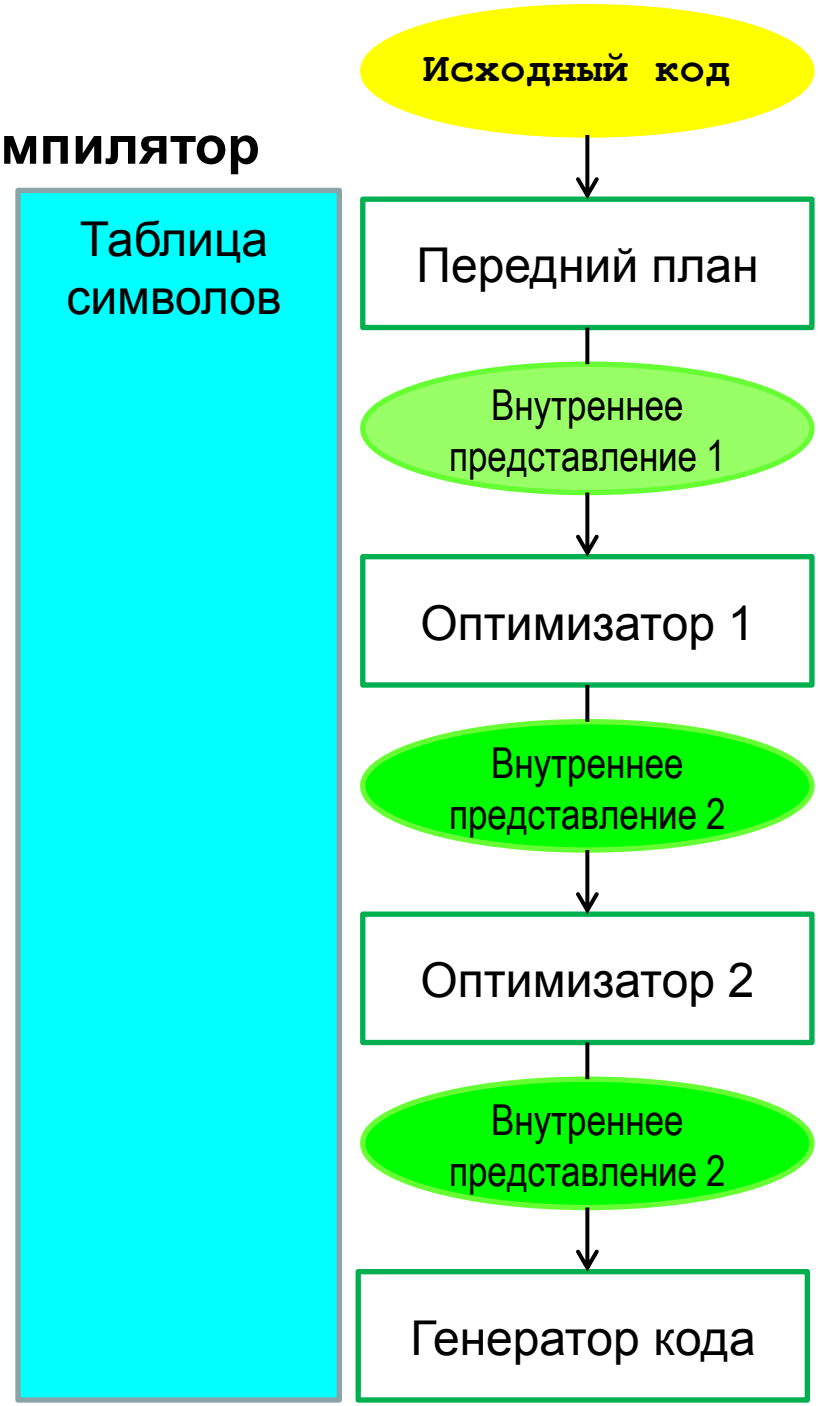
10. Заключение.

10.1 Что было рассказано

10.1.1. Что такое оптимизирующий компилятор

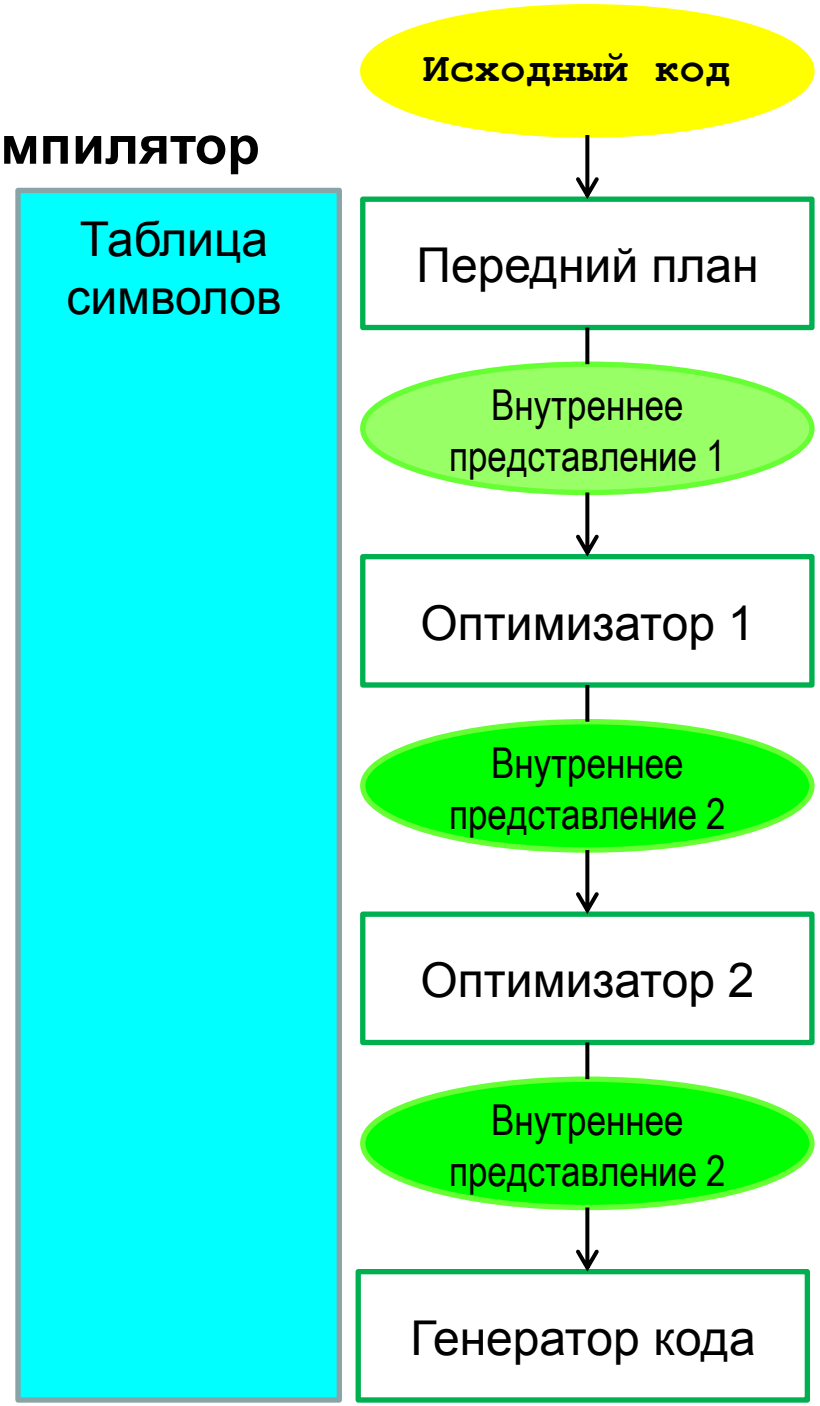
- **Оптимизатор 1: машинно-независимая оптимизация**
  - удаление *бесполезного* кода
  - удаление *избыточных* вычислений
  - удаление *мертвого* кода
  - удаление *недостижимого* кода



