2.3.10 Алгоритм раскраски графа конфликтов

- ***1 фаза. Установление порядка рассмотрения узлов***

  узлы по очереди удаляются из ГК и помещаются в стек.
  - Узел ГК называется неограниченным, если его степень $< n$, и ограниченным, если его степень $\geq n$.
  - Сначала в произвольном порядке удаляются неограниченные узлы вместе с дугами, соединяющими их со смежными узлами, при этом степень части смежных узлов понижается, так что некоторые из ограниченных узлов после удаления могут стать неограниченными.
  - Если после удаления всех неограниченных узлов в ГК все еще остаются узлы, то все они ограничены. Для каждого из ограниченных узлов вычисляется их степень (количество смежных узлов).
  - Ограниченные узлы удаляются из графа и помещаются в стек в порядке возрастания степени.

  В конце фазы граф конфликтов пуст, а все его узлы (ИЖ) находятся в стеке в некотором порядке.

2.3 Глобальное распределение и назначение регистров

2.3.10 Алгоритм раскраски графа конфликтов

◊ ***2 фаза. Раскраска узлов***

распределитель восстанавливает ГК, выбирая из стека очередной узел l и раскрашивая его в цвет, отличный от цвета смежных узлов. Если оказывается, что все цвета использованы, узел l остается нераскрашенным.

В конце фазы стек пуст, а ГК восстановлен и часть его узлов раскрашена.

2.3 Глобальное распределение и назначение регистров

2.3.10 Алгоритм раскраски графа конфликтов

◊ ***3 фаза. Проверка на окончание процесса раскраски***

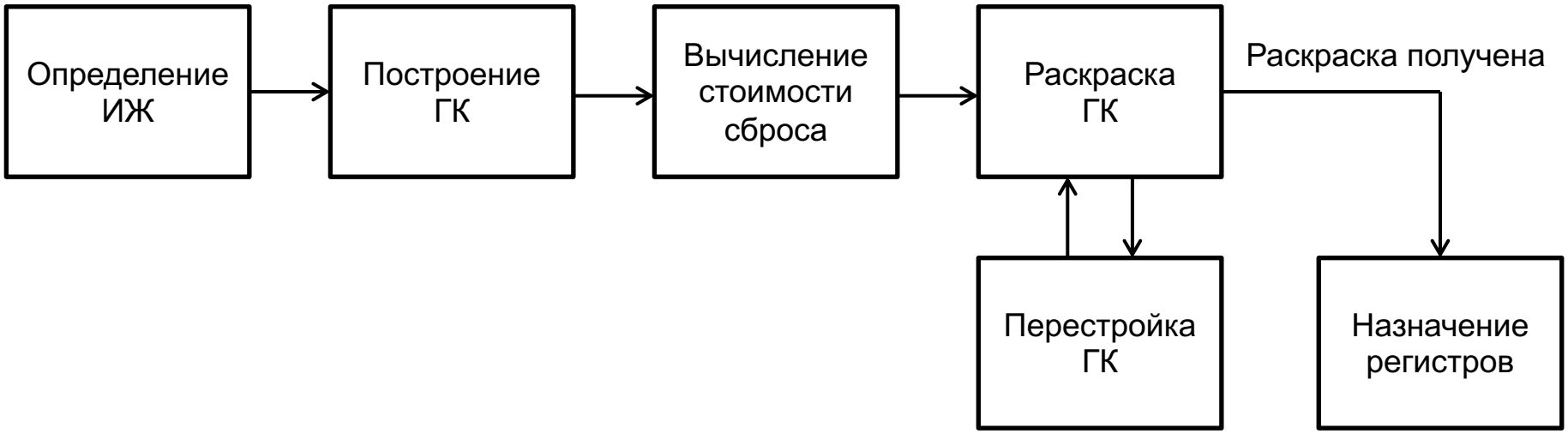
Если нераскрашенных узлов не осталось, алгоритм завершается. Если часть узлов ГК осталась нераскрашенной, то для каждого такого узла либо генерируются команды сброса (*слив*), либо интервал жизни, соответствующий узлу *расщепляется* (на два или более подинтервалов, рассматривается далее), после чего ГК перестраивается с учетом слива и/или разделенных узлов. После перестройки ГК делается переход на первую фазу.

◊ Наиболее распространенный эвристический критерий слива узла – минимум отношения

$$\frac{цена\_сброса}{степень\_узла}$$

2.3 Глобальное распределение и назначение регистров

2.3.11 Структура распределителя регистров



- Найти ИЖ всех переменных, построить ГК, вычислить стоимость сброса для каждого ИЖ, выполнить раскраску ГК. После этого либо каждый ИЖ получит цвет (положительный исход), либо часть ИЖ останутся неокрашенными (отрицательный исход).
- В случае положительного исхода каждому ИЖ присваивается физический регистр.
- В случае отрицательного исхода ГК перестраивается и снова выполняется раскраска ГК.

2.3 Глобальное распределение и назначение регистров

2.3.12 Перестройка графа конфликтов

◊ Перестройка ГК достигается помимо слияния ИЖ (п. 2.3.9) с помощью *слива* переменных (п. 2.3.13) и *расщепления* ИЖ (п.2.3.14).

2.3 Глобальное распределение и назначение регистров

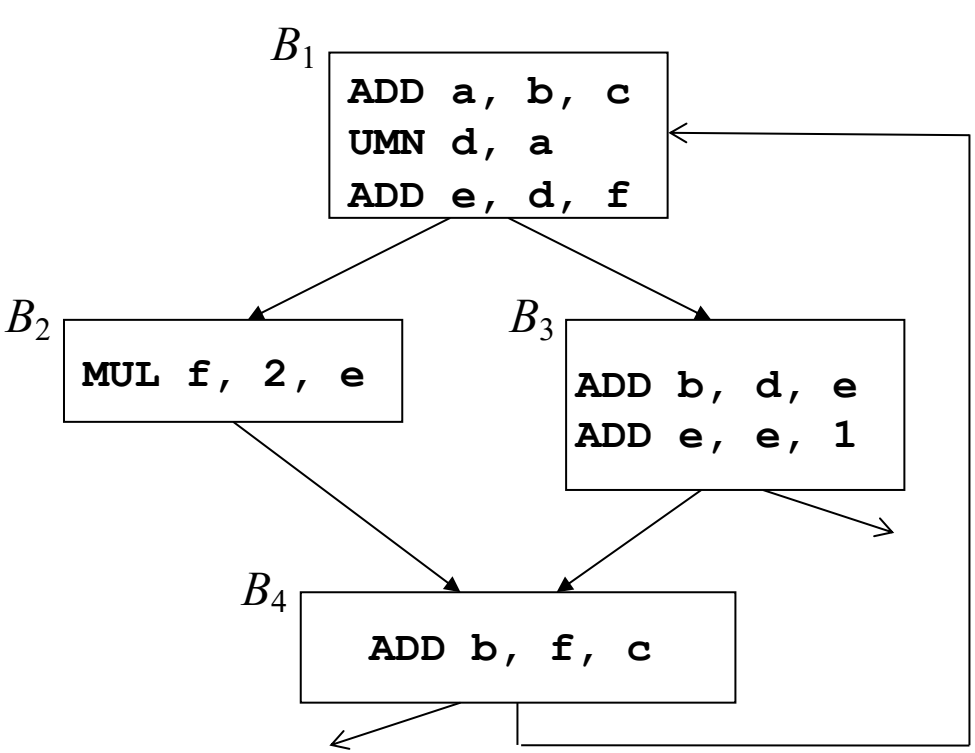
2.3.13 Сливание переменных

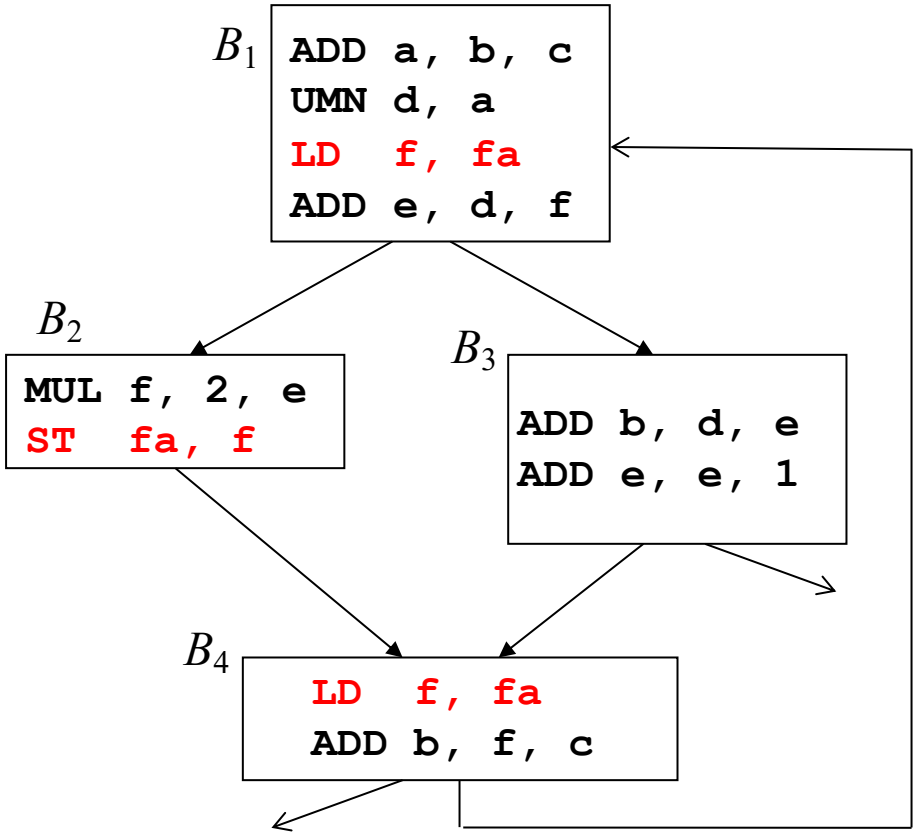
◊ *Слив* временной переменной **t** состоит в следующем:

