Конспект по компиляторам к госэкзамену

В программе госэкзамена есть следующие три вопроса по компиляторам:

Вопрос 25. Промежуточные представления программы: абстрактное синтаксическое дерево; последовательность трехадресных инструкций. Базовые блоки и граф потока управления.

Вопрос 26. Локальная оптимизация при компиляции программы. Ориентированный ациклический граф и метод нумерации значений.

Вопрос 27. Глобальная оптимизация при компиляции программы. Построение передаточных функций базовых блоков. Монотонные и дистрибутивные передаточные функции. Метод неподвижной точки и его применение для нахождения достигающих определений.

В данном конспекте рассматриваются эти вопросы и указываются ссылки на соответствующую литературу.

1. Промежуточные представления программы

Современный компилятор содержит следующие три фазы (рис. 1):

1. Фаза анализа исходного кода (ее часто называют передним планом – *Frontend*).

2. Фаза оптимизации компилируемой программы.

3. Фаза генерации целевого кода (ее часто называют задним планом – *Backend*).

Передний план компилятора анализирует исходный код компилируемой программы и строит *промежуточное представление верхнего уровня* (ППВУ). В неоптимизирующем компиляторе фаза оптимизации отсутствует, так что сразу после переднего плана начинает работать задний план, который переводит ППВУ в промежуточное представление нижнего уровня (ППНУ), а затем и в код целевого компьютера. Для этого обычно используется метод переписывания дерева. Конструирование неоптимизирующих компиляторов рассматривается в учебном пособии [1], доступном через интернет в формате pdf.

Передний план работает следующим образом.

Компилируемая (исходная) программа поступает в компилятор в виде строки символов. Анализ осуществляется тремя программами-автоматами: анализатором лексики, анализатором синтаксиса и анализатором статической семантики. Процессом построения дерева разбора управляет анализатор синтаксиса (*парсер*).

Он обращается к анализатору лексики (*сканеру*) за очередным токеном, строя дерево разбора. Впоследствии дерево разбора преобразуется в абстрактное

Рис. 1. Архитектура оптимизирующего компилятора

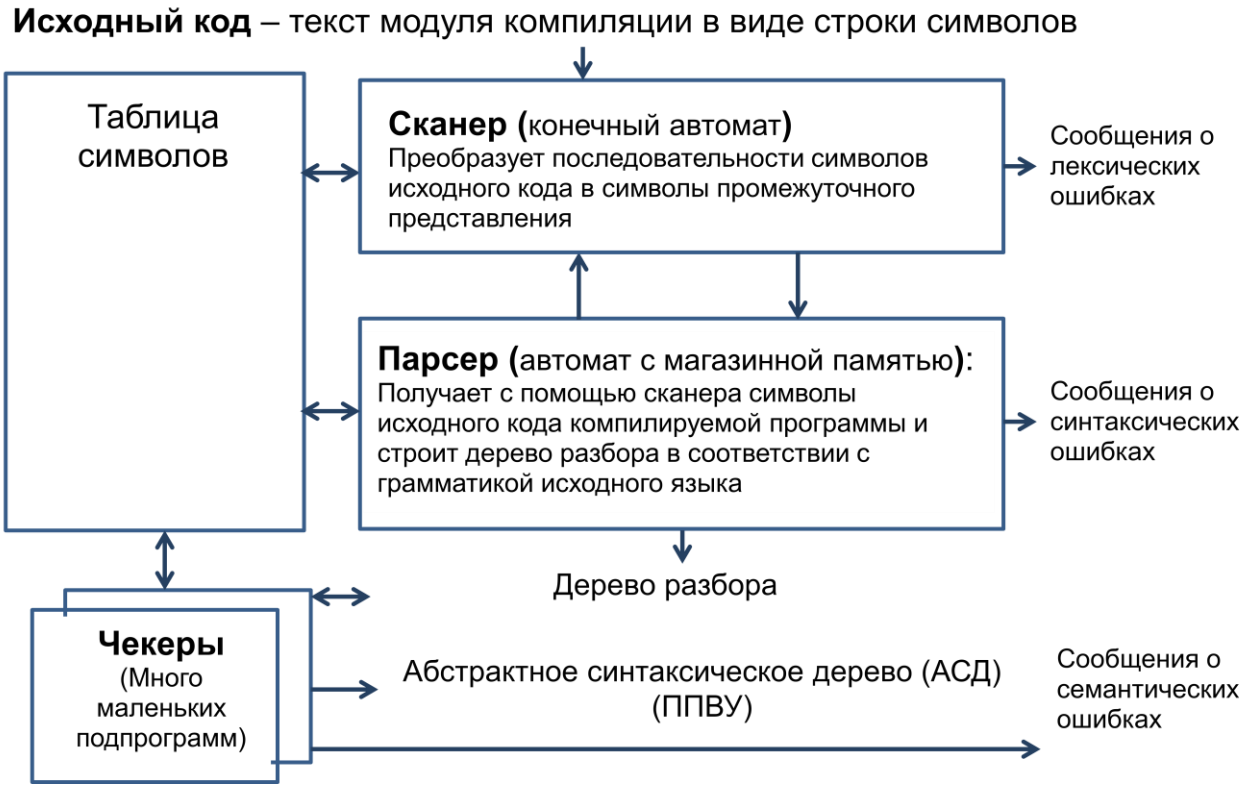
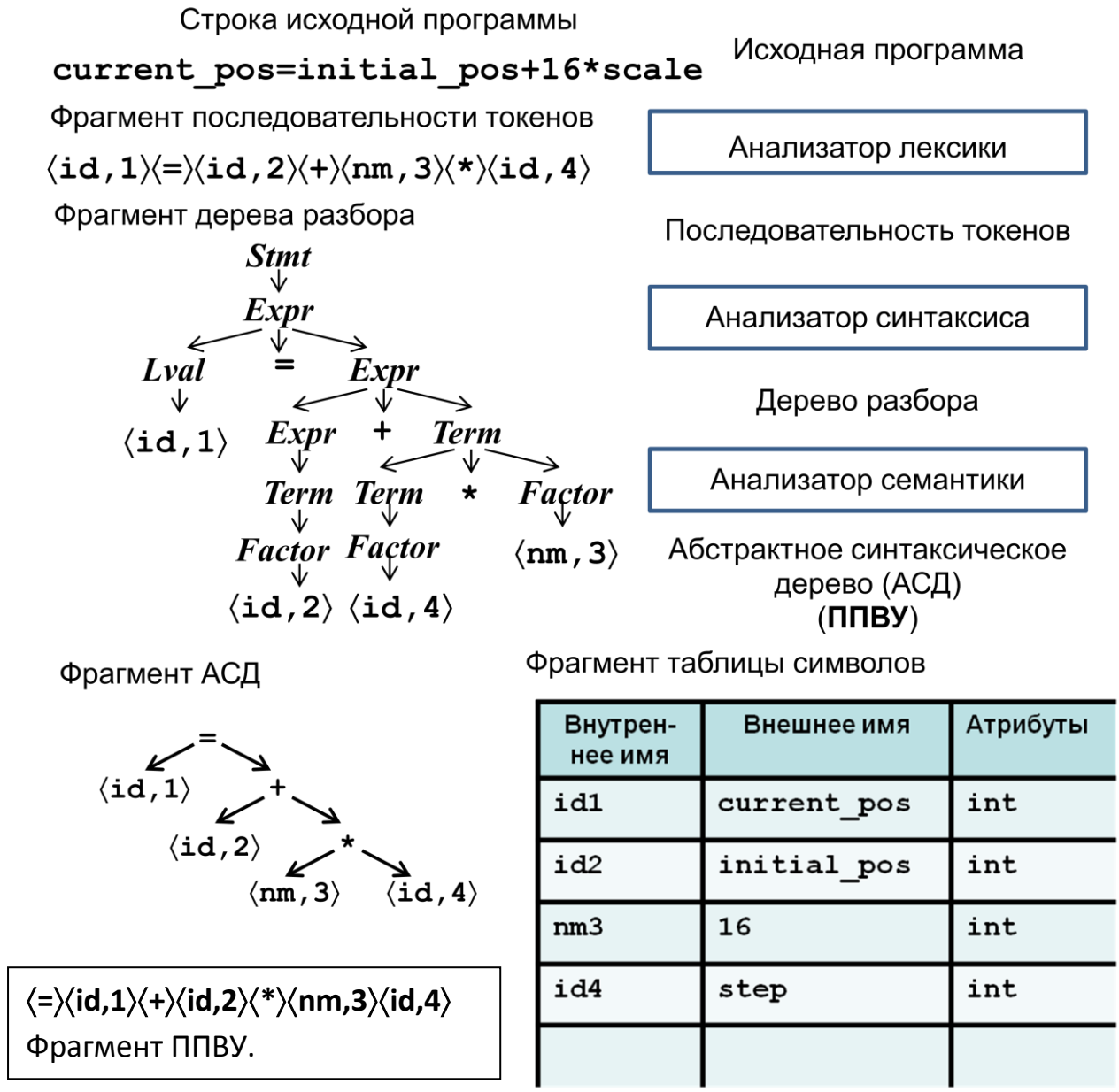