10.1 Что было рассказано

10.1.1. Что такое оптимизирующий компилятор

- **Оптимизатор 1: машинно-независимая оптимизация**
  - удаление *бесполезного* кода
  - удаление *избыточных* вычислений
  - удаление *мертвого* кода
  - удаление *недостижимого* кода
- **Оптимизатор 2: машинно-ориентированная оптимизация**
  - выбор команд
  - распределение регистров
  - планирование кода



10.1 Что было рассказано

10.1.1. Что такое оптимизирующий компилятор

- **Внутреннее представление 1:**
  - *инструкции* вида:

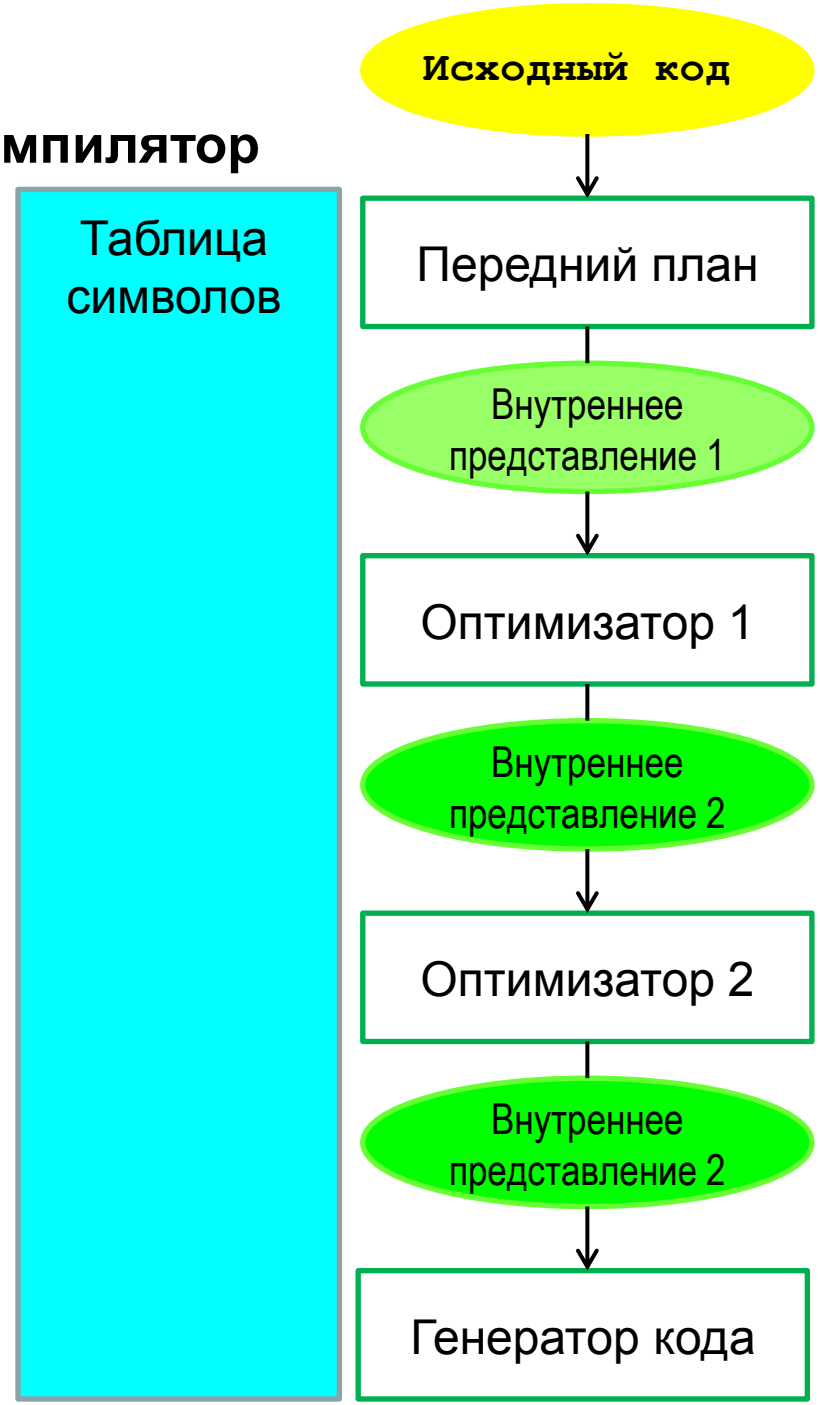
    `x ← op, y, z`

    где **`x, y, z`** – абстрактные области памяти, отведенные под соответствующие переменные

- **Внутреннее представление 2:**
  - *операции* вида:

    `OP X, Y, Z`

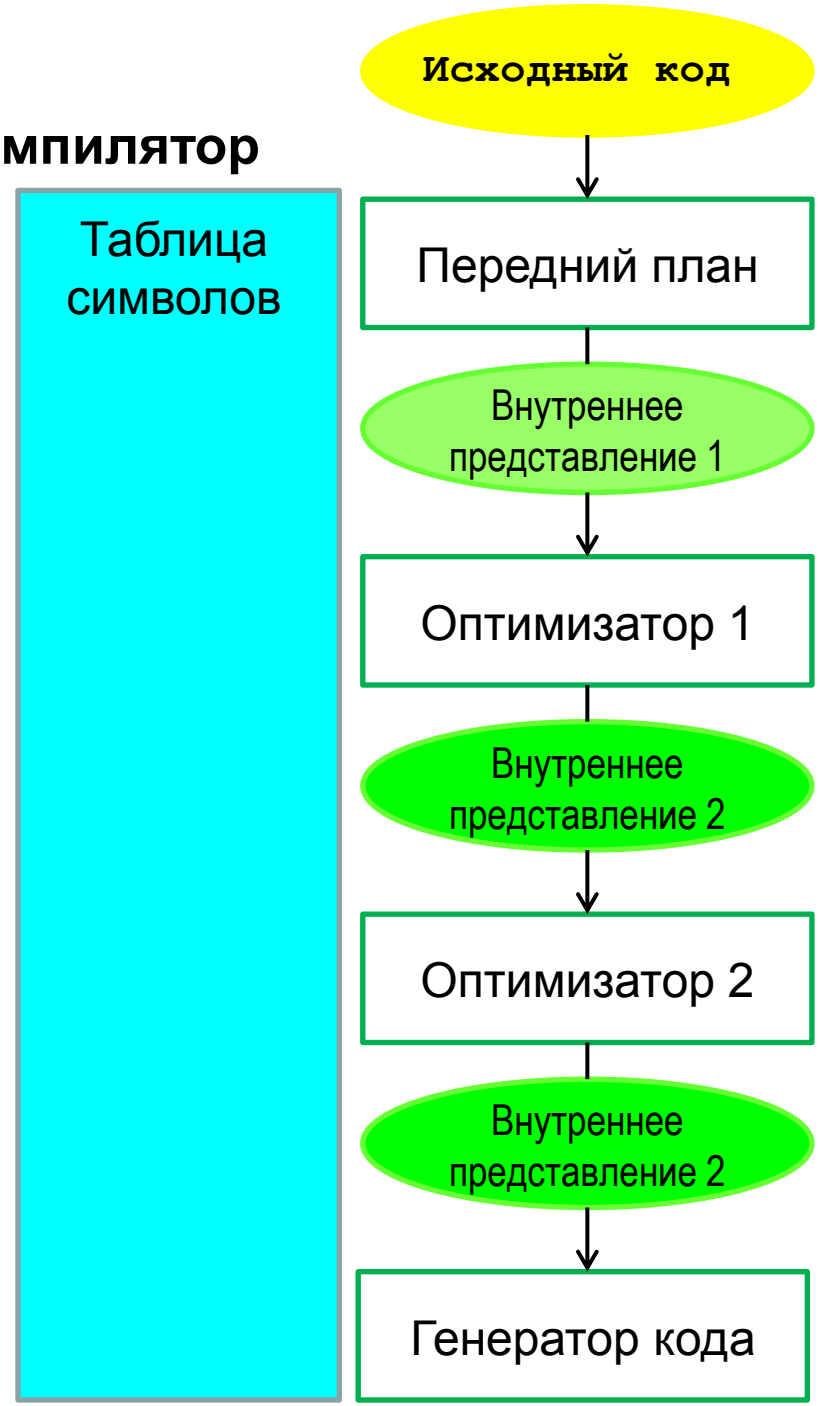
    где **`X, Y, Z`** – регистры или адреса ячеек памяти
  - операции объединяются в *команды*