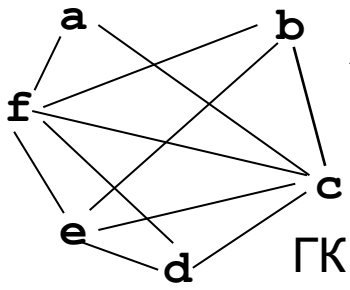
- в «автоматической» памяти процедуры выделить ячейку **ta** для хранения значений **t** (эту ячейку называют «дом» **t**);
- перед каждой командой, использующей **t**, вставить команду **LD t, ta**;
- после каждой команды, определяющей **t**, вставить команду **ST ta, t**;
- просматривают текст полученной процедуры с целью выявить и удалить лишние команды **LD** и **ST** (иногда это удается).

2.3 Глобальное распределение и назначение регистров

2.3.13 Слив переменных

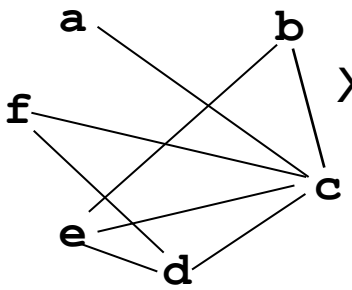






Хроматическое число 4

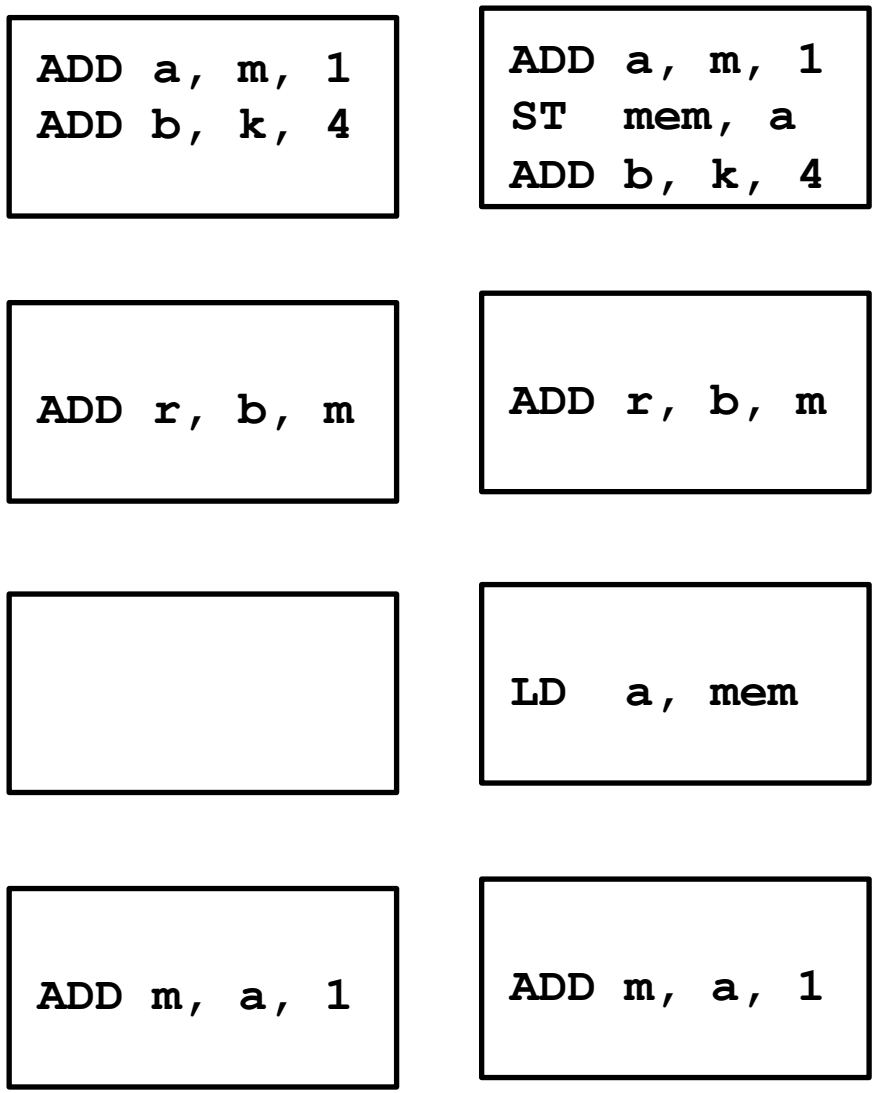
ГК



Хроматическое число 3

2.3 Глобальное распределение и назначение регистров

2.3.14 Расщепление интервалов жизни



a и **b** в конфликте

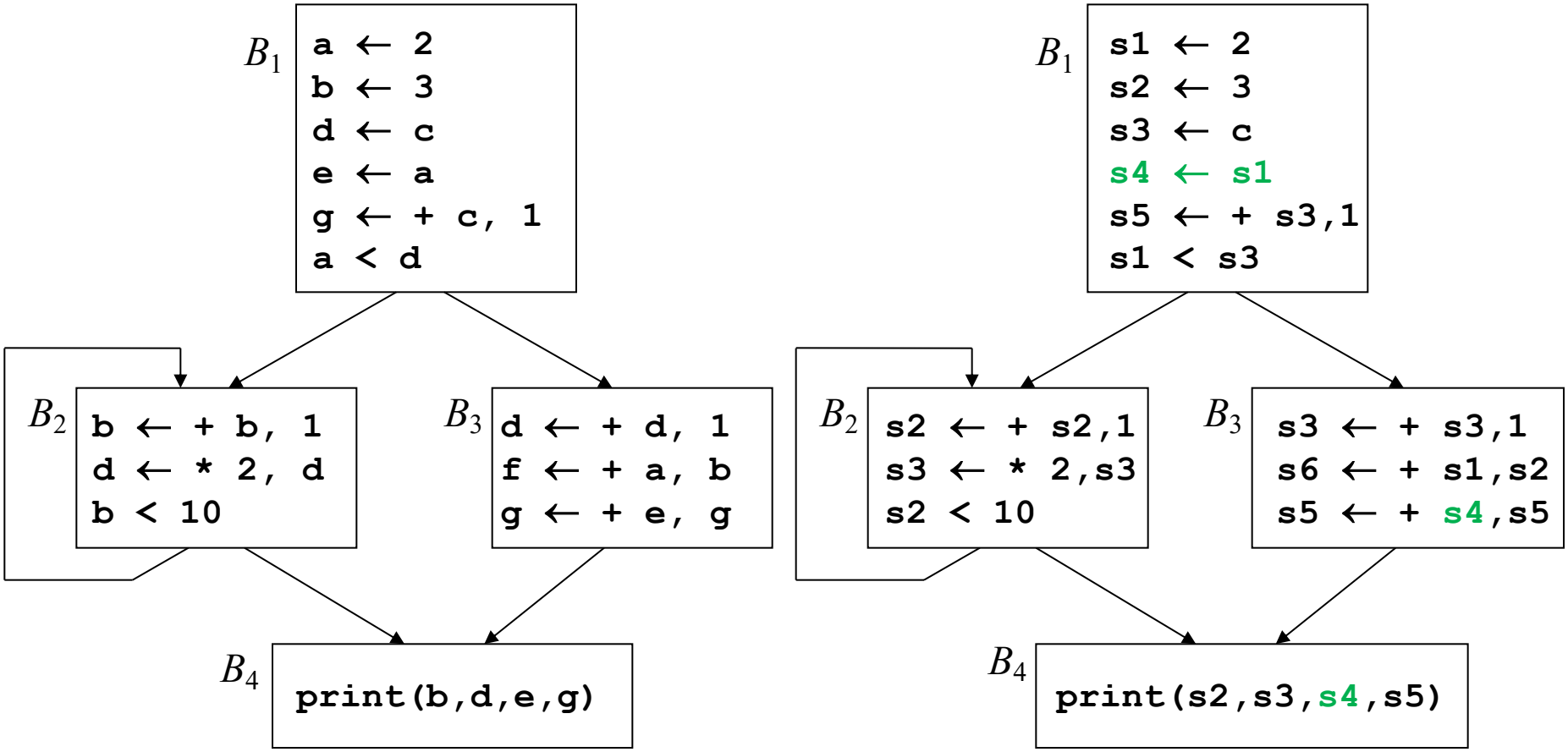
Конфликт между **a** и **b** устранен

- На левом рисунке LR_a и LR_b пересекаются.
  Расщепив LR_a на два **LR**, удается устранить конфликт.
- При этом команды **ST** и **LD** применяются только здесь, а не перед всеми **a**.