Рис.2. Синтаксически управляемая трансляция.

синтаксическое дерево (АСД). Внутренним представлением высокого уровня обычно является линейная бесскобочная запись АСД (рис. 3).



Внутрен-нее имя	Внешнее имя	Атрибуты
id1	current_pos	int
id2	initial_pos	int
nm3	16	int
id4	step	int

Рис.3. Синтаксически управляемая трансляция. Пример.

Полученное представление удобно для генерации целевой программы, но неудобно для оптимизации программы. Поэтому фаза оптимизации начинается с перевода ППВУ в промежуточное представление среднего уровня (ППСУ).

Программа на ППСУ представляет собой последовательность *инструкций* – «четверок» вида

res ← op, opd1, opd2

где **opd1** и **opd2** – идентификаторы абстрактных областей памяти, в которых размещены операнды, **op** – код операции, **res** – идентификатор абстрактной области памяти, в которой должен быть размещен результат операции **op**. Таким образом, программа на ППСУ похожа на программу на языке ассемблера. Далее следует краткое описание учебного ППСУ как языка программирования.

Инструкции присваивания	(**x**, **y** и **z** – имена (адреса) переменных или константы):
x ← op, y, z	выполнить бинарную операцию **op** над операндами **y** и **z** и поместить результат в **x**
x ← op, y	выполнить унарную операцию **op** над операндом **y** и поместить результат в **x**
x ← y	скопировать значение переменной **y** в переменную **x**
x[i] ← y	поместить значение **y** в **i**-ю по отношению к **x** ячейку памяти
x ← y[i]	поместить значение **i**-ой по отношению к **y** ячейке памяти в **x**

Не хватает указателей и адресной арифметики (т.н. *L*-выражений в терминологии языка *C*): **x[i]** – простейший частный случай *L*-выражения.

Инструкции перехода:	
goto L	Безусловный переход: следующей будет выполнена инструкция с меткой L
ifTrue x goto L	Условный переход: если x истинно, следующей будет выполнена инструкция с меткой L, иначе следующая инструкция
ifFalse x goto L	Условный переход: если x ложно, следующей будет выполнена инструкция с меткой L иначе следующая инструкция

На рис. 4 – схема перевода на ППСУ оператора **if (expr) stmt1;**

По аналогии можно составить схемы перевода в ППСУ операторов **else**, **while**, **for**, **switch** и других операторов управления.

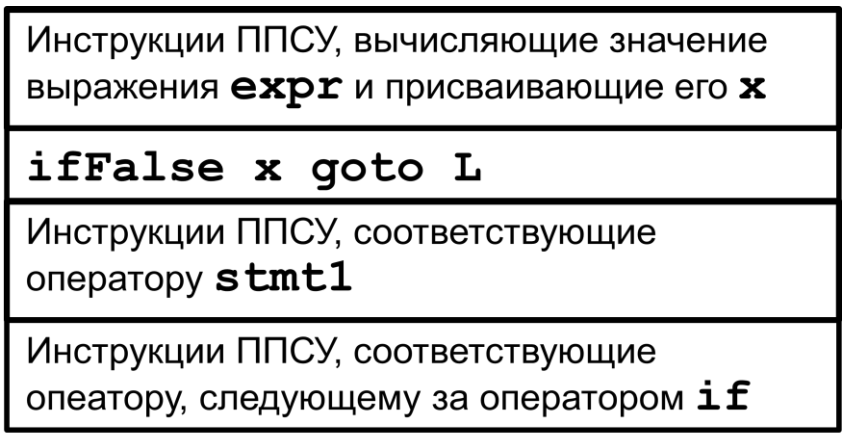


Рис. 4. Схема перевода в ППСУ оператора **`if (expr) stmt1;`**

Вызов процедуры:	
`param x`	Передача фактического параметра вызываемой процедуре (если вызываемая процедура имеет *n* параметров, то инструкции ее вызова предшествует *n* инструкций *param*)
`call p, n`	Вызов процедуры *p*, имеющей *n* параметров
`return y`	Возврат из процедуры *y* – возвращаемое значение

Замечание. Рассмотренное ППСУ – учебное и не содержит некоторых деталей, важных для реализации компилятора. ППСУ системы *GCC* называется *Gimple*. Промежуточное представление *IR* системы *LLVM* не ППСУ, а ППНУ.

Внешне ППНУ не отличается от ППСУ: такая же последовательность инструкций, похожих на ассемблерные. Основное отличие в том, что в инструкциях вместо ссылок на абстрактные области памяти содержатся ссылки на конкретные регистры и конкретные области памяти – переменные, массивы, структуры, элементы массивов, поля структур и т.п. ППНУ рассматривается при изучении машинно-ориентированной оптимизации, без которой невозможно

сгенерировать программу в кодах целевого компьютера, обеспечивающую приемлемый уровень параллельного выполнения команд целевой машины.

2. Базовые блоки и граф потока управления.

Рассмотрим ППСУ программы (рис. 5).

```
(1)        i ← -, m, 1
(2)        j ← n
(3)        t1 ← *, 4, n
(4)        v ← a[t1]
(5)   L1:  i ← +, i, 1
(6)        t2 ← *, 4, i
(7)        t3 ← a[t2]
(8)        ifTrue t3<v goto L1
(9)   L2:  j ← -, j, 1
(10)       t4 ← *, 4, j
(11)       t5 ← a[t4]
(12)       ifTrue t5>v goto L2
(13)       ifTrue i>=j goto L3
(14)       t6 ← *, 4, j
(15)       x ← a[t6]
(16)       t7 ← *, 4, i
(17)       t8 ← *, 4, j
(18)       t9 ← a[t8]
(19)       a[t7] ← t9
(20)       t10 ← *, 4, j
(21)       a[t10] ← x
(22)       goto L1
(23)  L3:  t11 ← *, 4, i
(24)       x ←a[t11]
(25)       t12 ← *, 4, i
(26)       t13 ← *, 4, n
(27)       t14 ← a[t13]
(28)       a[t12] ← t14
(29)       t15 ← *, 4, n
(30)       a[t15] ← x
```

Рис. 5. Пример программы в ППСУ.

В ППСУ можно выделить несколько базовых блоков. Это последовательности следующих одна за другой инструкций **ППСУ**, со следующими свойствами:
(1)поток управления может входить в базовый блок только через его первую инструкцию, т.е. нет переходов в средину базового блока;
(2) поток управления покидает базовый блок без останова или ветвления, за исключением, возможно, в последней инструкции базового блока.
Пример базового блока – инструкции (14) – (22) из примера с рис 5.

```
(14) t6 ← *, 4, i
(15) x ← a[t6]
(16) t7 ← *, 4, i
(17) t8 ← *, 4, j
(18) t9 ← a[t8]
(19) a[t7] ← t9
(20) t10 ← *, 4, j
(21) a[t10] ← x
(22) goto L1
```

Рис. 6. Один из базовых блоков программы с рис. 5..