10.1 Что было рассказано

10.1.1. Что такое оптимизирующий компилятор

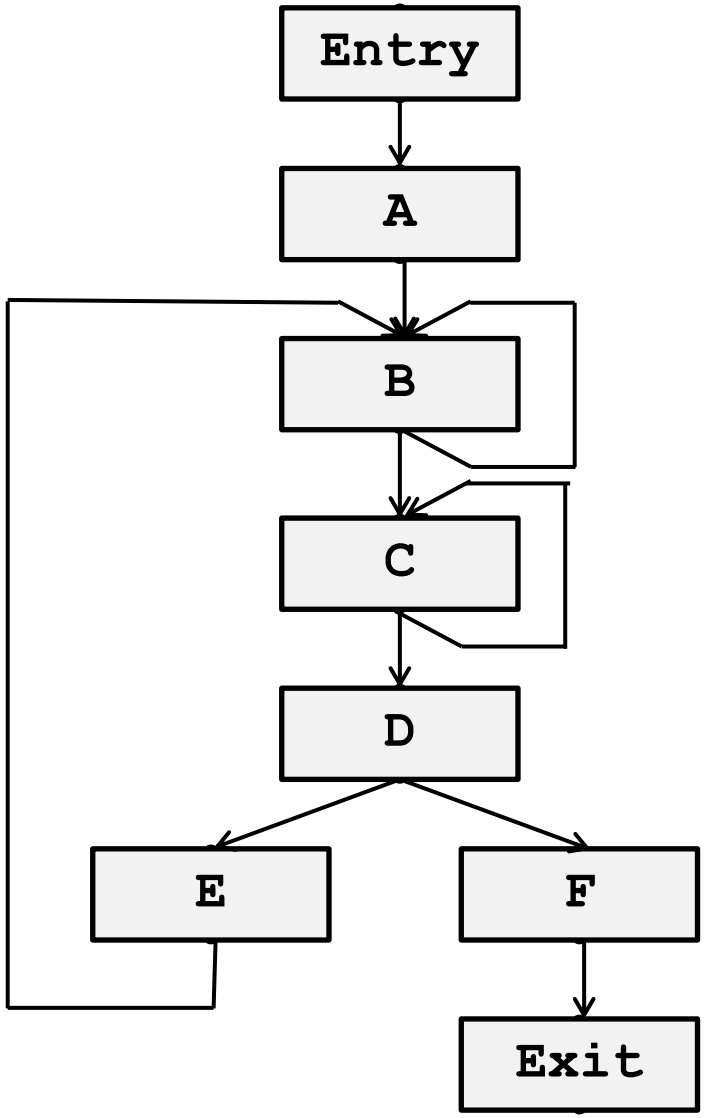
- **Оптимизирующий компилятор:**
  - много оптимизирующих преобразований, которые в некотором порядке применяются к коду компилируемой программы, повышая ее качество;
  - порядок оптимизирующих преобразований зависит от компилируемой программы, в настоящее время делаются первые попытки обеспечить автоматический подбор оптимизирующих преобразований по исходному коду программы.



10.1 Что было рассказано

10.1.2. Граф потока управления

- ◊ В программе выделяются базовые блоки (линейные участки) и строится ГПУ.
- ◊ Оптимизация в пределах одного базового блока называется *локальной*, оптимизация в пределах нескольких базовых блоков (например, суперблока) – *региональной*, оптимизация в пределах процедуры – *глобальной*
  в настоящее время рассматривается также *межпроцедурная* и даже *межмодульная* оптимизация



10.1 Что было рассказано

10.1.3. Локальная оптимизация

◊ Каждый базовый блок задается тройкой объектов:

$B = \langle P, Input, Output \rangle$

где P – последовательность инструкций блока B,

$Input$ – множество переменных, определения которых достигают блока B,

$Output$ –множество переменных, живых после блока B.

◊ Все задачи локальной оптимизации позволяет решить *метод* локальной *нумерации значений*.

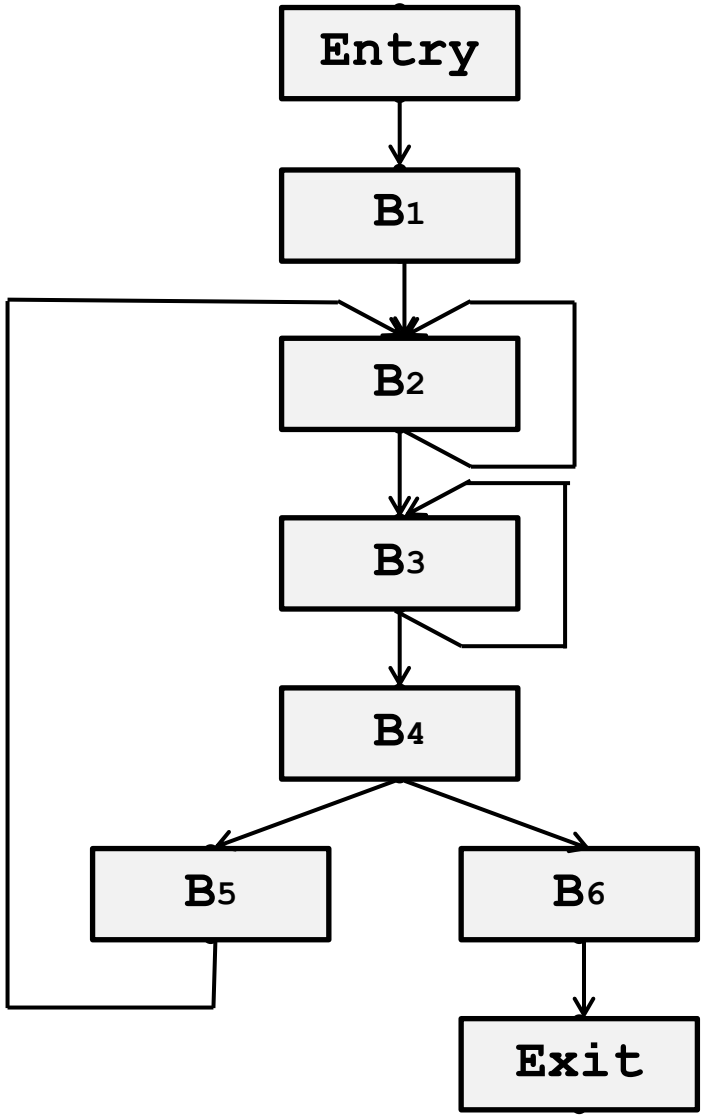
Избыточные вычисления внутри базового блока автоматически удаляются в процессе построения *ОАГ* (*ориентированного ациклического графа*).