2.3 Глобальное распределение и назначение регистров

2.3.15 Примеры глобального распределения регистров

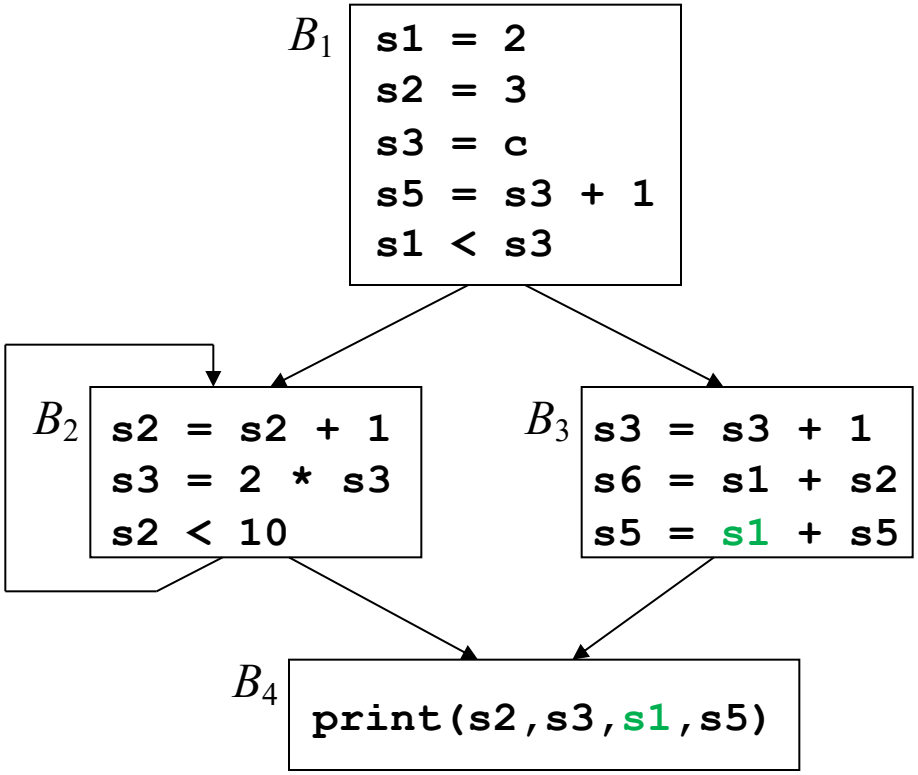
1) Распределение виртуальных (символических) регистров



Применим **слияние** интервалов к команде копирования **s4 = s1** (в блоке *B*1).

2.3 Глобальное распределение и назначение регистров

Применим **слияние** интервалов к команде копирования **s4 = s1** (в блоке *B*1). Получим



Оценим стоимости сброса для оставшихся символических регистров **s1**, **s2**, **s3**, **s5** и **s6**

Символич. регистр	Стоимость сброса B_1	B_2	B_3	B_4
s1	2.0			
s2	1.0 + 21.0 + 2.0 + 2.0 = 26.0			
s3	6.0 + 20.0 + 4.0 + 2.0 = 32.0			
s5	2.0		+ 4.0 + 2.0 = 8.0	
s6				∞

Стоимость сброса вычисляется по формуле

$$Spill\_cost = DefWt \cdot \sum_{def \in LR} 10^{Depth(def)} + UseWt \cdot \sum_{use \in LR} 10^{Depth(use)} - CopyWt \cdot \sum_{copy \in LR} 10^{Depth(copy)}$$

где *def*, *use* и *copy* – отдельные команды определения, использования и копирования в *LR*, а *DefWt*, *UseWt* и *CopyWt* – стоимости соответствующих команд

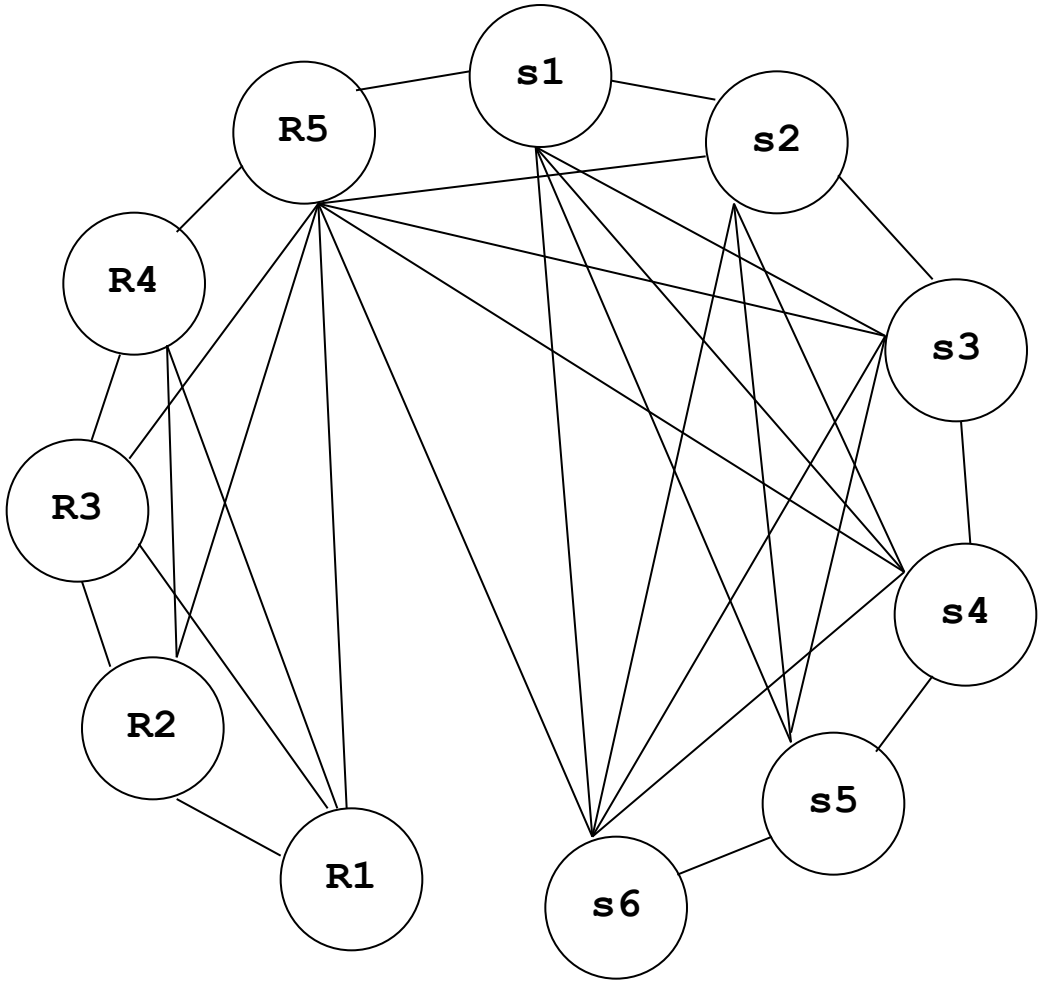
При вычислении стоимости сброса считается $DefWt = UseWt = 2.0$, $CopyWt = 1.0$

Стоимость s6 = ∞, так как регистр s6 – не является живым

2.3 Глобальное распределение и назначение регистров

2.3.15 Примеры глобального распределения регистров

2) Построение графа конфликтов
(**s1**, …, **s6** – виртуальные регистры, **R1**, …, **R5** – физические регистры)