Вершинами *графа потока управления* являются базовые блоки, а дуги соединяют выход из вершины со входом в вершину, которая может выполняться следующей. В частности, если последней инструкцией базового блока является инструкция условного перехода, то из него будет выходить две дуги.
Если первая инструкция базового блока имеет метку, то в этот базовый блок будут входить дуги из всех базовых блоков, последняя инструкция которых будет инструкцией условного или безусловного перехода на эту метку.

3. Алгоритм построения графа потока управления (ГПУ)

Вход: программа в ППСУ – последовательность инструкций.
Выход: список базовых блоков для данной последовательности инструкций, такой что каждая инструкция принадлежит только одному базовому блоку.
Метод:

1) Строится упорядоченное множество начал базовых блоков (НББ)
   По определению НББ это либо первая инструкция программы, либо помеченная инструкция программы, либо инструкция, следующая за инструкцией перехода.
2) Каждому НББ соответствует ББ, который определяется как последовательность инструкций, содержащая само НББ и все инструкции до следующего НББ (не включая его) или до конца последовательности инструкций.
3) Строится множество дуг графа потока управления:
   1. если последняя инструкция ББ не является инструкцией перехода, строится дуга, соединяющая ББ со следующим ББ;

2. если последняя инструкция ББ является инструкцией безусловного перехода, строится дуга, соединяющая ББ с ББ, НББ которого имеет соответствующую метку;
3. если последняя инструкция ББ является инструкцией условного перехода, строятся обе дуги.

Применим алгоритм к программе из примера с рис. 5.

```
(1)        i ← -, m, 1
(2)        j ← n
(3)        t1 ← *, 4, n
(4)        v ← a[t1]
(5)   L1:  i ← +, i, 1
(6)        t2 ← *, 4, i
(7)        t3 ← a[t2]
(8)        ifTrue t3<v
                goto L1
(9)   L2:  j ← -, j, 1
(10)       t4 ← *, 4, j
(11)       t5 ← a[t4]
(12)       ifTrue t5>v
                goto L2
(13)       ifTrue i>=j
                goto L3
(14)       t6 ← *, 4, j
(15)       x ← a[t6]
(16)       t7 ← *, 4, i
(17)       t8 ← *, 4, j
(18)       t9 ← a[t8]
(19)       a[t7] ← t9
(20)       t10 ← *, 4, j
(21)       a[t10] ← x
(22)       goto L1
(23)  L3:  t11 ← *, 4, i
(24)       x ←a[t11]
(25)       t12 ← *, 4, i
(26)       t13 ← *, 4, n
(27)       t14 ← a[t13]
(28)       a[t12] ← t14
(29)       t15 ← *, 4, n
(30)       a[t15] ← x
```

Началами базовых блоков являются инструкции с номерами (1), (5), (9). (13), (14), (23). Алгоритм позволят построить следующие базовые блоки (рис. 7). Соединив полученные базовые блоки дугами согласно алгоритму, получим граф потока управления (ГПУ) (рис. 8). Для удобства анализа к ГПУ добавлены два дополнительных пустых узла: *entry* (вход) и *exit* (выход).

Блок A

```
(1)     i ← -, m, 1
(2)     j ← n
(3)     tl ← *, 4, n
(4)     v ← a[tl]
```

Блок B

```
(5) L1:  i ← +, i, 1
(6)      t2 ← *, 4, i
(7)      t3 ← a[t2]
(8)   ifTrue t3<v gotoL1
```

Блок C

```
(9)  L2:  j ← -, j, 1
(10)      t4 ← *, 4, j
(11)      t5 ← a[t4]
(12) ifTrue t5>v goto L2
```

Блок D

```
(13) ifTrue i>=j goto L3
```

Блок E

```
(14)     t6 ← *, 4, i
(15)     x ← a[t6]
(16)     t7 ← *, 4, i
(17)     t8 ← *, 4, j
(18)     t9 ← a[t8]
(19)     a[t7] ← t9
(20)     t10← *, 4, j
(21)     a[t10] ← x
(22)     goto L1
```

Блок F

```
(23) L3: t11 ← *, 4, i
(24)     x ← a[t11]
(25)     t12 ← *, 4, i
(26)     t13 ← *, 4, n
(27)     t14 ← a[t13]
(28)     a[t12] ← t14
(29)     t15 ← *, 4, n
(30)     a[t15] ← x
```

Рис. 7. Базовые блоки рассматриваемого примера.

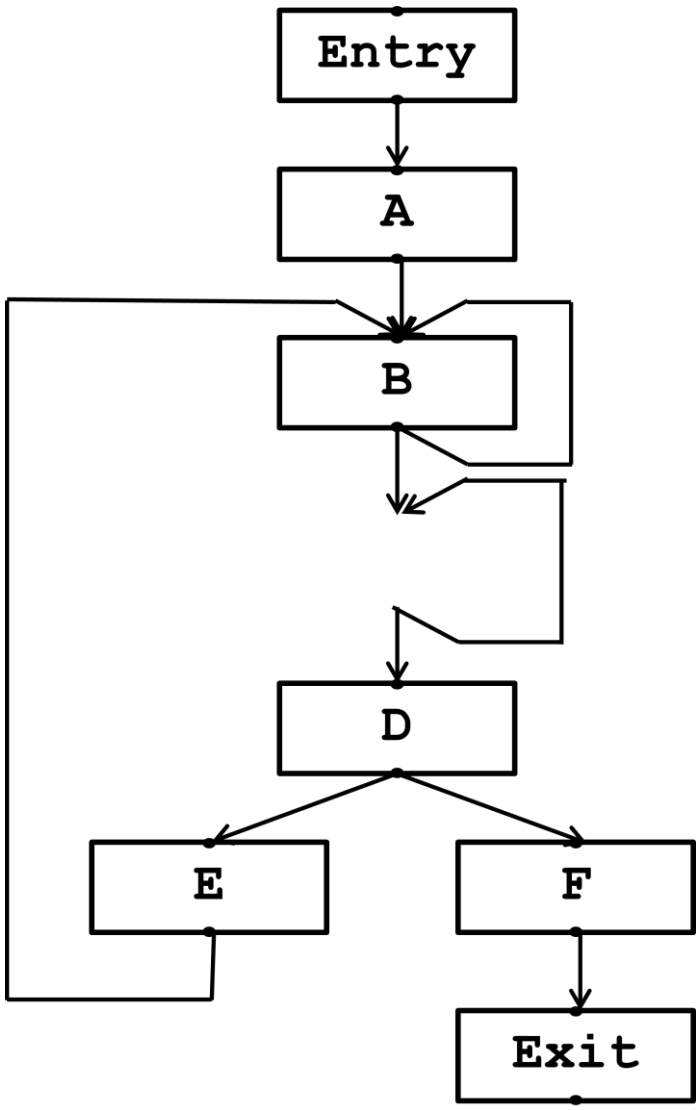


Рис. 8.

4.Локальная оптимизация.

4.1 Постановка задачи локальной оптимизации

Базовым блоком B будем называть тройку объектов

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

где *P* – последовательность инструкций блока *B*,
Input – множество переменных, определенных до входа в блок *B*, и доступных в блоке *B*
Output – множество переменных, используемых после выхода из блока *B*.
Оптимизация – это выполнение в блоке *B* следующих преобразований:

1) Удаление *общих подвыражений* (инструкций, повторно вычисляющих уже вычисленные значения).

2) Удаление *мертвого кода* (инструкций, вычисляющих значения, которые впоследствии не используются).
3) *Сворачивание констант* (вычисление выражений над известными во время компиляции константами). Например, выражение 2 * 3.1415926 можно заменить значением 6.2831852.
4) Изменение порядка инструкций, там, где это возможно.

Все вышеперечисленные преобразования можно выполнить за один просмотр ББ, представив его в виде *ориентированного ациклического графа* (ОАГ).