Множества $Input$ и $Output$ позволяют фиксировать использование неинициализированных переменных и исключить присваивание значений мертвым переменным.

10.1 Что было рассказано

10.1.4. Нумерация вершин ГПУ

◊ Для *нумерации вершин ГПУ* строится остовное дерево с корнем в *Entry* и выполняется его обход «сначала в глубину», используя «обратную нумерацию» (номер присваивается при первом посещении вершины).
Нумерация определяет порядок обработки вершин ГПУ при выполнении методов итерации.
На рисунке представлен ГПУ после нумерации его вершин.



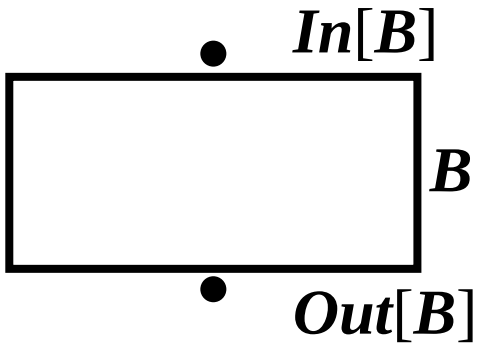
10.1 Что было рассказано

10.1.5. Поток данных

- Все преобразования программы при ее оптимизации должны сохранять *семантику программы*.
- Семантика каждого фрагмента программы (инструкции, базового блока, группы базовых блоков, процедуры, модуля и т.п.) описывается *парой состояний* процессора: состоянием во входной точке соответствующего фрагмента и состоянием в его выходной точке.
  - *входная точка* каждой *инструкции* расположена между этой инструкцией и предшествующей инструкцией; *выходная точка инструкции* расположена между ней и следующей инструкцией;

$$\ldots \cdot I_{j-1} \cdot I_j \cdot I_{j+1} \cdot \ldots$$

  - *входной точкой базового блока* является входная точка его первой инструкции; *выходной точкой базового блока* является выходная точка его последней инструкции



10.1 Что было рассказано

10.1.6. Передаточная функция

- Пара состояний во входной и выходной точках фрагмента определяет *передаточные функции* этого фрагмента:
  - *передаточная функция прямого обхода* (f) переводит состояние во входной точке в состояние в выходной точке
  - *передаточная функция обратного обхода* (f^b)переводит состояние в выходной точке в состояние во входной точке
- **Для инструкций**:

$$Out[I_j] = f_{I_j}(In[I_j])$$

$$In[I_j] = f^b_{I_j}(Out[I_j])$$

- **Для базовых блоков:**

$$f_B = f_{I_1} \circ f_{I_2} \circ \ldots \circ f_{I_n}$$

$$f^b_B = f^b_{I_n} \circ f^b_{I_{n-1}} \circ \ldots \circ f^b_{I_1}$$

10.1 Что было рассказано

10.1.7. Достигающие определения

- ◊ *Определением переменной* x называется инструкция, которая присваивает значение переменной x.
- ◊ *Использованием переменной* x является инструкция, одним из операндов которой является переменная x.
- ◊ Каждое новое определение переменной x *убивает* ее предыдущее определение.

  Определение d *достигает* точки p, если существует путь от точки, непосредственно следующей за d, к точке p, такой, что вдоль этого пути d остается живым.
- ◊ Для достигающих определений передаточная функция f_I инструкции I может быть записана в виде: $f_I(x) = gen_I \cup (x - kill_I)$

10.1 Что было рассказано

10.1.7. Достигающие определения

◊ Для базового блока B из n инструкций передаточная функция имеет вид $f_B(x) = gen_B \cup (x - kill_B)$

где $kill_B = kill_1 \cup kill_2 \cup ... \cup kill_n$

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup ...$$

$$\cup (gen_1 - kill_2 - kill_3 - ... - kill_n)$$

◊ Определения, достигающие входа в каждый базовый блок, получаются в результате решения системы уравнений

$$Out[B_i] = gen_{B_i} \cup (In[B_i] - kill_{B_i})$$

$$In[B_i] = \bigcup_{P \in Pred(B_i)} Out[P]$$

с дополнительным условием $Out[Entry] = \varnothing$ методом итераций.

◊ Множество $Input[B]$ для базового блока B – это множество $In_{RD}[B]$, которое строится при исследовании достигающих определений

10.1 Что было рассказано

10.1.8. Живые переменные

◊ Множества $Output$ для базовых блоков строятся как результат анализа, позволяющего выявить *живые переменные*, т.е. переменные, используемые в базовых блоках, в которые попадает управление после выхода из исследуемого базового блока.

◊ $Out_{LV}[B]$ – множества переменных, **живых** на выходе из блоков B получаются в результате решения системы уравнений

$$In[B] = use_B \cup (Out[B] - def_B)$$

$$Out[B] = \bigcup_{S \in Succ(B)} In[S]$$

С дополнительным условием $In\ [Exit] = \varnothing$

◊ def_B – множество переменных, определяемых в блоке B до их использования в этом блоке

◊ use_B – множество переменных, используемых в блоке B до их определения в этом блоке

10.1 Что было рассказано

10.1.9. Полурешетки

◊ *Полурешетка* – это множество L, на котором определена бинарная операция «*сбор*» $\wedge$, такая, что для всех x, y и $z \in L$:

$$x \wedge x = x \qquad (\textit{идемпотентность})$$

$$x \wedge y = y \wedge x \qquad (\textit{коммутативность})$$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z \qquad (\textit{ассоциативность})$$

Полурешетка имеет *верхний элемент* (или *верх*) $\mathsf{T} \in L$ такой, что для всех $x \in L$ выполняется $\mathsf{T} \wedge x = x$

◊ Для всех пар $x, y \in L$ определим отношение $\leq$:
$x \leq y$ тогда и только тогда, когда $x \wedge y = x$

◊ Отношение $\leq$ является отношением частичного порядка.

10.1 Что было рассказано

10.1.10. Структура потока данных

◊ *Структурой потока данных* называется четверка $\langle D, F, L, \wedge \rangle$, где

- D – направление анализа ($Forward$ или $Backward$),
- F – семейство передаточных функций,
- L – поток данных,
- $\wedge$ - реализация операции сбора.

◊ Семейство передаточных функций F называется *замкнутым*, если:

- F содержит тождественную функцию I: $\forall x \in L: \; I(x) = x$.
- F замкнуто относительно композиции:

$$\forall\, f, g \in F \Rightarrow h(x) = g(f(x)) \in F.$$

◊ Семейства передаточных функций, используемые при анализе достигающих определений и живых переменных являются замкнутыми.

10.1 Что было рассказано

10.1.10. Структура потока данных

◊ Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной*, если

$$\forall x, y \in L, \forall f \in F \quad f(x \wedge y) \leq f(x) \wedge f(y).$$

◊ Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *дистрибутивной*, если $\forall x, y \in L, \forall f \in F: \quad f(x \wedge y) = f(x) \wedge f(y)$.

◊ Семейства передаточных функций, используемые при анализе достигающих определений и живых переменных являются дистрибутивными.