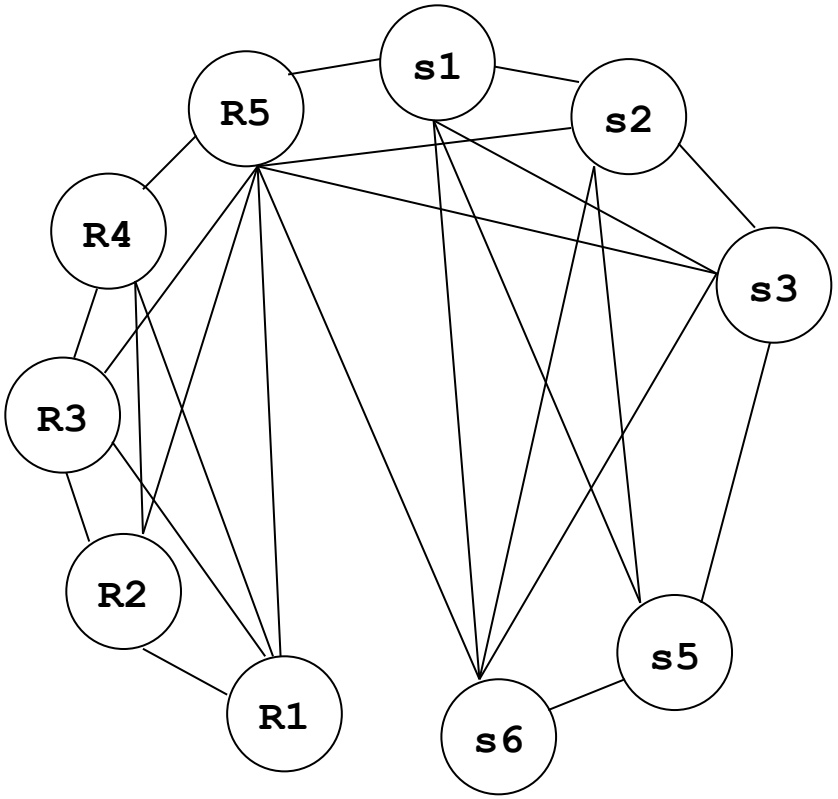


- ◊ Все физические регистры считаются всегда живыми
- ◊ Пусть частота выполнения блоков B_1, B_3 и B_4 равна 1, а частота выполнения блока B_2 равна 7
- ◊ Каждому символическому регистру соответствует один интервал жизни, поэтому, например, **LR_a** и **s1** – синонимы
- ◊ На символических регистрах **s1**, **s2**, **s3**, **s4** хранятся значения, поэтому результаты всех вычислений в блоках B_2 и B_3 будем помещать на **R5**

2.3 Глобальное распределение и назначение регистров

2.3.15 Примеры глобального распределения регистров

Граф конфликтов и его матрица смежности примут вид

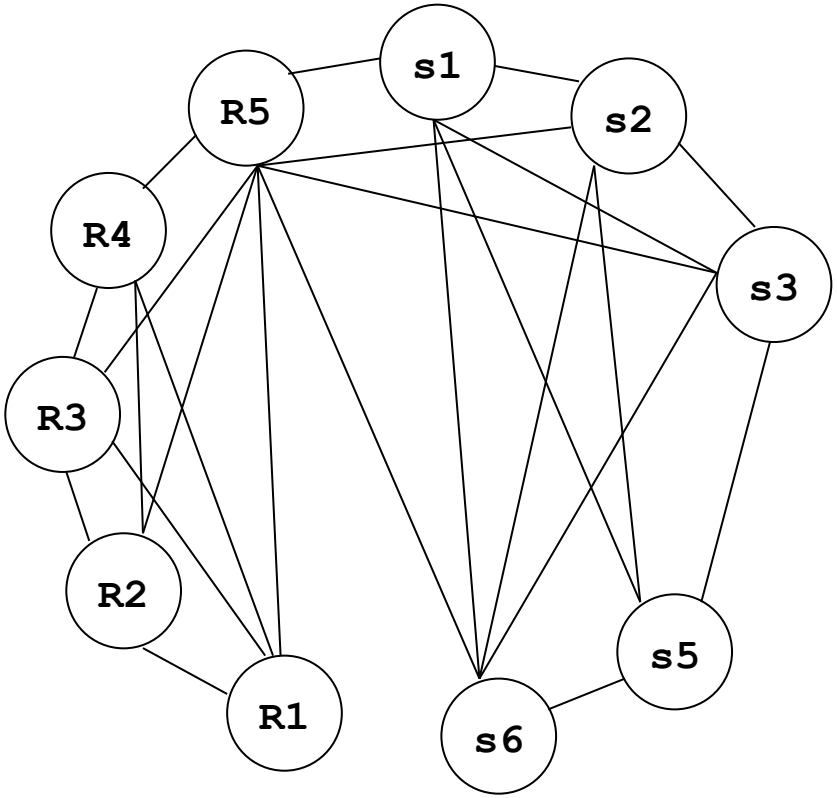


	$R1$	$R2$	$R3$	$R4$	$R5$	$s1$	$s2$	$s3$	$s5$
$R2$	1								
$R3$	1	1							
$R4$	1	1	1						
$R5$	1	1	1	1					
$s1$	0	0	0	0	1				
$s2$	0	0	0	0	1	1			
$s3$	0	0	0	0	1	1	1		
$s5$	0	0	0	0	0	1	1	1	
$s6$	0	0	0	0	1	1	1	1	1

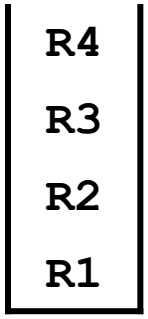
2.3 Глобальное распределение и назначение регистров

2.3.15 Примеры глобального распределения регистров

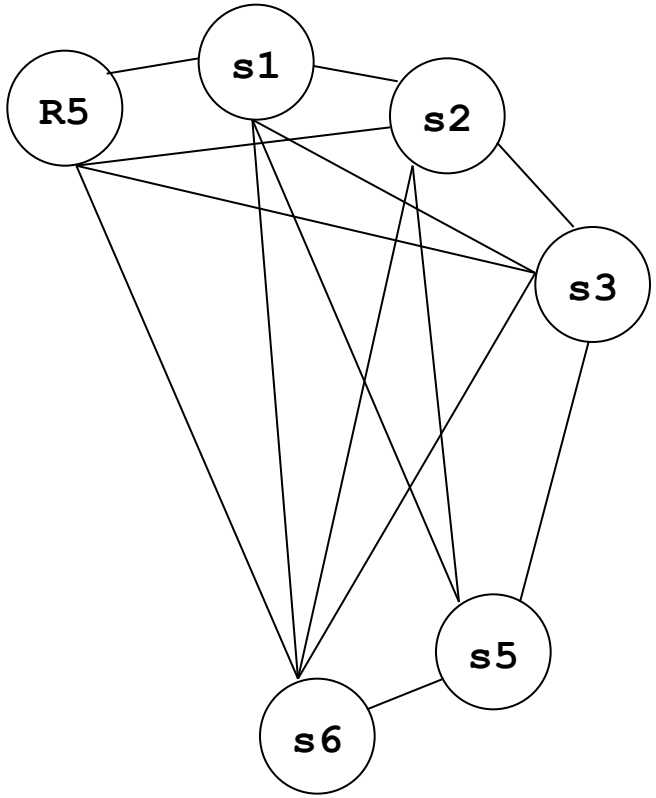
Поскольку каждый из узлов **R1**, **R2**, **R3**, **R4** имеет меньше пяти смежных узлов, заталкиваем их в стек (порядок произвольный) и удаляем из графа конфликтов. Получается граф, изображенный на правом рисунке



Граф конфликтов до исключения узлов **R1**, **R2**, **R3**, **R4**



стек

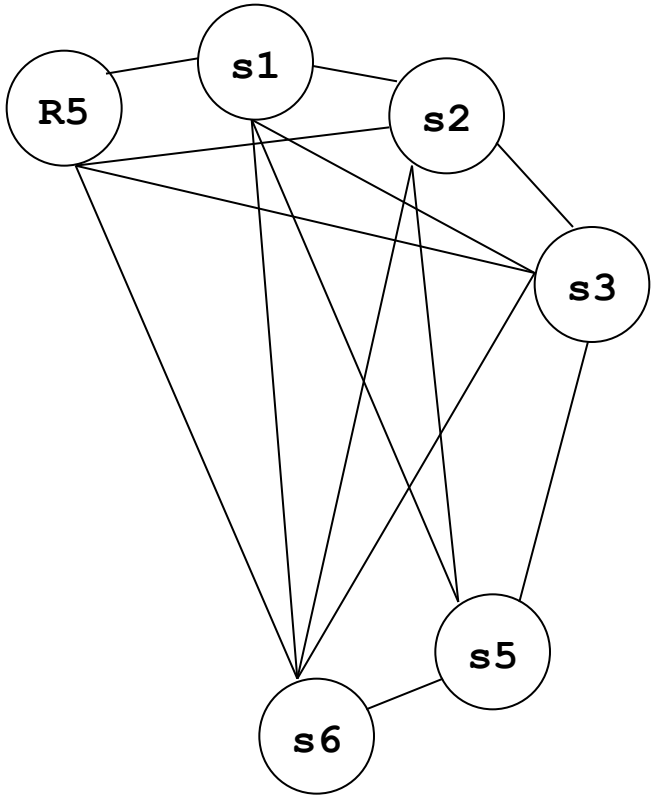


Граф конфликтов после исключения узлов **R1**, **R2**, **R3**, **R4**

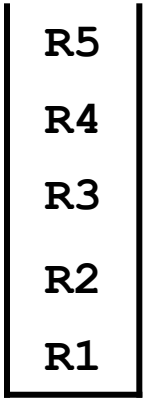
2.3 Глобальное распределение и назначение регистров