4.2. Представление базового блока в виде ориентированного ациклического графа

Простейший пример. Выражение

```
a + a * (b-c) + (b-c) * d
```

можно представить в виде фрагмента АСД (левая часть рис. 9), либо в виде ориентированного ациклического графа (правая часть рис. 9)

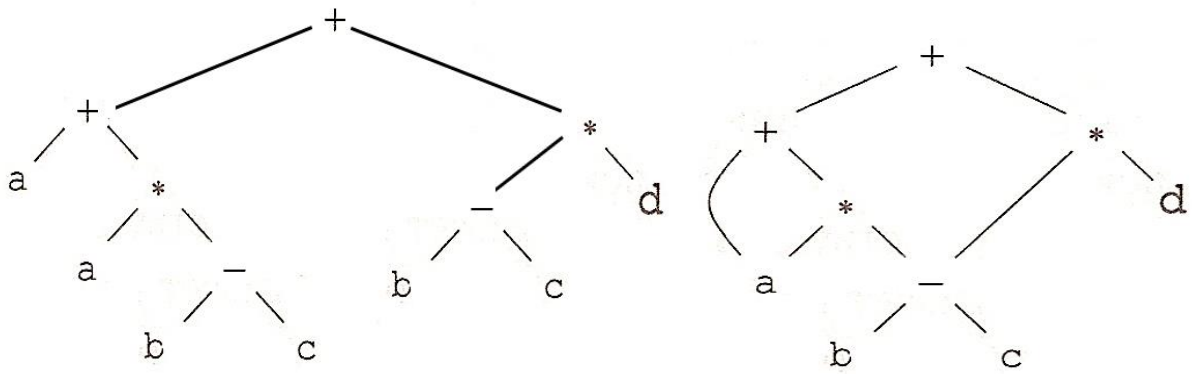


Рис. 9. Слева – фрагмент АСД, справа – ОАГ.

Основное отличие DAG от АСД в том, что в DAG каждое значение представляется только один раз: узлы, представляющие в АСД одинаковые значения, «склеиваются».

Пример. Выражение в исходном коде:

```
a = a + y*(b +(y-z)*b)+(y-z)* b
```

На ОАГ (правая часть рис. 10) видно (в отличие от АСД), что значение переменной **b** используется в двух вычислениях, что выражение **y-z** можно вычислить только один раз и что выражение **(y-z)* b** можно вычислить только один раз.

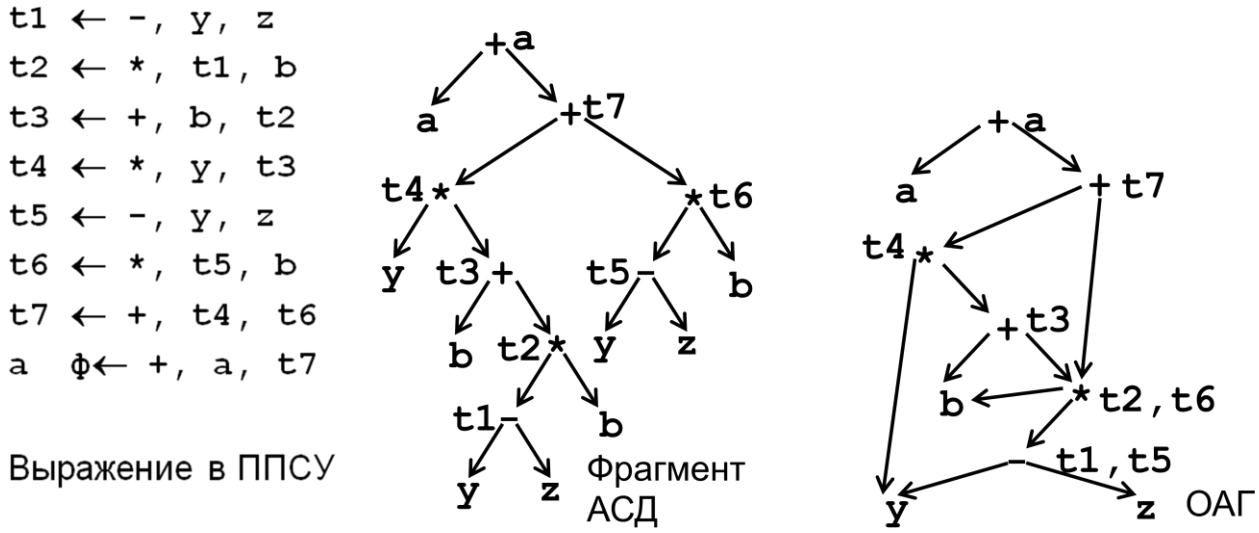


Рис. 10.

4.3 Метод нумерации значений (локальный)

Представим *ОАГ* в виде хеш-таблицы

Каждая строка хеш-таблицы представляет один узел ОАГ.

Первое поле каждой записи представляет код операции

Каждой правой части операции ⟨**op, left, right**⟩, где **op** – код операции, а **left** и **right** – левый и правый операнды, соответствует ее *сигнатура* ⟨**op, #left, #right**⟩, где **op** – код операции, а **#left** и **#right** – номера значений левого и правого операндов (у унарных операций **#right** равен 0).

Унарные операции **id** и **nm** определяют соответственно имена переменных и константы (листовые узлы).

Номер значения – это номер строки таблицы значений (ТЗ), в которой определяется это значение.

Алгоритм (на псевдокоде) построения ОАГ для базового блока *B*, содержащего *n* инструкций вида $t_i \leftarrow Op_i, l_i, r_i$.

Функция **#val(s)** определяет номер значения, определяемого сигнатурой **s = (Op, #val(l), #val(r))**.

for each "$t_i \leftarrow Op_i, l_i, r_i$" do
s_i = (Op_i, #val(l_i), #val(r_i))
if(ТЗ содержит s_j == s_i)
then вернуть j в качестве номера значения #val(s_i)
else завести в ТЗ новую строку $ТЗ_k$, записать сигнатуру s_i в строку $ТЗ_k$, вернуть k в качестве номера значения #val(s_i)

Пример применения алгоритма. Таблица значений рассматриваемого примера имеет вид:

1	id	ссылка в ТС		a
2	id	ссылка в ТС		b
3	id	ссылка в ТС		y
4	id	ссылка в ТС		z
5	-	3	4	t1, t5
6	*	5	2	t2, t6
7	+	2	6	t3
8	*	3	7	t4
9	+	8	6	t7
10	+	1	9	a
# значения	КОП	# операнда	# операнда	Присоединенные переменные
	Определение значения (сигнатура)			

Рис. 11.

t1 ← -, y, z t2 ← *, t1, b t3 ← +, b, t2 t4 ← *, y, t3 t5 ← -, y, z t6 ← *, t5, b t7 ← +, t4, t6 a ← +, a, t7	$t1^5$ ← -, y^3, z^4 $t2^6$ ← *, $t1^5$, b^2 $t3^7$ ← +, b^2, $t2^6$ $t4^8$ ← *, y^3, $t3^7$ $t5^5$ ← -, y^3, z^4 $t6^6$ ← *, $t5^5$, b^2 $t7^9$ ← +, $t4^8$,$t6^6$ a^{10} ← +, a^1, $t7^9$	t1 ← -, y, z t2 ← *, t1, b t3 ← +, b, t2 t4 ← *, y, t3 t5 ← t1 t6 ← t2 t7 ← +, t4, t6 a ← +, a, t7
До оптимизации	После нумерации значений	После оптимизации

Рис. 12.

Следующий пример (рис. 13) разъясняет, для чего нужны множества *Input*, и *Output*. Эти множества отражают связь между рассматриваемым базовым блоком и другими базовыми блоками программы. В следующем разделе будет рассмотрен метод глобального анализа, позволяющий вычислить эти множества.