10.1 Что было рассказано

10.1.11. Обобщенный итеративный алгоритм

◊ **Алгоритм.** Итеративное решение задачи анализа потока данных

- **Вход**: граф потока управления,
  структура потока данных $\langle D, F, L, \wedge \rangle$,
  передаточная функция $f_B \in F$
  константа из $l \in L$ для граничного условия
- **Выход**: значения из L для $In[B]$ и $Out[B]$ для каждого блока B в графе потока.
- **Метод**: **`if`** $(D = Forward)$ {

  $Out[Entry] = l;$

  `for each` $(B \neq Entry)\ Out[B] = \top;$

  `while` (внесены изменения в Out)

  `for` $(each\ B \neq Entry)$ {

$$In[B] = \bigwedge_{P \in Pred(B)} Out[P]$$

$$Out[B] = f_B(In[B])$$

10.1 Что было рассказано

10.1.11. Обобщенный итеративный алгоритм

◊ Обобщенный итеративный алгоритм сходится к решению уравнений потока данных, которое является максимальной фиксированной точкой системы уравнений $In[B] = \bigwedge_{P \in Pred(B)} Out[P]$

$$Out[B] = f_B(In[B])$$

т.е. представляет собой решение $\{In[B_i]^{max}, Out[B_i]^{max}\}$ этой системы, такое, что для любого другого решения $\{In[B_i], Out[B_i]\}$ выполняются условия $In\ [B_i] \le In[B_i]^{max}$ и $Out[B_i] \le Out[B_i]^{max}$

где $\le$ – полурешеточное отношение частичного порядка.

Итеративный алгоритм

(1) посещает базовые блоки не в порядке их выполнения, а в порядке обхода ГПУ (на каждой итерации каждый узел посещается только один раз)

(2) в каждой точке сбора применяет операцию сбора к значениям потока данных, полученным к этому моменту

(3) иногда в пределах итерации базовый блок B посещается до посещения его предшественников

10.1 Что было рассказано

10.1.12. Доминаторы

◊ В ГПУ вершина d является *доминатором* вершины n (этот факт записывается как $d\ dom\ n$ или $d = Dom(n)$), если любой путь от вершины $Entry$ до вершины n проходит через вершину d. Каждая вершина n является доминатором самой себя, так как путь от $Entry$ до n проходит через n.

◊ Отношение dom рефлексивно, антисимметрично и транзитивно, т.е. является отношением частичного порядка.

◊ Для любой вершины n ГПУ каждый ациклический путь от $Entry$ до n проходит через все доминаторы n, причем на всех таких путях доминаторы проходятся в *одном и том же порядке*

10.1 Что было рассказано

10.1.12. Доминаторы

◊ Вершина i ГПУ является *непосредственным доминатором* вершины n (i *idom* n), если

(1) i *dom* n

(2) не существует вершины m, $m \neq i$, $m \neq n$, такой что i *dom* m и m *dom* n.

◊ У каждой вершины n за исключением *Entry* существует единственный непосредственный доминатор.

◊ Вершина s ГПУ является *строгим доминатором* вершины n (s *sdom* n), если s *dom* n и $s \neq n$.

◊ Множество строгих доминаторов вершины n является пересечением множеств доминаторов всех ее предшественников.

10.1 Что было рассказано

10.1.12. Доминаторы

Алгоритм вычисления доминаторов

Область определения	Множество подмножеств базовых блоков
Направление обхода	$Forward$
Передаточная функция	$f_B(x) = x \cup \{B\}$
Граничное условие	$Out\,[Entry] = Entry$
Операция сбора ($\wedge$)	$\cap$
Система уравнений	$Out[B] = f_B(In[B])$ $In[B] = \bigcap_{P \in Pred(B)} Out[P]$
Начальное приближение	$Out\,[B] = N$

10.1 Что было рассказано

10.1.13. SSA-форма

◊ *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).

◊ 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.**

10.1 Что было рассказано

10.1.13. SSA-форма

◊ 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.

10.1 Что было рассказано

10.1.13. SSA-форма

◊ *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.

10.1 Что было рассказано

10.1.14. Граница доминирования

◊ Множество узлов m, удовлетворяющих условиям:

(1) n является доминатором предшественника m

$$p \in Pred(m) \;\&\; n \in Dom(p)$$

(2) n не является строгим доминатором m

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

называется *границей доминирования* n и обозначается $DF(n)$.

◊ **Неформально**: $DF(n)$ содержит все первые узлы, которые достижимы из n, на любом пути графа потока, проходящем через n, но над которыми n не доминирует.

10.1 Что было рассказано

10.1.14. Граница доминирования

◊ Алгоритм построения границы доминирования

Шаг 1. Найти все точки сбора j графа потока, т.е. все узлы j, у которых $|Pred(j)| > 1$.

Шаг 2. Исследовать каждый узел $p \in Pred(j)$ и продвинуться по дереву доминаторов, начиная с p и вплоть до непосредственного доминатора j: при этом j входит в состав границы доминирования каждого из пройденных узлов, за исключением непосредственного доминатора j.

10.1 Что было рассказано

10.1.15. 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*.

◊ Для этого вычисляется множество глобальных имен $Globals$. При этом используются результаты анализа живых переменных.

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

10.1 Что было рассказано

10.1.15. 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** $Globals = Globals \cup \{z\}$;
            $def_B = def_B \cup \{x\}$;
            $Blocks(x) = Blocks(x) \cup \{B\}$
        }
    }

10.1 Что было рассказано

10.1.15. Размещение ϕ-функций

Алгоритм размещения ϕ-функций

- **Вход**: исходный граф потока
- **Выход**: преобразованный граф потока
- **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};
    }
}
```

10.1 Что было рассказано

10.1.16. Алгоритм переименования переменных

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

◊ **Выход**: программа, в которой переменным сопоставлены их *SSA-имена*

◊ **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.

10.1 Что было рассказано

10.1.16. Алгоритм переименования переменных

- Основной алгоритм:

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

- Функция **NewName**:

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

10.1 Что было рассказано

10.1.16. Алгоритм переименования 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]);
```

10.1 Что было рассказано

10.1.17. Восстановление кода из SSA-формы

◊ 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:

10.1 Что было рассказано

10.1.18. Нумерация значений в суперблоках

◊ *Расширенный базовый блок* или *суперблок* E – это множество базовых блоков $B_1, B_2, \ldots, B_n$, где у блока B_1 может быть несколько предшественников (они не входят в состав суперблока), а каждый из блоков B_i, $2 \le i \le n$ имеет в суперблоке единственного предшественника.

Блоки $B_i \in E$ формируют дерево с корнем B_1.

У суперблока E может быть несколько выходов на блоки (или суперблоки), не входящие в состав E.

◊ Алгоритм нумерации значений в суперблоках будем называть *расширенным алгоритмом локальной нумерации значений*.

◊ Используется стек (обход дерева блоков)

10.1 Что было рассказано

10.1.19. Рекурсивный алгоритм глобальной нумерации значений

◊**Вход**: (1) граф потока управления $\langle N, E \rangle$, дерево доминаторов DT

(2) множество Val значений переменных, констант и выражений

◊**Выход**: отображение VN: $Val \to N \cup \{0\}$ (N – множество натуральных чисел), ставящее в соответствие каждому значению его номер: натуральное число или 0.