2.3.15 Примеры глобального распределения регистров

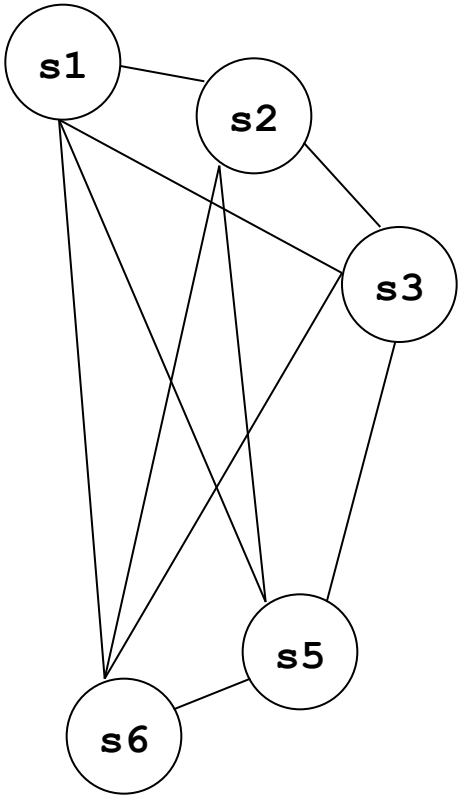
Теперь узел **R5** имеет меньше пяти смежных узлов; заталкиваем и его в стек и удаляем из графа конфликтов. Получается граф, изображенный на правом рисунке



Граф конфликтов
до исключения узла **R5**



стек

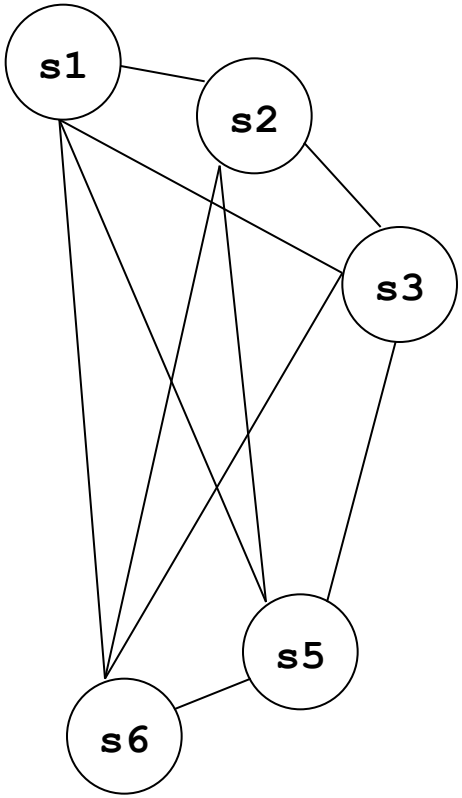


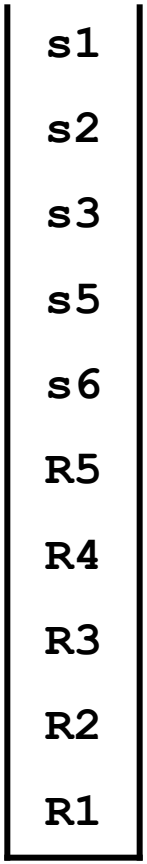
Граф конфликтов
после исключения узла **R5**

2.3 Глобальное распределение и назначение регистров

2.3.15 Примеры глобального распределения регистров

Теперь все оставшиеся узлы графа конфликтов имеют меньше пяти смежных узлов; заталкиваем их (в произвольном порядке) в стек и удаляем из графа конфликтов.





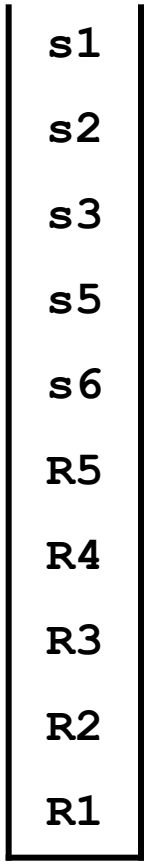
стек

2.3 Глобальное распределение и назначение регистров

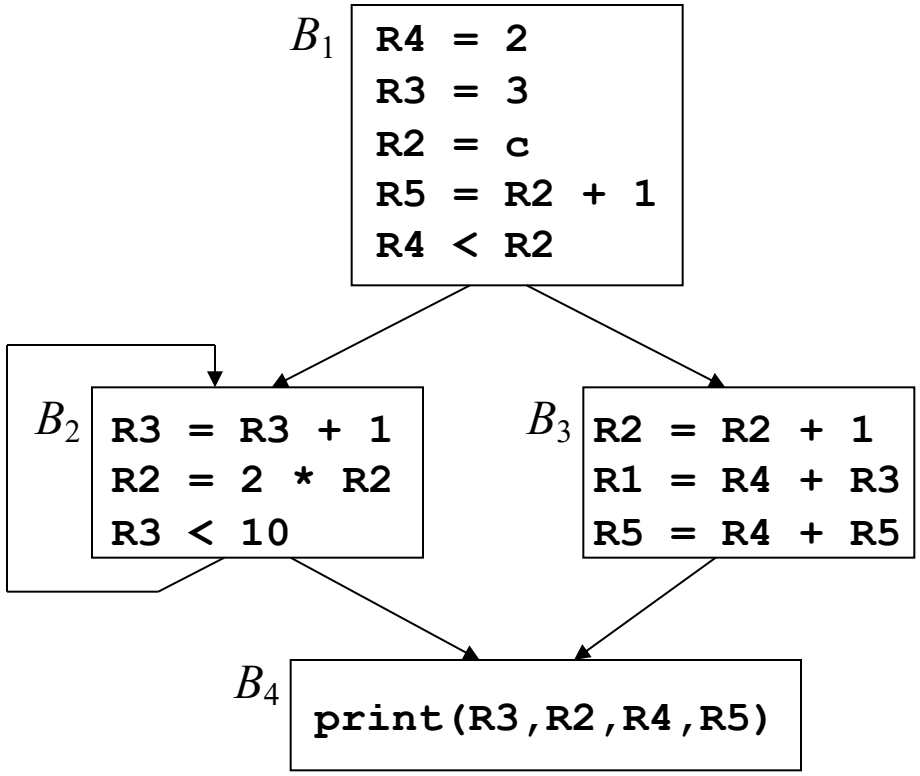
2.3.15 Примеры глобального распределения регистров

Выталкиваем регистры из стека и присваиваем каждому свободный цвет (всего имеется пять цветов). **R5** имеет 4 смежных узла: **s1**(1), **s2**(2), **s3**(3) и **s6**(5). Следовательно, ему можно присвоить цвет 4 (только он свободен).

Регистр	Цвет
s1	1
s2	2
s3	3
s5	4
s6	5
R5	4
R4	1
R3	2
R2	3
R1	5



стек



2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

- ◊ Пусть доступно R регистров и пусть составлен список из n интервалов жизни. Алгоритм Linear scan должен распределить на регистры как можно больше ИЖ таким образом, чтобы никакие пересекающиеся ИЖ не были распределены на один и тот же регистр.
- ◊ Если n > R и ИЖ пересекаются в некоторой точке, то по крайней мере n − R ИЖ должны быть размещены в памяти.
- ◊ Количество пересекающихся ИЖ изменяется только в начале и в конце одного из ИЖ.
- ◊ Все ИЖ сначала помещаются в список, отсортированный по возрастанию их начал. Тогда алгоритм может быстро просмотреть все ИЖ, перескакивая с одного начала ИЖ к следующему. На каждом шаге алгоритм поддерживает список активных ИЖ, которые пересекаются в данной точке и уже размещены на регистрах. Этот список активных ИЖ поддерживается отсортированным по возрастанию концов входящих в него ИЖ.

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ На псевдокоде

```
active ← {}
for each live interval i в порядке
    возрастания начал
        ExpireOldIntervals(i) //окончание
                              старых интервалов
    if length(active) = R then
        SpillAtInterval(i)  //слив интервала i
    else
        register[i] ← регистр, удаляемый из
            пула свободных регистров
        добавить i в active, отсортированный
            по возрастанию концов ИЖ
```

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ На псевдокоде

```
SpillAtInterval(i)
     spill ← last interval in active
     if endpoint[spill] > end point[i] then
          register[i] ← register[spill]
          location[spill] ← new stack location
          remove spill from active
          add i to active, sorted by increasing
               end point
     else
          location[i] ← new stack location
```

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ На псевдокоде

```
ExpireOldIntervals(i)
      foreach interval j in active, in order of
        increasing end point
           if endpoint[j] ≥ startpoint[i] then
                return
           remove j from active
           add register[j] to pool of free
                      registers
```

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

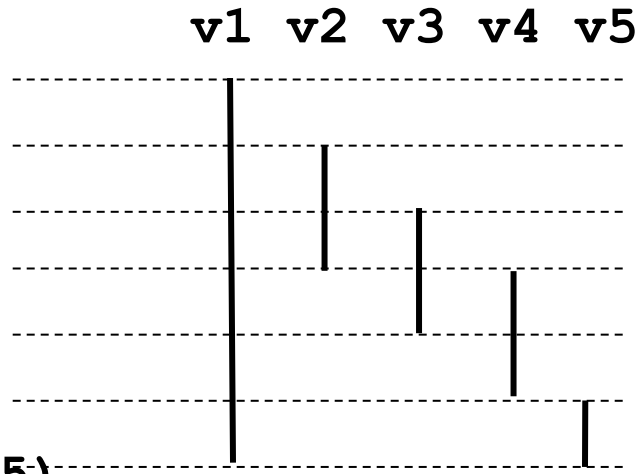
◊ При обнаружении каждого нового ИЖ алгоритм просматривает список **`active`** с начала до конца. Во время просмотра из списка удаляются все закончившиеся ИЖ (ИЖ, которые больше не пересекаются, так как конец одного из них раньше начала другого) и делает соответствующий регистр доступным для распределения.