Пример. Рассмотрим базовый блок $B = \langle P, \{a, b, c, d\}, \{a, b\}\rangle$

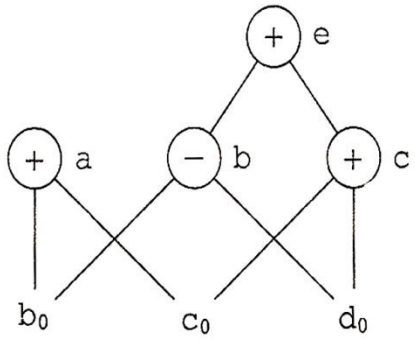
a ← +, b, c **b ← -, b, d** **c ← +, c, d** **e ← +, b, c**	**a ← +, b, c** **b ← -, b, d**	
Блок *B* до оптимизации	После оптимизации	ОАГ базового блока *B*

Рис. 13.

Новое значение переменных **c** и **e** можно не вычислять, так как они не входят в множество *Output*

4.4 Восстановление базового блока по его ОАГ

Правила

1. Для каждого узла с одной или несколькими связанными переменными строится трехадресная инструкция, которая вычисляет значение одной из этих переменных.

2. Если у узла несколько присоединенных живых переменных, то следует добавить команды копирования, которые присвоят корректное значение каждой из этих переменных.

Пример. Пусть ОАГ представлен следующим массивом записей (рис. 14).

1	b_0	ссылка	в ТС	
2	c_0	ссылка	в ТС	
3	d_0	ссылка	в ТС	
4	+	1	2	a
5	-	4	3	d, b
6	+	5	2	c
8	a	ссылка	в ТС	
9	b	ссылка	в ТС	
10	c	ссылка	в ТС	
11	d	ссылка	в ТС	

Применяя правила 1 и 2, получим базовый блок $B = \langle P, Input, Output \rangle$

где $P =$ $\{a \leftarrow +, b_0, c_0$

$d \leftarrow -, a, d_0$

$c \leftarrow +, d, c_0$

$b \leftarrow d\}$

$Input = \{b_0, c_0, d_0\}$

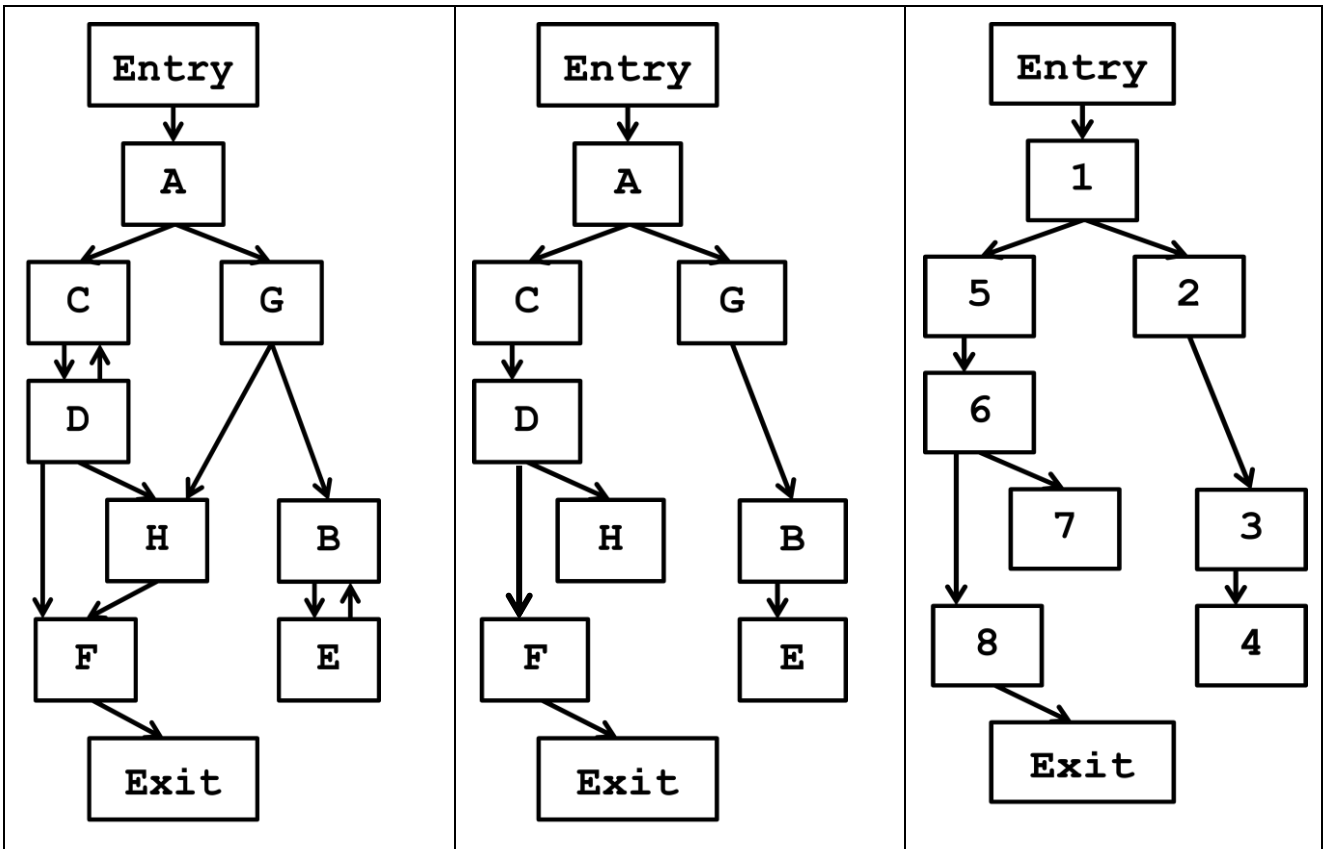
$Output = \{a, b, c, d\}$

Стремление разработать полноценную локальную оптимизацию привело к необходимости построить связи по данным оптимизируемого базового блока с другими базовыми блоками, т.е. к необходимости глобального анализа оптимизируемой процедуры.

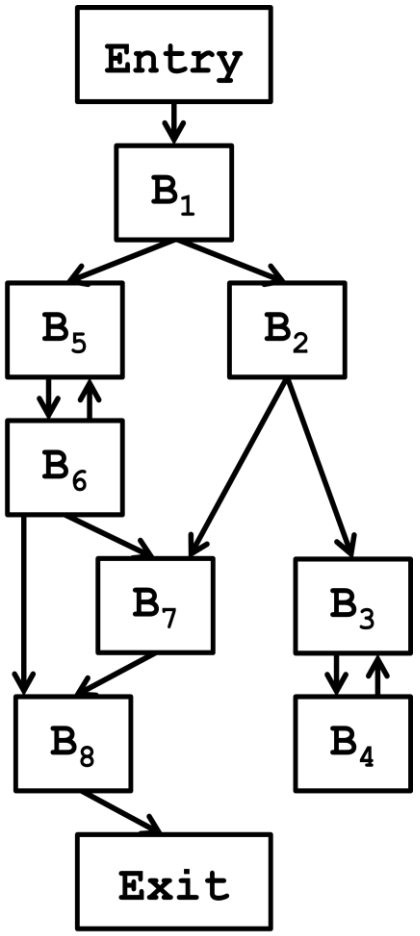
5. Глобальная оптимизация

5.1 Нумерация вершин ГПУ

Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию» (рис. 15 и 16).



ГПУ до нумерации вершин	Глубинное ОД	Нумерация вершин ОД



ГПУ после нумерации вершин

Остовное дерево графа содержит все вершины графа и часть его дуг.

Обратная нумерация используется для того, чтобы, например, вершина A имела номер 1, а не 8.

Построенное ОД будем называть глубинным остовным деревом, чтобы помнить, как пронумерованы его вершины

В случае обратной нумерации вершин графа, содержащего *n* вершин *i*-ой вершине присваивается номер *n* - *i*

Присвоив номера вершин остовного дерева соответствующим базовым блокам, получим ГПУ с пронумерованными вершинами.

Далее следует рекурсивный алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ (на псевдокоде).

Рис. 15. Нумерация вершин ГПУ.

Вход: ГПУ $G = \langle N, E \rangle$ с корнем $Entry \in N$

Выход: глубинное остовное дерево графа *G* и нумерация графа *G*, соответствующая упорядочению в глубину.