◊**Метод**: **`procedure DBGVN(Block B)`**

||Обработка ϕ-функций

Отметить начало новой области имен

`for each p ∈ B`, где p – ϕ-функция вида "$n \leftarrow \phi(\ldots)$"

`if p` бессмысленна или избыточна

поместить номер значения **`p`** в **`VN[n]`**

удалить **`p`**

`else VN[n] ← n`;

добавить **`p`** в ТЗ

10.1 Что было рассказано

10.1.19. Рекурсивный алгоритм глобальной нумерации значений

(2) Обработка остальных инструкций

`for each a∈B` где a – присваивание вида “$x \leftarrow op, y, z$”

заменить y на $VN[y]$ и z на $VN[z]$

$expr \leftarrow op, y, z$ || $expr$ – вход в ТЗ

`if` $expr$ может быть упрощено до $expr'$

Заменить a на “$x \leftarrow expr'$”

$expr \leftarrow expr'$

`if` $expr$ имеется в ТЗ с номером v

$VN[x] \leftarrow v$

Удалить a

`else` Добавить $expr$ в ТЗ с номером x $VN[x] \leftarrow x$

10.1 Что было рассказано

10.1.19. Рекурсивный алгоритм глобальной нумерации значений

(2) Обработка остальных инструкций

`for each a∈B` где a – присваивание вида “$x \leftarrow op, y, z$”

заменить y на $VN[y]$ и z на $VN[z]$

$expr \leftarrow op, y, z$ || $expr$ – вход в ТЗ

`if` $expr$ может быть упрощено до $expr'$

Заменить a на “$x \leftarrow expr'$”

$expr \leftarrow expr'$

`if` $expr$ имеется в ТЗ с номером v

$VN[x] \leftarrow v$

Удалить a

`else` Добавить $expr$ в ТЗ с номером x $VN[x] \leftarrow x$

10.1 Что было рассказано

10.1.19. Рекурсивный алгоритм глобальной нумерации значений

(3) Окончание обработки блока B и переход к обработке его дочерних блоков (по дереву доминаторов)

`for each` $s \in Succ(B)$

скорректировать входы ϕ-функций в s

`for each` дочернего блока c узла B по дереву доминаторов

$DBGVN(c)$ || рекурсивный вызов

Очистить ТЗ при выходе из области (стэк)

10.1 Что было рассказано

10.1.20 Распространение копий

◊ Пусть в оптимизируемой процедуре есть инструкция копирования

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

Распространение копий означает замену всех последующих вхождений переменной **x** на переменную **y**.

◊ Рассмотрим множество всех команд копирования анализируемой процедуры. Каждая команда копирования описывается четверкой $\langle \mathbf{x}, \mathbf{y}, b, p \rangle$, где **x** и **y** представляют инструкцию копирования $\mathbf{x} \leftarrow \mathbf{y}$, находящуюся в строке p базового блока b.

Множество всех таких четверок обозначим через U. Множество U содержит все инструкции копирования анализируемой процедуры.

10.1 Что было рассказано

10.1.20 Распространение копий

- Система уравнений составляется по аналогии с системой уравнений для достигающих определений:
  - сначала из множества инструкций копирования удаляются инструкции «убитые» в блоке b, потом в него добавляются инструкции копирования блока b.
  - Второе уравнение содержит операцию пересечения, так как по всем путям должны приходить одинаковые копии.

$$Out_{CP}(b) = copy(b) \cup (In_{CP}(b) - kill(b))$$

$$In_{CP}(b) = \bigcap_{p \in Pred(b)} Out_{CP}(p)$$

10.1 Что было рассказано

10.1.21 Оптимизация циклов

Построение натурального цикла по обратной дуге

Вход: ГПУ $G = \langle N, E \rangle$ с входным узлом $Entry$.

Обратная дуга $e = \langle n, d \rangle \in E$

Выход: подграф $C \subseteq G$, являющийся натуральным циклом.

Метод:

(1) начальное значение C – множество $\{n, d\}$.

(2) узел d помечается как «посещенный».

(3) начиная с узла n выполняется поиск в глубину на обратном графе потока (направления дуг заменены на противоположные).

(4) все узлы, посещенные на шаге (3), добавляются в C.

10.1 Что было рассказано

10.1.21 Оптимизация циклов

Перемещение кода, инвариантного относительно цикла

◊ Инструкция *инвариантна относительно цикла*, если она удовлетворяет одному из следующих условий:

- ее операнды – константы
- все определения операндов, достигающие инструкции находятся вне цикла
- внутри цикла имеется в точности одно определение операнда, но оно само инвариантно относительно цикла.

◊ **Алгоритм**:

1. Перед заголовком цикла вставить пустой базовый блок (будущий предзаголовок).

Для всех инструкций в теле цикла:

2. Отметить как инвариантные все операнды-константы
3. Отметить как инвариантные все операнды, у которых все определения, достигающие инструкции, находятся вне цикла
4. Отметить как инвариантные все инструкции, все операнды которых отмечены
5. Повторять шаги 2 – 4, пока инвариантные инструкции не перестанут выделяться
6. Переместить все выделенные инструкции в предзаголовок.

10.1 Что было рассказано

10.1.21 Исключение бесполезного кода

- Программа может содержать *бесполезный код* – инструкции, не влияющие на результат вычислений. Как правило, бесполезный код появляется в программе в результате работы некоторых алгоритмов анализа и оптимизации, реализованных в компиляторе.
- Существует несколько разновидностей бесполезного кода:
  - *Мертвый код* – инструкции, результат которых не используется в дальнейших вычислениях.
  - *Недостижимый код* – инструкции, которые не содержатся ни в одном реальном пути выполнения.
  - *Избыточный код* – инструкции, повторно вычисляющие уже вычисленные значения (например, доступные выражения или инвариантные вычисления в циклах).
- Требуется обнаружить и удалить бесполезный (в частности, недостижимый и мертвый) код

10.1 Что было рассказано

10.1.21 Исключение бесполезного кода

Алгоритм *Mark & Sweep*.

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

10.1 Что было рассказано

10.1.22 Генерация кода

Основные фазы генерации кода

- ◊ Выбор команд
- ◊ Распределение регистров
- ◊ Выбор оптимального порядка команд (планирование кода)

Выбор команд

- ◊ Каждая команда целевого языка имеет стоимость.
- ◊ Стоимость команды равна единице плюс сумме стоимостей, связанных с режимами адресации операндов:
  - ◈ Стоимость операнда на регистре равна 0
  - ◈ Стоимость операнда из ячейки памяти равна 1
  - ◈ Стоимость операнда-константы равна 1
  - ◈ Стоимость каждого разыменования равна 1
- ◊ **Стоимость программы** (на целевом языке) равна сумме стоимостей ее команд. Чем меньше стоимость программы, тем быстрее она выполняется
- ◊ Алгоритм генерации кода должен минимизировать стоимость программы.

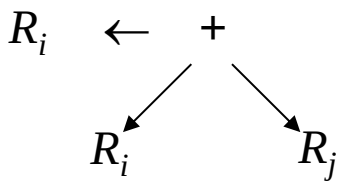
10.1 Что было рассказано

10.1.22 Генерация кода

Схема трансляции дерева

- Схема трансляции задается набором *правил свертки*
- Каждое правило свертки содержит:
  - метку узла, на который заменяется поддерево при свертке
  - поддерево, соответствующее рассматриваемому шаблону
  - команды, которые помещаются в объектную программу при свертке

Пример правила: правило для команды сложения одного регистра с другим имеет вид:




```
{ADD Ri, Ri, Rj}
```

если входное дерево содержит поддерево, соответствующее данному шаблону, то это поддерево можно заменить одним узлом с меткой Ri и сгенерировать команду **`ADD Ri, Ri, Rj`**.

Такая замена называется *замещением поддерева*.

В случае использования *обобщенных шаблонов* для выбора команд в частных случаях могут использоваться семантические проверки

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров

- *Распределение регистров*

  отображает неограниченное множество имен (псевдорегистров) на конечное множество физических регистров целевой машины, 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).

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров

Локальное распределение регистров

- ◊ *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

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров
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**

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров
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 $R_x = R_y$.

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров

Глобальное распределение регистров

- ◊ 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, 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.

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров
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.

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров

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.