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ **Пример**. Пусть доступно всего два регистра **r1** и **r2**.

```
(1) v1 = 10
(2) v2 = 20
(3) v3 = v1 + v2
(4) v4 = v2 + v3
(5) v1 = v3 + v4
(6) v5 = v4 + v1
(7) return (v1 + v5)
```



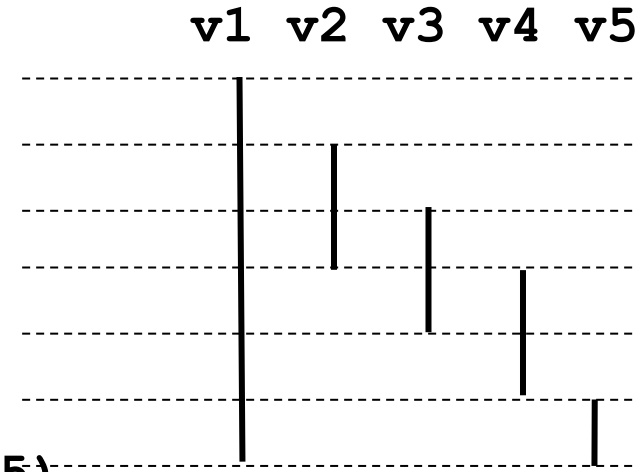
◊ ИЖ обрабатываются в порядке **v1,v2,v3,v4,v5.**
Сначала список **Active** пуст.
На первом шаге обрабатывается ИЖ **v1**. Так как список **Active** пуст, регистр **r1** выделяется для **v1**, и **v1** заносится в список **Active**.

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ **Пример**. Пусть доступно всего два регистра **r1** и **r2**.

```
(1) v1 = 10
(2) v2 = 20
(3) v3 = v1 + v2
(4) v4 = v2 + v3
(5) v1 = v3 + v4
(6) v5 = v4 + v1
(7) return (v1 + v5)
```

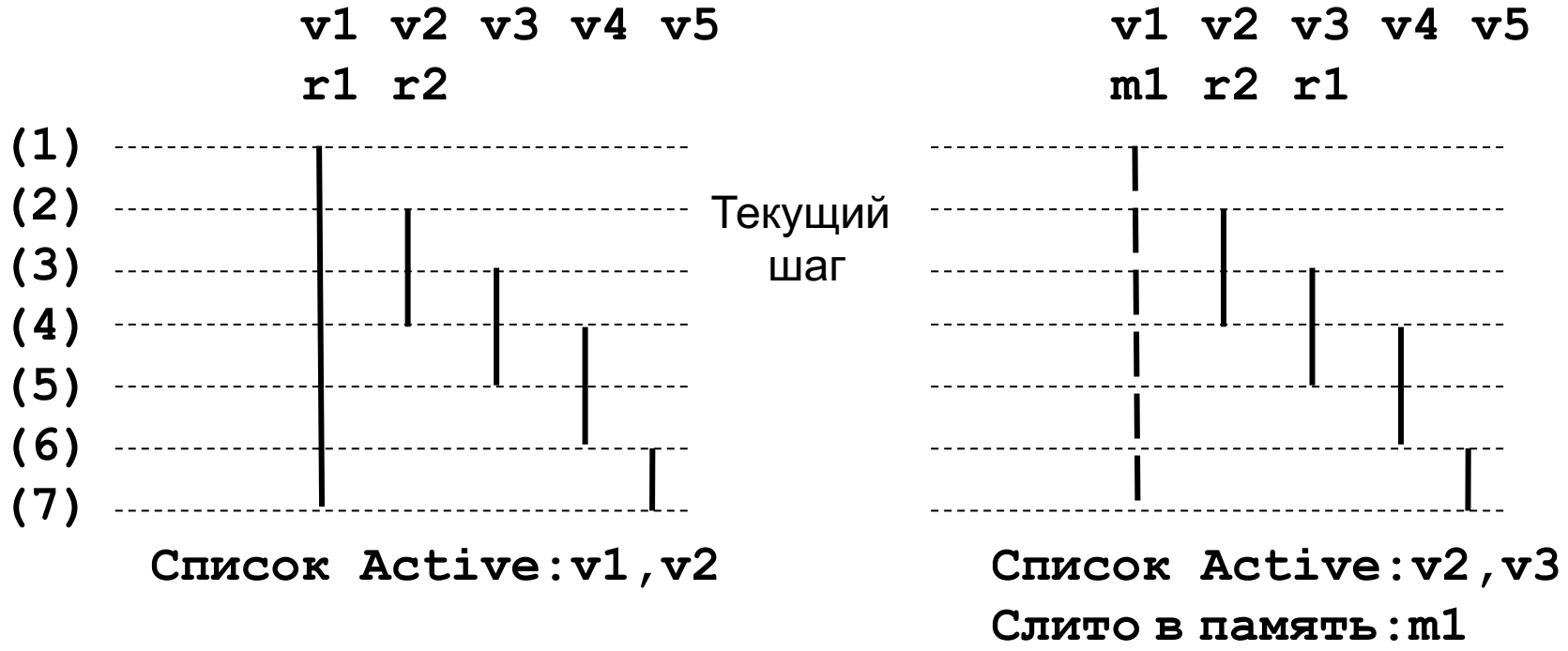


◊ ИЖ обрабатываются в порядке **v1,v2,v3,v4,v5.**
После первого шага список **Active** содержит ИЖ **v1.**
На втором шаге обрабатывается ИЖ **v2**. Так как **v1** продолжает оставаться активным, **v1** не исключается из списка **Active**, и для ИЖ **v2** выделяется единственный свободный регистр **r2**, а **v2** заносится в список **Active**.

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ **Пример**. Пусть доступно всего два регистра **r1** и **r2**.



◊ На рисунках показано состояние до и после третьего шага.

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ **Пример**. Пусть доступно всего два регистра **r1** и **r2**.

```
(1) v1 = 10              (1) m1 = 10
(2) v2 = 20              (2) r2 = 20
(3) v3 = v1 + v2         (3) r1 = m1 + r2
(4) v4 = v2 + v3         (4) r2 = r2 + r1
(5) v1 = v3 + v4         (5) m1 = r1 + r2
(6) v5 = v4 + v1         (6) r1 = r2 + m1
(7) return (v1 + v5)     (7) return (m1 + r1)
```

◊ Полученное распределение регистров показано на рисунке. Метод раскраски графа конфликтов обеспечивает для рассмотренного примера лучшее распределение регистров.

2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

◊ **Пример**. Пусть доступно всего два регистра **r1** и **r2**.

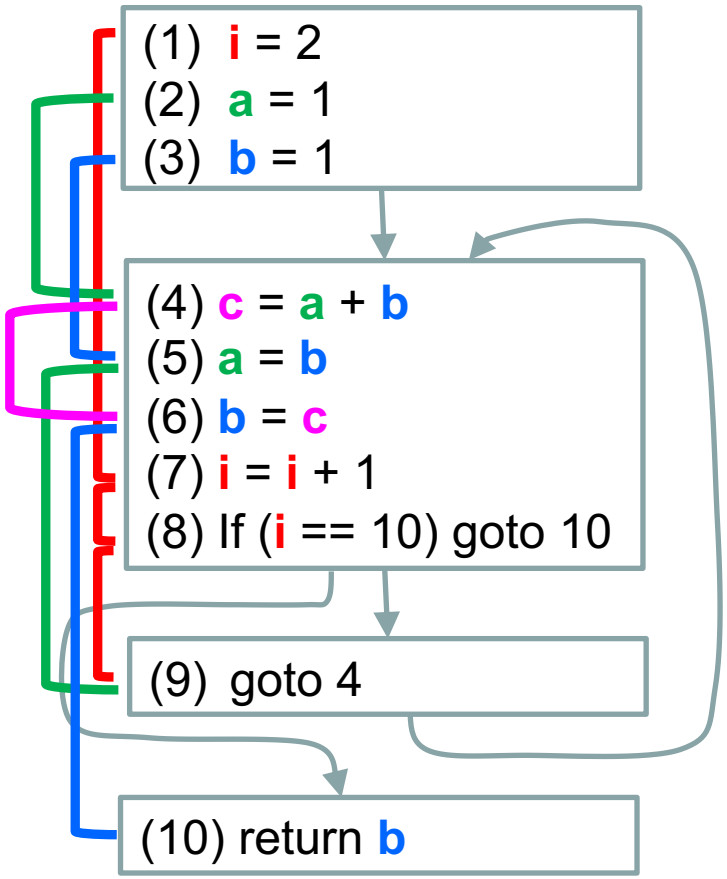
◊ Метод раскраски графа конфликтов обеспечивает для рассмотренного примера лучшее распределение регистров.

```
(1) v1 = 10              (1) r2 = 10
(2) v2 = 20              (2) r1 = 20
(3) v3 = v1 + v2         (3) r2 = r2 + r1
(4) v4 = v2 + v3         (4) r1 = r1 + r2
(5) v1 = v3 + v4         (5) r2 = r2 + r1
(6) v5 = v4 + v1         (6) r1 = r1 + r2
(7) return (v1 + v5)     (7) return (r2 + r1)
```

◊ Но метод раскраски графа конфликтов имеет асимптотическую временную сложность $O(n^2)$.
Метод линейного сканирования реализует распределение регистров за один просмотр текста программы и имеет асимптотическую временную сложность $O(n)$.