Метод:

Все узлы $n \in N$ помечаются как *nv* (*not visited*) после чего вызывается рекурсивная процедура **`DFST(n0);`** когда процедура **`DFST`** завершится, будет построен массив номеров узлов **`dfn.`**

Процедура **`DFST():`**

```
void DFST(n) {
     Отмечаем n как v;
     for each s ∈ Succ(n)
          if (s.vst == nv){T ∪= {n→s}; DFST(s);}
     dfn[c] = n - c;
     c--;
}
```

5.2. Анализ потока данных

Состояние программы – множество значений всех переменных программы, включая переменные во фреймах стека времени выполнения, находящихся ниже текущей вершины стека. Состояния программы определены в ее точках.
Точки программы (**…, p_j, p_{j+1}, p_{j+2}, …**) расположены между ее *инструкциями* (**…, I_j, I_{j+1}, …**). На рис. 16 точки программы изображены в виде черных кружочков.

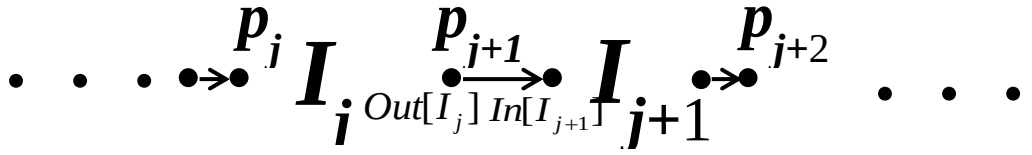


Рис. 16. Инструкции и точки программы

Инструкция программы I_j описывается парой состояний: состоянием в *точке программы* p_j *перед* инструкцией I_j и состоянием в *точке программы* p_{j+1} *после* инструкции I_j. Состояние программы в точке p_j называется входным состоянием инструкции I_j и обозначается $In[I_j]$ (правая точка, помеченная p_j), состояние программы в точке p_{j+1} называется выходным состоянием инструкции I_j и обозначается $Out[I_j]$ (левая точка, помеченная p_{j+1}). Точки разделяются и соединяются стрелкой, так как инструкция I_{j+1} может быть помеченной и тогда в точку, соответствующую ее входу будет входить две и более стрелок (по числу инструкций `goto`, ссылающихся на соответствующую метку).

Считается, что с каждой инструкцией I_j связаны две передаточные функции: передаточная функция *прямого* обхода ГПУ (от входного до выходного состояния) f_{I_j} и передаточная функция *обратного* обхода (от выходного до входного состояния) $f^b_{I_j}$.
Таким образом, имеют место соотношения: $Out[I_j] = f_{I_j}(In[I_j])$ при прямом обходе и $In[I_j] = f^b_{I_j}(Out[I_j])$ при обратном.
Рассмотрим базовый блок $B = \langle P, Input, Output\rangle$, где $P = I_1, ..., I_n$ (в указанном порядке). По определению $In[B] = In[I_1]$, $Out[B] = Out[I_n]$.

Передаточная функция f_B блока B по определению равна композиции передаточных функций его инструкций $I_1, ..., I_n$:

$$f_B(x) = f_{I_n}(f_{I_{n-1}}(...f_{I_1}(x)...)) = (f_{I_1} \circ f_{I_2} \circ ... \circ f_{I_n})(x)$$

или

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

При прямом обходе (от *Entry* к *Exit*) ГПУ (а следовательно, и каждого базового блока) соотношение между потоком данных при выходе из блока B и потоком данных при входе в него имеет вид $Out[B] = f_B(In[B])$.

При обратном обходе ГПУ соотношение между потоком данных при входе в блок B и потоком данных при выходе из него имеет вид $In[B] = f_B^b(Out[B])$.

2.3 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 определение переменной x *убивает* (отменяет) все другие ее определения.

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.

Замечание. Отметим, что во время анализа достигающих определений рассматриваются не переменные, а их определения, причем каждая переменная может иметь несколько определений в разных базовых блоках.

Пример. Рассмотрим процедуру (рис.17). В базовых блоках этой процедуры 7 определений ее переменных (пара $\langle i, B_1\rangle$ обозначает определение переменной i в блоке B_1):
$d_1 = \langle i, B_1\rangle$, $d_2 = \langle j, B_1\rangle$, $d_3 = \langle a, B_1\rangle$, $d_4 = \langle i, B_2\rangle$, $d_5 = \langle j, B_2\rangle$, $d_6 = \langle a, B_3\rangle$, $d_7 = \langle i, B_4\rangle$,

Начало блока B_1 **не достигается** ни одним определением,

Начало блока B_2 **достигается** определениями: d_1, d_2, d_3, d_5, d_6, d_7. Начало блока B_2 **не достигается** определением: d_4, так как его убивает определение d_7.

Начало блока B_3 **достигается** определениями: d_3, d_4, d_5, d_6. Начало блока B_3 **не достигается** определением: d_7, так как его убивает определение d_4.

Начало блока B_4 **достигается** определениями: d_3, d_4, d_5, d_6. Начало блока B_4 **не достигается** определением: d_7, так как его убивает определение d_4.

B_1
`i ← -, m, 1`
`j ← n`
`a ← u1`
	B_2
	`i ← +, i, 1`
	`j ← -, j, 1 a`

B_3 `a ← u2`	B_4 `i ← u3`

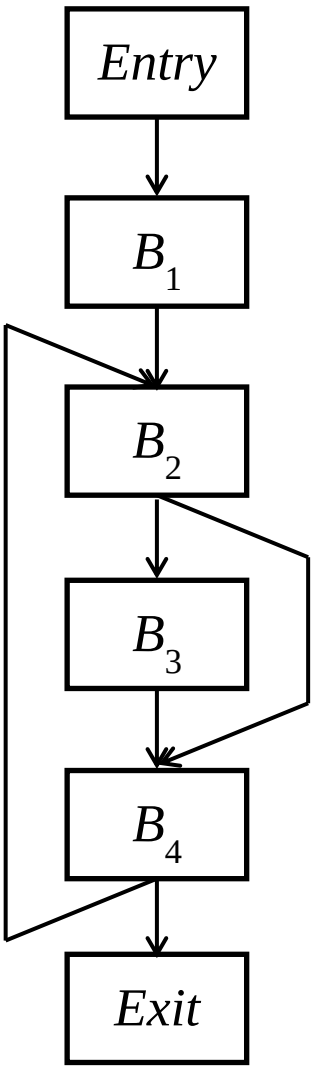


Рис. 17. Пример достигающих определений.

2.4. Передаточные функции достигающих определений

Рассмотрим инструкцию I
d: **`u = v + w`**,
расположенную между точками p_1 и p_2 программы.
По определению передаточной функции $\boldsymbol{y} = f_I(\boldsymbol{x})$ инструкция I сначала убивает все предыдущие определения **u** , а потом порождает d – новое определение **u**.
Следовательно, $\boldsymbol{y} = gen_I \cup (\boldsymbol{x} - kill_I)$, и передаточная функция f_I инструкции I может быть записана в виде:

$$f_I(x) = gen_I \cup (x - kill_I)$$

где $\boldsymbol{x}$ – состояние во входной точке инструкции I.

Определение. Передаточные функции, определяемые соотношением

$$f(x) = gen \cup (x - kill)$$

будем называть передаточными функциями вида *gen-kill*.

Утверждение 1. Композиция двух функций вида *gen-kill* является функцией вида *gen-kill.*

В самом деле

$$\begin{aligned}
&(f_1 \circ f_2)(x) = f_2(f_1(x)) = \\
&= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2) = \\
&= gen_2 \cup (gen_1 - kill_2) \cup (x - kill_1 - kill_2)
\end{aligned}$$

Откуда следует, что

$$(f_1 \circ f_2)(x) = gen_{f_1 \circ f_2} \cup (x - kill_{f_1 \circ f_2}),$$

$$gen_{f_1 \circ f_2} = gen_2 \cup (gen_1 - kill_2)$$

$$kill_{f_1 \circ f_2} = kill_1 \cup kill_2$$

Утверждение 2.