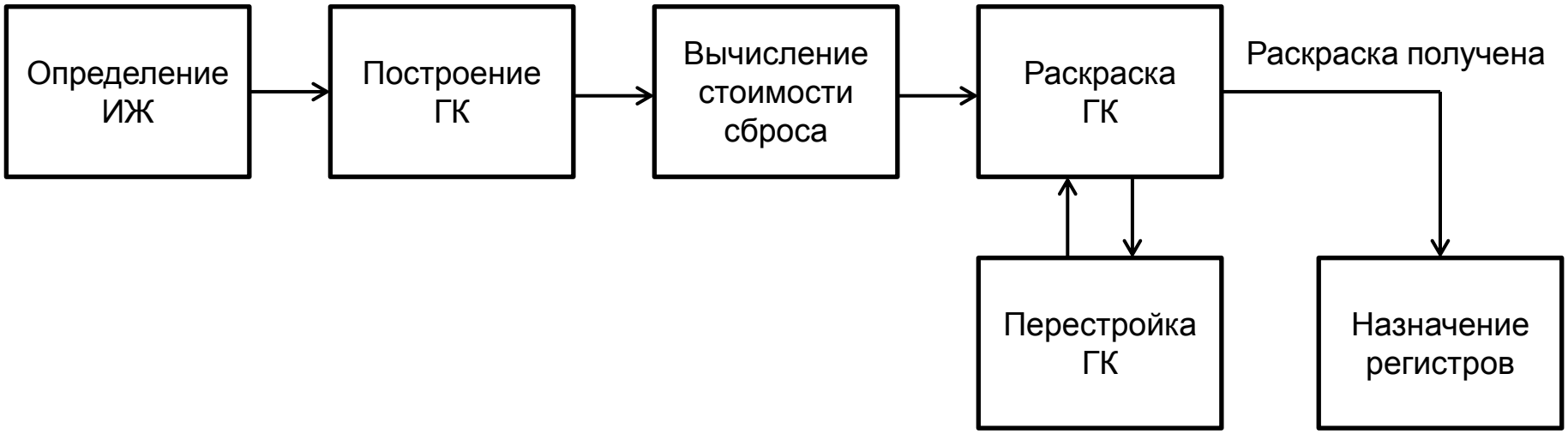
◊ ***3 фаза. Проверка на окончание процесса раскраски***

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), после чего ГК перестраивается с учетом слива и/или разделенных узлов. После перестройки ГК делается переход на первую фазу.

◊ Ключевым моментом является порядок в котором узлы ГК помещаются в стек.

10.1 Что было рассказано

10.1.23 Распределение и назначение регистров
Алгоритм раскраски графа конфликтов



- 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.

10.1 Что было рассказано

10.1.24 Планирование кода

- ◊ Цель планирования кода – выбрать такую последовательность команд, которая, не меняя семантики программы, обеспечит оптимальное использование особенностей архитектуры целевого процессора
- ◊ Прежде всего, это необходимость обеспечить правильное использование возможностей параллельного выполнения команд, реализованных в аппаратуре данного целевого процессора

10.1 Что было рассказано

10.1.24 Планирование кода

- Требование сохранения семантики программы проще всего выразить в форме ограничений, которым должна удовлетворять целевая программа. Эти ограничения должны гарантировать, что оптимизированная программа будет давать такие же результаты, что и исходная.
- На планирование кода накладывается следующие три типа ограничений:
  - *Ограничения управления*. Все операции, выполняемые в исходной программе, должны выполняться и в оптимизированной программе.
  - *Ограничения данных.* Операции в оптимизированной программе должны выдать те же результаты, что и соответствующие операции в исходной программе
  - *Ограничения ресурсов.* Планирование кода не должно требовать чрезмерного количества ресурсов машины.

10.1 Что было рассказано

10.1.24 Планирование кода

Зависимости по данным

- **Определение**. Две команды называются *зависимыми по данным*, если изменение порядка их выполнения может привести к изменению результата вычислений, выполняемых программой.
- Виды зависимостей по данным:
  - *Истинная зависимость*: **чтение после записи**.
    Если команда C_1 записывает значение в некоторую ячейку памяти (или на регистр), а команда C_2 считывает это значение, то команды C_1 и C_2 зависимы.
  - *Антизависимость*: **запись после чтения**.
    Если команда C_1 считывает значение из некоторой ячейки памяти, а команда C_2 записывает в эту ячейку новое значение, то команды C_1 и C_2 зависимы.
  - *Зависимость по выходу*: **запись после записи**.
    Если команды C_1 и C_2 записывают значения в одну и ту же ячейку памяти, то команды C_1 и C_2 зависимы.

10.1 Что было рассказано

10.1.24 Планирование кода

Алгоритм планирования с помощью списков включает следующие четыре шага

◊ 1. *Переименование значений, чтобы избежать антизависимостей*.
Значения переименовываются таким образом, чтобы каждое значение имело уникальное имя. Это нужно для того, чтобы найти и те расписания, которые были бы исключены из-за антизависимостей.

2. *Построение графа зависимостей D*.
Базовый блок обходится снизу вверх. Для каждой операции строится вершина графа, представляющая значение, вычисленное этой операцией. Построенная вершина соединяется дугами с вершинами, использующими это значение.

10.1 Что было рассказано

10.1.24 Планирование кода

Алгоритм планирования с помощью списков включает следующие четыре шага

◊ 3. *Присваивание приоритета каждой операции.*
Планировщик использует эти приоритеты при выборе операций. Каждая вершина снабжается несколькими оценками, используемыми планировщиком для определения очередной операции, которая должна быть запланирована, и для выбора операций, когда основная оценка у них одинакова. Известно несколько различных схем назначения приоритетов. Классическая схема использует в качестве основного приоритета продолжительность вычисления узла с учетом задержек.

4. *Итеративный процесс выбора вершин и планирования их*.
Алгоритм устанавливает счетчик тактов в 1 и пытается запланировать как можно больше операций. Потом увеличивает счетчик тактов и переходит к планированию следующего такта. И так далее пока все операции не будут запланированы.

10.1 Что было рассказано

10.1.24 Планирование кода

Алгоритм планирования с помощью списков

- ◊ Когда алгоритм должен выбирать из нескольких готовых операций, все зависит от качества алгоритма выбора. Алгоритм выбирает операцию с наивысшим приоритетом. Если приоритеты равны требуется рассмотреть дополнительные критерии.
- ◊ Операции обмена с памятью часто имеют неопределенные и непостоянные задержки: этому способствуют промахи кэшей и прочие особенности работы с памятью. Фактическая задержка при этом может меняться от 0 до нескольких тысяч тактов. Если планировщик исходит из наихудшей возможности, он рискует получить достаточно длительные простои процессора, если из наилучшей, он спровоцирует дополнительные промахи кэша.
- ◊ Наиболее привлекательным представляется алгоритм *сбалансированного планирования*, когда задержка оценивается в процессе планирования в зависимости от контекста (а также, возможно, характеристик целевого процессора).

10.1 Что было рассказано

10.1.24 Планирование кода

Глобальное планирование кода
Зависимости по управлению

◊ Команда **`s1`** *зависит по управлению* от команды **`s2`**, если выполнение команды **`s2`** определяет, будет ли выполняться команда **`s1`**.