Построение интервалов жизни переменных для Linear scan при наличии циклов

(1) i = 2
(2) a = 1
(3) b = 1
(4) c = a + b
(5) a = b
(6) b = c
(7) i = i + 1
(8) If (i == 10) goto 10
(9) goto 4
(10) return b



Имя	ИЖ
i	[1 .. 8]*
a	[2 .. 8]*
b	[3 .. 10]
c	[4 .. 6]

i, a, b – живы через обратную дугу (т.е. определяются в конце, а читаются в начале цикла).

* Последняя команда перехода не включается в интервал жизни (если только переменная в ней не читается), чтобы не создавать преимущество в распределении на регистр переменным, которые читались в самом конце цикла, перед теми, которые живы через обратную дугу.

Пример работы алгоритма Linear scan (2 регистра)

(1) i = 2
(2) a = 1
(3) b = 1
(4) c = a + b
(5) a = b
(6) b = c
(7) i = i + 1
(8) If (i == 10) goto 10
(9) goto 4
(10) return b

Name	Interval	Register	Active set
i	[1 .. 8]	R1	i
a	[2 .. 8]		
b	[3 .. 10]		
c	[4 .. 6]		

Пример работы алгоритма Linear scan (2 регистра)

(1) i = 2
(2) a = 1
(3) b = 1
(4) c = a + b
(5) a = b
(6) b = c
(7) i = i + 1
(8) If (i == 10) goto 10
(9) goto 4
(10) return b

Name	Interval	Register	Active set
i	[1 .. 8]	R1	i
a	[2 .. 8]	R2	i, a
b	[3 .. 10]		
c	[4 .. 6]		

Пример работы алгоритма Linear scan (2 регистра)

(1) i = 2
(2) a = 1
(3) b = 1
(4) c = a + b
(5) a = b
(6) b = c
(7) i = i + 1
(8) If (i == 10) goto 10
(9) goto 4
(10) return b

Name	Interval	Register	Active set
i	[1 .. 8]	R1	i
a	[2 .. 8]	R2	i, a
b	[3 .. 10]	mem	i, a
c	[4 .. 6]		

Пример работы алгоритма Linear scan (2 регистра)

(1) i = 2
(2) a = 1
(3) b = 1
(4) c = a + b
(5) a = b
(6) b = c
(7) i = i + 1
(8) If (i == 10) goto 10
(9) goto 4
(10) return b

Name	Interval	Register	Active set
i	[1 .. 8]	R1	i
a	[2 .. 8]	~~R2~~ mem	i, a
b	[3 .. 10]	mem	i, a
c	[4 .. 6]	R2	c, i

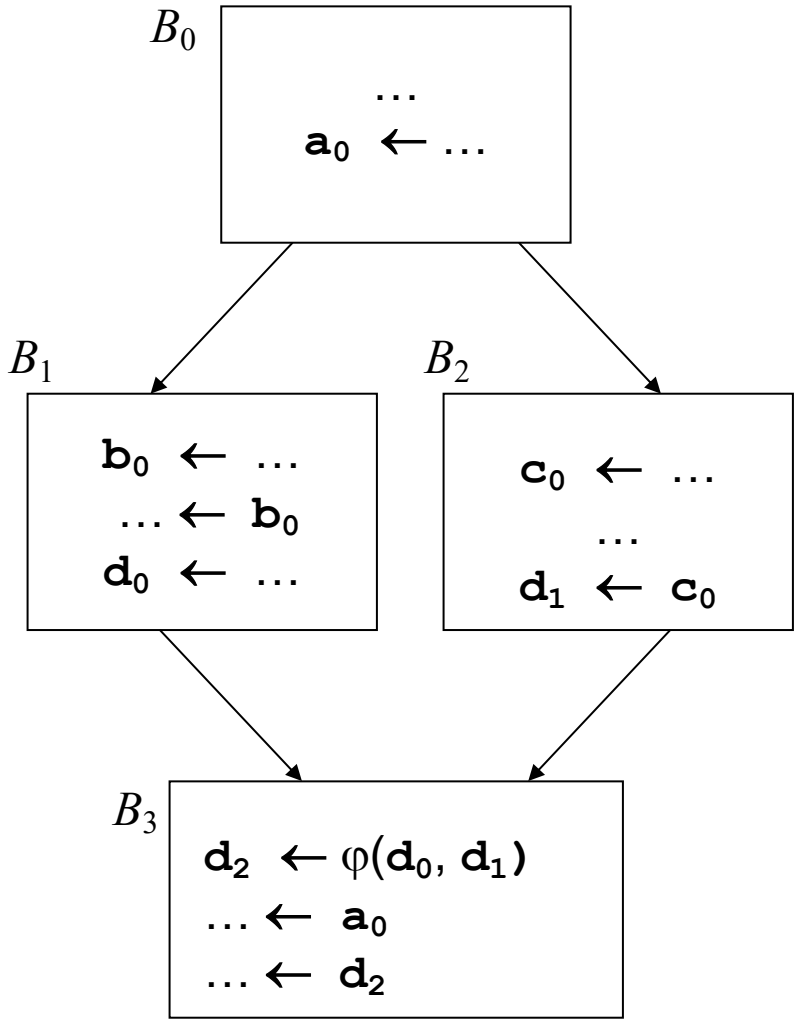
2.3 Глобальное распределение и назначение регистров

2.3.11 Быстрый алгоритм распределения регистров – метод линейного сканирования (Linear scan)

- ◊ В отличие от метода раскраски графа конфликтов, имеющего асимптотическую временную сложность $O(n^2)$, метод линейного сканирования реализует распределение регистров всего за один просмотр текста программы и имеет асимптотическую временную сложность $O(n)$.
- ◊ Разработчики *JIT*-компиляторов (например, для языка *Java* или для языка *Python*) предпочитают метод линейного сканирования.
- ◊ В последнее время используется модификация метода линейного сканирования, называемая *Second Chance Binpacking*.
  Во многих случаях она позволяет увеличить точность линейного сканирования, сохраняя асимптотическую временную сложность $O(n)$.

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления



Фрагмент кода
в *SSA*-представлении

◊ Программа как множество ***меток*** с выделенной меткой ***start***

◊ Метки содержат инструкции и соединены ***ребрами*** графа потока управления

◊ Все ***φ-инструкции*** на одной метке выполняются одновременно перед основными инструкциями

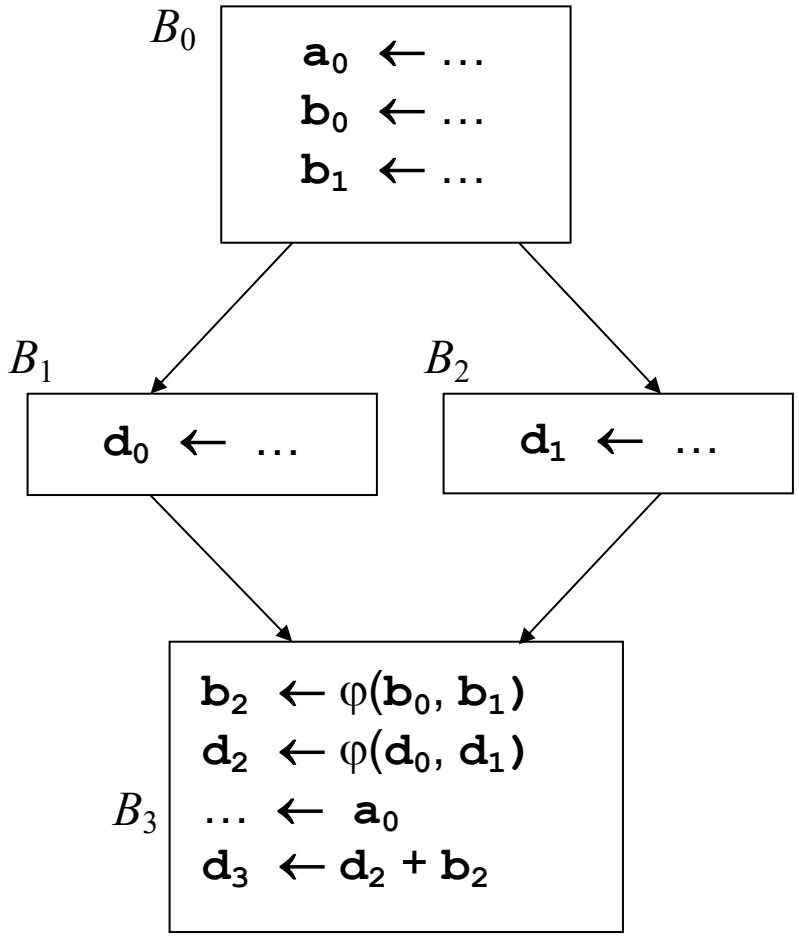
◊ Метка (базовый блок) *A* ***доминирует*** метку *B*, если все пути из ***start*** в *B* проходят через *A*