Пусть базовый блок *B* содержит *n* инструкций, каждая из которых имеет передаточную функцию $f_i(x) = gen_i \cup (x - kill_i)$ *i = 1, 2, ..., n*. Тогда передаточная функция для базового блока *B* может быть записана как

$$f_B(x) = gen_B \cup (x - kill_B),$$

где

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

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup ...$$
$$\cup (gen_1 - kill_2 - kill_3 - ... - kill_n)$$

Следствие.

Если какая-либо переменная определяется в блоке B несколько раз, то в gen_B войдет только ее последнее определение.

2.5. Система уравнений и ее решение

Таким образом, для каждого базового блока B_i можно выписать уравнение

$$Out[B_i] = f_B(In[B_i]),$$

которое в случае анализа достигающих определений имеет вид

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

Если ГПУ содержит n базовых блоков, получается n уравнений относительно $2 \cdot n$ неизвестных $In[B_i]$ и $Out[B_i]$, $i = 1, 2, \ldots, n$.

Еще n уравнений получается с помощью сбора вкладов путей.

Определение достигает точки программы, тогда и только тогда, когда существует по крайней мере один путь, вдоль которого эта точка может быть достигнута. Этот путь должен пройти через какую-нибудь вершину из $Pred(B)$, причем если путь, проходящий через вершину $P \in Pred(B)$, не проходит ни через одну вершину, содержащую определение какой-либо переменной из B, то $Out(P)=\varnothing$. Следовательно, $In[B] = \bigcup_{P \in Pred(B)} Out[P]$.

Напоминаю: $Pred(B)$ это множество всех вершин ГПУ, которые непосредственно предшествуют вершине B.

Получается система уравнений:

$$Out[B_i] = gen_{B_i} \cup (In[B_i] - kill_{B_i}),$$
$$In[B_i] = \bigcup_{P \in Pred(B_i)} Out[P].$$

($i = 1, 2, \ldots, n$).

Отметим, что в случае обхода ГПУ сверху вниз, интересны только множества переменных на входе в каждый базовый блок. Поэтому, произведя очевидную подстановку, приведем полученную систему уравнений к виду

$$In[B_i] = \bigcup_{P \in Pred(B_i)} \left(gen_P \cup (In[P] - kill_P)\right)$$

($i = 1, 2, \ldots, n$).

Полученную систему уравнений можно решать методом итераций.
В качестве начальных итераций $In[B_i]$ возьмем пустые множества $\left(In[B_i]\right)^0 = \varnothing$

Если для одного или более блоков B_i множество $Pred(B_i)$ будет содержать вершину $Entry$, будем использовать «граничное условие» $In[Entry] = \varnothing$

При обходе ГПУ базовые блоки посещаются в порядке их номеров: $Entry$, B_1, B_2 и так далее.

2.6. Алгоритм для вычисления достигающих определений

Алгоритм «Достигающие определения»

Вход: ГПУ (N, E), в котором для каждого базового блока $B_i \in N$ вычислены множества gen_{B_i} и $kill_{B_i}$

Выход: множества $In[B_i], (i = 1,2,..., n)$ достигающих определений на входе в каждый базовый блок B_i графа потока управления

Метод: Используется метод итераций с начальной итерацией $(In[B_i])^0 = \varnothing$.На каждой итерации r (r – номер итерации) $(In[Entry])^r = \varnothing$ (граничное условие) Итерации продолжаются до тех пор, пока все множества $(In[B_i])^r$ не перестанут изменяться.

Необходимо учесть, что для каждого базового блока окончательный результат получается за свое количество итераций. Поэтому в алгоритме используется список **WorkList,** в который сначала включаются все базовые блоки, а в дальнейшем только те, для которых множество **In** изменилось. Это значительно ускоряет алгоритм.

```
In[Entry] = ∅;
WorkList = ∅;
for (каждый базовый блок B,отличный от Entry){
поместить B в WorkList;
In[B] = ∅;/* Каждому In[B] присваивается значение
его нулевой итерации */
};
do {                                    /*основной цикл*/
  Выбрать из очереди WorkList очередной блок B
  Вычислить InNew[B], используя уравнение
```

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

```
  /*При вычислении InNew[B] множества In[P],
    где P ∈ Pred(B), могут иметь значения либо
    текущей, либо следующей итерации*/
     if (InNew[B] ≠ In[B]){
         In[B] = InNew[B];
         Поместить In[B] в конец очереди WorkList}
     } while |WorkList|>0;
}
```

Применим алгоритм к примру из раздела 2.3 (рис. 18).

B_1 **i ← -, m, 1** **j ← n** **a ← u1**	B_2 **i ← +, i, 1** **j ← -, j, 1 a**
B_3 **a ← u2**	B_4 **i ← u3**

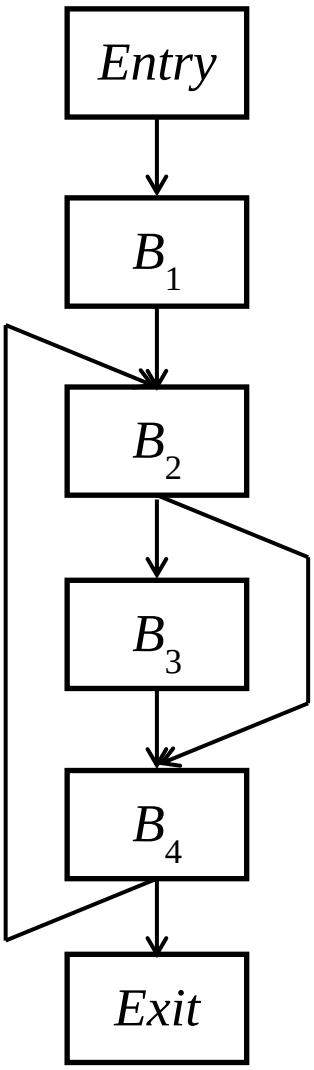


Рис. 18. Вычисление достигающих определений для примера с рисунка 17.

В простой программе справа 7 определений. Требуется определить, какие определения достигают входов в ее 5 базовых блоков **B1, B2, B3,B4** и **Exit.**

Вычислим множества *gen* и *kill* для каждого базового блока

B	gen_B	$kill_B$
B_1	**(1110000)**	**(0001111)**
B_2	**(0001100)**	**(1100001)**
B_3	**(0000010)**	**(0010000)**
B_4	**(0000001)**	**(1001000)**

Множества из 7 элементов удобно представлять битовыми векторами, длины 7, каждый разряд соответствует одному элементу множества (он равен 1, если соответствующий элемент принадлежит рассматриваемому множеству, и 0 в противном случае). **(0000000)** – пустое множество, **(1111111)** – множество **{d1, d2, d3, d4, d5, d6, d7}**, **(1110000)** – множество **{d1, d2, d3}**.

Множества Pred(B): **Pred(B_1)= {Entry}, Pred(B_2)= {B_1, B_4}, Pred(B_3)= {B_2}, Pred(B_4)= {B_2, B_3}, Pred(Exit)= {B_4}**

Рассмотрим первую итерацию. В **WorkList** загружается нулевая итерация – пустые множества для B_1, B_2, B_3, B_4 и *Exit*.

Вычисление $(In[B_1])^1$. Поскольку $Pred(B_1) = \{Entry\}$ формула для вычисления $(In[B_1])^1$ имеет вид $(In[B_1])^1 = gen_{Entry} \cup ((In[Entry])^0 - kill_{Entry}) = \varnothing = (In[B_1])^0$.

Вычисление $(In[B_2])^1$.

$$(In[B_2])^1 = \left(gen_{B_1} \cup \left(\left(In[B_1]\right)^1 - kill_{B_1}\right)\right) \cup \left(gen_{B_4} \cup \left(\left(In[B_4]\right)^0 - kill_{B_4}\right)\right) =$$

= (1110000) ∪ (∅ - (0001111)) ∪ (0000001) ∪ (∅ - (1001000)) = (1110001)

$(In[B_1])^1 = (In[B_1])^0$ и в **WorkList** не попадает. $(In[B_2])^1 \neq (In[B_2])^0$ и снова попадает в **WorkList**.