Простые примеры

- в конструкции **`if(c) s1; else s2;`**
  `s1` и **`s2`** зависят по управлению от **`c`**
- в конструкции **`while (c) s;`**
  `s` зависит по управлению от **`c`**

10.1 Что было рассказано

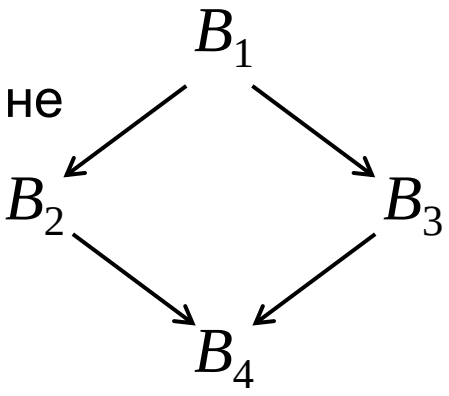
10.1.24 Планирование кода

Глобальное планирование кода
Эквивалентность по управлению

◊ **Определение**. Если базовый блок B является доминатором базового блока B', а блок B' - постдоминатором блока B, то блоки B и B' называются *эквивалентными по управлению*, что означает, что блок B выполняется тогда и только тогда, когда выполняется блок B'.

◊ Возможны следующие отношения между блоками:

- ◈ Блоки B_1 и B_4 эквивалентны по управлению: $B_1 = Dom(B_4)\ \&\ B_4 = Postdom(B_1)$
- ◈ Блок B_1 доминирует над блоком B_2, а блок B_2 не постдоминирует над блоком B_1.
- ◈ Блок B_2 не доминирует над блоком B_4, а блок B_4 постдоминирует над блоком B_2.
- ◈ Блоки B_2 и B_3 не связаны ни отношением доминирования, ни отношением постдоминирования.



10.1 Что было рассказано

10.1.24 Планирование кода

Взаимодействие с динамическим планировщиком

- ◊ Динамический планировщик – это, как правило, аппаратный планировщик *OoO*-процессора
- ◊ Динамический планировщик обладает тем преимуществом, что он может создавать новые планы в зависимости от условий времени выполнения, и не рассматривать заблаговременно все возможные планы.
- ◊ Если целевой процессор оснащен динамическим планировщиком (принадлежит классу *OoO*), основная функция статического планировщика состоит в обеспечении ранней выборки команд с большими задержками, чтобы динамический планировщик мог запланировать как можно более раннее выполнение этих команд. Это особенно эффективно в том случае, когда в целевом процессоре имеются команды предвыборки данных.
- ◊ В частности, предвыборка данных помогает предотвратить промахи кэша, представляющие собой класс непредсказуемых событий, которые могут привести к большим разбросам производительности программы.

10.2 Что не было рассказано

10.2.1 Машинно-независимая оптимизация

- Распространение констант (не дистрибутивная, а всего лишь монотонная структура анализа потока данных)
- Устранение частичной избыточности (взаимодействие четырех структур анализа потока данных)
- Ускорение итераций путем объединения базовых блоков в более крупные области (наподобие суперблоков)
- Анализ указателей – пример межпроцедурного анализа
- Контекстно-зависимый и трассо-зависимый анализ потока данных
- Дополнительные оптимизации циклов:
  - раскрутка циклов
  - выявление и преобразование индуктивных переменных
- Открытая вставка процедур

10.2 Что не было рассказано

10.2.2 Машинно-ориентированная оптимизация

- ◊ Конвейеризация вычислений при планировании кода
- ◊ Векторизация циклов (при наличии векторных ФУ)
- ◊ Приватизация данных при параллельном выполнении циклов в различных потоках над общей памятью
- ◊ Методы обеспечения локальности данных
- ◊ Агрессивные методы глобального планирования кода

10.2.3 Динамическая оптимизация в JIT-компиляторах

- ◊ Структура JIT-компилятора языка Java
- ◊ Регистровая Java-машина: Dalvik для ОС Android (Google)
- ◊ JIT-компилятор для C/C++ (LLVM)
- ◊ Некоторые методы динамической оптимизации

10.3 Как будет проходить экзамен

10.3.1 Выписка из расписания экзаменов

420	421	423	424	425	427	428	
							16.04
							17.04
							18.04
							19.04
КК	КК	КК					20.04
			КК	КК	КК	КК	21.04
							22.04
							23.04
							24.04
							25.04
							26.04
ЧМ	ЧМ	ЧМ					27.04
			ЧМ	ЧМ	ЧМ	ЧМ	28.04
							29.04
							30.04
							01.05
							02.05
							03.05
							04.05
							05.05
20	19	14	10	13	12	17	
420	**421**	**423**	**424**	**425**	**427**	**428**	
Конструирование компиляторов Численные методы							

ауд 105

10.3 Как будет проходить экзамен

1. Если студент выполнил все четыре домашних задания, он получает «отлично», не сдавая экзамена.
2. Если студент выполнил любые три домашних задания, он получает «хорошо», не сдавая экзамена, но может взять дополнительный вопрос, чтобы получить «отлично».
3. Чтобы получить билет с вопросом, студент должен ответить на предварительные вопросы (от трех до пяти) по определениям (все эти определения сформулированы в заключительной лекции).
   - 3а. Если студент не сможет ответить на предварительные вопросы, он получит оценку «неудовлетворительно» и будет сдавать экзамен повторно
   - 3б. Если студент ответит на предварительные вопросы, он может отказаться брать билет и получить оценку «удовлетворительно».
   - 3в. Если студент ответит на предварительные вопросы, но не сможет ответить на вопрос билета, он получит оценку «удовлетворительно».

10.3 Как будет проходить экзамен

Вопросы к экзамену (билет будет содержать один вопрос)

1. Внутреннее представление программы, применяемое на оптимизирующих фазах.
2. Граф потока управления и его построение.
3. Локальная оптимизация методом нумерации значений. Ориентированный ациклический граф.
4. Глобальная оптимизация с помощью анализа потока данных. Анализ достигающих определений. Построение множеств входных данных для базовых блоков.
5. Глобальная оптимизация с помощью анализа потока данных. Анализ живых переменных. Построение множеств выходных данных для базовых блоков.
6. Глобальная оптимизация с помощью анализа потока данных. Анализ доступных выражений.
7. Полурешетки и их свойства. Полурешеточная операция (сбор), полурешеточное отношение частичного порядка и его связь с полурешеточной операцией. Диаграммы полурешеток.
8. Структура потока данных и ее свойства. Монотонные структуры. Дистрибутивные структуры.
9. Обобщенный итеративный алгоритм. Сходимость алгоритма к решению. Отношение решения, полученного с помощью обобщенного итеративного алгоритма, к точному (идеальному) решению.

10.3 Как будет проходить экзамен

Вопросы к экзамену (билет будет содержать один вопрос)

10. SSA-форма внутреннего представление программы. Алгоритм вычисления доминаторов. Построение дерева доминаторов. Построение максимальной SSA-формы.
11. Построение частично усеченной SSA-формы. Алгоритм нахождения границ доминирования.
12. Построение частично усеченной SSA-формы. Размещение φ-функций.
13. Построение частично усеченной SSA-формы. Переименование переменных.
14. Восстановление кода из SSA-формы.
15. Нумерация значений в суперблоках. Глобальная нумерация значений.
16. Простые оптимизации: сворачивание констант, алгебраические упрощения и перегруппировка, распространение копий.
17. Оптимизация циклов. Построение натурального цикла по обратной дуге.
18. Оптимизация циклов. Перемещение кода, инвариантного относительно цикла.
19. Исключение бесполезного кода. Исключение недостижимого кода. Оптимизация потока управления.
20. Машинно-ориентированная оптимизация. Генерация оптимального кода для выражений.

10.3 Как будет проходить экзамен

Вопросы к экзамену (билет будет содержать один вопрос)

21. Генерация кода с помощью алгоритма динамического программирования
22. Выбор команд методом переписывания дерева.
23. Распределение регистров методом раскраски графа конфликтов.
24. Планирование команд в базовом блоке методом переписывания дерева.
25. Глобальное планирование команд.