◊ Если переменная в базовом блоке используется только в φ-инструкциях, считаем, что такое использование происходит на входящем ребре из соответствующего базового блока

◊ Такая дополнительная абстракция позволяет утверждать, что в корректной SSA-форме **определение значения доминирует все его использования**

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления



Фрагмент кода
в *SSA*-представлении

◊ Граф конфликтов: SSA-переменные ***конфликтуют***, если существует точка в программе, когда они обе живы

◊ Таким переменным **не** может быть назначен один регистр

◊ Рассматриваем не «интервалы жизни», а «части» графа потока управления

◊ Если переменная в базовом блоке используется только в φ-инструкциях, она может быть не жива в нем, но жива на выходе из соответствующего блока

◊ Если переменная жива в двух блоках, то она жива и во всех блоках на любом пути между ними

◊ В примере слева:

- ◊ a_0 конфликтует со всеми, кроме d_3
- ◊ d_3 **не** конфликтует с b_2 и d_2
- ◊ b_2 **не** конфликтует с b_0, b_1, d_0, d_1
- ◊ b_0 конфликтует с b_1 и d_0 , но не с d_1
- ◊ попарно **не** конфликтуют d_0, d_1, d_2

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Обозначим метку, на которой определяется SSA-переменная ***v***, как D_v.

Лемма 1. Если SSA-переменные ***a*** и ***b*** конфликтуют, то либо D_a доминирует D_b, либо наоборот D_b доминирует D_a.
Доказательство. Пусть ***a*** и ***b*** одновременно живы в метке **L**. D_a доминирует **L** и D_b доминирует **L**. Возьмем один произвольный путь из **start** в **L**. Он по определению доминирования проходит через метки D_a и D_b, без ограничения общности можно считать что именно в таком порядке.
Предположим, D_a не доминирует D_b. Тогда существует путь из **start** в D_b не проходящий через D_a, назовем его «альтернативный». Если в изначальном пути из **start** в **L** заменить участок пути до D_b на «альтернативный», мы получим новый путь в **L**, не проходящий через D_a. Получили противоречие с тем, что D_a доминирует **L**.

Лемма 2. Если SSA-переменные ***a*** и ***b*** конфликтуют, и D_a доминирует D_b , то переменная ***a*** жива в метке D_b.
Доказательство. В силу того, что существует путь из D_a в **L**.

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Теорема 1. Для любой клики в графе конфликтов, состоящей из вершин, соответствующих SSA-переменным c_1, c_2, … c_n существует метка, где определены все эти переменные.

Доказательство. По лемме 1 метки D_{c1}, D_{c2}, … D_{cn} где определены соответствующие SSA-переменные образуют полностью упорядоченное множество относительно отношения доминирования. При сортировке по отношению доминирования нужно взять нижнюю метку (которую доминируют все остальные), и по лемме 2 в такой метке D_{ci} будут живы все переменные из множества c_1, c_2, … c_n.

Лемма 3. Если в графе конфликтов для трех переменных ***a***, ***b*** и ***c*** существуют ребра ***ab*** и ***bc***, но отсутствует ребро ***ac***, и D_a доминирует D_b, то D_b доминирует D_c.

Доказательство. По лемме 1: D_b доминирует D_c или D_c доминирует D_b. Предположим, D_c доминирует D_b. Тогда переменная ***c*** жива в D_b. Поскольку ***a*** и ***b*** конфликтуют и D_a доминирует D_b, по лемме 2 переменная ***a*** также жива в D_b. Получается, переменные ***a*** и ***c*** одновременно живы в D_b, что противоречит условию отсутствия ребра ***ac***.

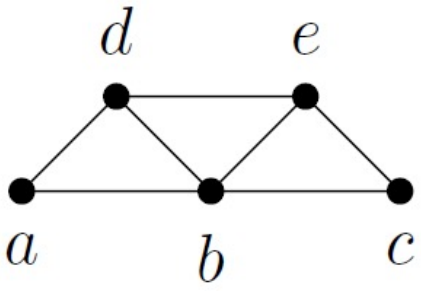
2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Вернемся к алгоритму раскраски графа конфликтов, при котором вершины по очереди из него удаляются и помещаются в стек. Далее, при восстановлении вершина раскрашивается в один из цветов, который еще не используется на уже восстановленных соседних вершинах.

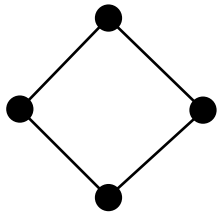
Представим, что в процессе удаления на каждом шаге удается подобрать вершину, оставшиеся **соседи которой образуют клику**. Тогда при восстановлении соседи уже раскрашены в разные цвета и хроматическое число графа соответствует размеру максимальной клики в графе.

Такой порядок называется *perfect elimination order (PEO)*. Графы, для которых существует такой порядок, называются *хордальными*.



PEO: a, d, b, e, c

No PEO: b, a, c, d, e



2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Теорема 2. Переменная ***v*** может быть добавлена в стек PEO графа конфликтов, если все переменные, чьи определения доминируются меткой D_v, уже добавлены в данный PEO (т. е. удалены из графа ранее).
Доказательство. Предположим, ***v*** нельзя добавить. Значит, у нее существует два соседа ***a*** и ***b*** в графе конфликтов, не конфликтующих между собой. По лемме 1 и свойству уже удаленных переменных, D_a доминирует D_v. С другой стороны, в графе существуют ребра ***av*** и ***vb***, но нет ребра **ab**. Получаем, по лемме 3, D_v доминирует D_b. Но это противоречит условию, так как тогда ***b*** должна быть уже удалена.

Алгоритм 1. Получается, что для программы в SSA-форме можно построить PEO ее графа конфликтов обойдя ее дерево доминирования. Для раскраски рекурсивно обходим базовые блоки по дереву доминирования.

1. Сначала по порядку распределяем регистр/цвет для переменных данного базового блока. Все переменные с которыми очередная выбранная конфликтует – уже конфликтуют между собой, и можно выбрать любой регистр/цвет, которого нет в клике соседей.
2. После обработки текущего блока, по очереди рекурсивно переходим к каждому из поддеревьев в дереве доминаторов.

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Минусы обычного ***циклического подхода*** (схема выше):

1) Может быть много итераций, если регистров мало и требуется много операций сброса
2) Каждый раз после сброса требуется перестроить граф конфликтов
3) Эвристический алгоритм раскраски может не найти решение, даже если оно на самом деле существует – будут лишние сбросы
4) Слияние в редких случаях увеличивает хроматическое число графа

Новый ***прямой подход*** позволяет избавиться от многократных перестроений графа для случая, когда регистров недостаточно. Можно заранее построить операции сброса таким образом, чтобы регистров гарантированно хватало в каждой точке программы и после вставки операций сброса раскраска гарантированно будет успешной.

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Согласно теореме 1, для любой клики в графе конфликтов существует точка в программе, где все переменные клики живы одновременно. Поэтому, при построении операций сброса, достаточно убедиться, что ни в какой точке программы число живых переменных не превышает число регистров. Тогда в графе конфликтов не может существовать клика б**о**льшего размера, чем число регистров.

Эвристическим ***критерием*** выбора значения для сброса в память является наибольшее по числу инструкций расстояние до «ближайшего» будущего использования. Таких точек использования может быть множество, учитывается именно кратчайшее расстояние по числу инструкций до одной из них по путям графа потока управления.

2.3 Глобальное распределение и назначение регистров

2.3.20 Алгоритм распределения регистров для SSA–представления

Алгоритм 2 (эвристический алгоритм сброса). Для каждой переменной имеется область графа потока управления где она «жива».

1. Сначала базовые блоки обрабатываются отдельно в произвольном порядке.
2. Пусть I_B это множество всех переменных живых в начале блока B, включая результаты его φ-инструкции, *k* – число доступных регистров.
3. Если размер I_B превосходит *k*, то выкидываем из него элементы согласно ***критерию***, пока их не станет ровно *k.*
4. Далее, обрабатываем по порядку инструкции внутри базового блока и поддерживаем множество не более чем *k* живых значений, хранящихся в данный момент на регистрах. Если для результата очередной инструкции требуется освободить один из регистров, выполняем сброс одного из значений по ***критерию****.*
5. Перед будущими использованиями появляется инструкция восстановления значения из памяти, которой тоже будет нужен регистр, (и она не повлияет на число живых переменных в этой точке)
6. Если какое-то значение из I_B в процессе работы над базовым блоком было сброшено до своего первого использования – оно также выкидывается из I_B.
7. В конце базового блока фиксируется множество O_B – живых значений, хранящихся на регистрах в конце блока.
8. После обработки всех блоков, для каждого ребра в графе потока управления, идущего из блока P в блок B, если в множество I_B входят какие-то значения, которых нет в O_P, то на ребре необходимо создать дополнительный базовый блок, содержащий восстановление из памяти таких значений.