И так далее.

B	$(In[B])^1$	$(In[B])^2$	$(In[B])^3$
B_1	0000000	0000000	0000000
B_2	1110001	1110111	1110111
B_3	0011100	0011110	0011110
B_4	0011111	0011110	0011110
Exit	0010111	0010111	0010111

Результаты сведены в таблицу. Как видим, для рассматриваемого простейшего примера хватило двух итераций.

При локальной оптимизации в качестве множеств *Input*[*B*] для базовых блоков обычно берутся множества *In* [*B*], вычисленные для достигающих определений достигающих определений.

2.6. Анализ живых переменных.

Живыми называются переменные, которые после их определения, используются в полезных вычислениях (т.е. в вычислениях, влияющих на результат программы). Цель анализа живых переменных – для определения переменной *x* в точке *p*

программы выяснить, будет ли указанное значение *x* использоваться вдоль какого-нибудь пути, начинающегося в точке *p*. Если будет, то переменная *x жива* в точке *p*, если нет, то переменная *x мертва* в точке *p*, а ее определение в точке *p* является мертвым кодом.

Пусть в блоке *B* используется переменная *v*. Возможны два случая:
1) используется определение *v* в одном из блоков $B' \in Pred^*(B)$;
2) используется определение *v* в самом блоке *B*.

В первом случае говорят, что *v* жива на выходе из *B′*, во втором случае говорят, что *v* мертва на выходе из *B′*. Поскольку все зависит от использования *v* в блоках между B и $Exit$, анализ целесообразно проводить, обходя ГПУ в направлении

Рассмотрим тривиальный пример: пусть в самом начале блока *B* имеются инструкции (определения переменных **x, y** и **z** находятся в блоке $B' \in Pred^*(B)$):

```
x ← +, x, 1
y ← -, x, 2
z ← +, x, y
```

На входе в блок B жива только переменная **x**, а определения **y** и **z** вне блока B убиваются второй и третьей инструкциями блока *B*.

Определим два множества:

use_B – множество переменных, используемых в блоке B до их определения в этом блоке (любая переменная из use_B *жива* на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred^*(B)$

def_B – множество переменных, определяемых в блоке *B* до их использования в этом блоке (любая переменная из def_B *мертва* на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred^*(B)$)

Уравнения потока данных связывают def и use с неизвестными In и Out, следующим образом:

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

К ним добавляется граничное условие $In\ [Exit] = \varnothing$

Если значение In, определяемое первым уравнением подставить во второе уравнение, множества In будут исключены из системы уравнений и получится система уравнений, содержащая в качестве неизвестных только множества Out:

$$Out[B] = \bigcup_{S \in Succ(B)} (use_S \cup (Out[S] - def_S))$$

В заключение приведем алгоритм решения системы уравнений методом итераций с обходом ГПУ в направлении от *Exit* к *Entry*.

```
{Out[Exit] = ∅;
    WorkList = ∅;
    for(каждый базовый блок B,отличный от Exit){
            Out[B] = ∅;
        WorkList ∪= {B}
        }
    /*основной цикл*/
    do { Извлечь блок B из WorkList (исключив B);


         if (OutNew[B] ≠ Out[B]){
             Out[B] = OutNew[B] ;
             WorkList ∪= {B}
             }
        } while |WorkList|>0;
    }
```

3. Полурешетки.

При анализе потока данных рассматриваются множества переменных для описания состояния и такие операции как *объединение* ($\cup$) и *пересечение* ($\cap$) множеств.

Свойства операций $\cup$ и $\cap$:

$A \cup A = A$	$A \cap A = A$	*(идемпотентность)*
$A \cup B = B \cup A$	$A \cap B = B \cap A$	*(коммутативность)*
$A \cup (B \cup C) = (A \cup B) \cup C)$	$A \cap (B \cap C) = (A \cap B) \cap C$	*(ассоциативность)*
Если $A \cup B = A$, то $B \subseteq A$	Если $A \cap B = A$, то $B \supseteq A$	отношение частичного порядка,

	связанное с операцией

Пусть множество U содержит все элементы, тогда любое множество $A \subseteq U$ и $A \cup U = U \cup A = U$

Для пересечения ($\cap$) роль U играет $\varnothing$: любое множество $A \supseteq \varnothing$ и $A \cap \varnothing = \varnothing \cap A = \varnothing$

Полурешетка – это абстрактная алгебраическая структура, над элементами которой определена абстрактная операция $\wedge$ (**сбор**), обладающая свойствами операций $\cup$ и $\cap$.

Определение. *Полурешетка* представляет собой множество L, на котором определена бинарная операция «*сбор*» $\wedge$, такая, что для всех x, y и $z \in L$:

$x \wedge x = x$ (*идемпотентность*)

$x \wedge y = y \wedge x$ (*коммутативность*)

$x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (*ассоциативность*)

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

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.

(1) *Рефлексивность* $\leq$ следует из идемпотентности $\wedge$: $x \leq x \Leftrightarrow x \wedge x = x$

(2) *Антисимметричность* $\leq$ следует из коммутативности $\wedge$: пусть $x \leq y$ и $y \leq x$; тогда $x = x \wedge y = y \wedge x = y$

(3) *Транзитивность* $\leq$ следует из ассоциативности $\wedge$: пусть $x \leq y$ и $y \leq z$; тогда по определению $\leq$
$x \wedge y = x$ и $y \wedge z = y$; $(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) - (x \wedge y) = x$, $(x \wedge z) = x \Leftrightarrow x \leq z$.

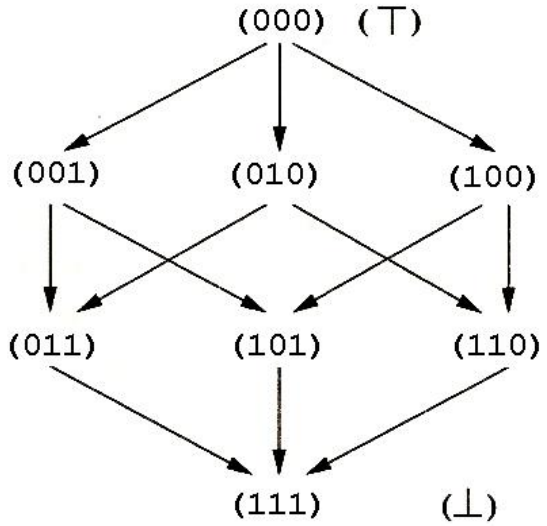
Определение. Пусть $\langle L, \leq \rangle$ – частично упорядоченное множество (в частности, полурешетка). *Наибольшей нижней границей* $inf(x, y)$ элементов x и $y \in L$

называется элемент $g \in L$, такой, что $g \le x$; $g \le y$; и если $z \in L$, такой, что $z \le x$ и $z \le y$, то $z \le g$.

Утверждение. Если x и $y \in L$, где $\langle L, \wedge \rangle$ – полурешетка, то $inf\,(x, y) = x \wedge y$.

Диаграмма полурешетки $\langle L, \wedge \rangle$ представляет собой граф, узлами которого являются элементы L, а ребра направлены от x к y, если $y \le x$.

Пример. На рисунке 19 – диаграмма полурешетки $\langle U, \cup \rangle$, $|U| = 8$:
элемент множества U представляется битовым 3-вектором.



Замечание. Определение элемента $\perp$ внизу диаграммы: для любого $x \in L$:
$\perp \wedge x = \perp$
Этот элемент называется «низом», так как по определению отношения $\le$ для любого $x \in L$ $\perp \le x$.
Полурешетка с операцией $\wedge$ может и не содержать $\perp$.

Рис. 19.

4. Структура потока данных

Определение. 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 (множество элементов полурешетки),
$\wedge$ - реализация операции сбора.

Примеры.
1. Структура потока данных для анализа **достигающих определений**:
$\langle Forward, GK, Def, \cup \rangle$,
где GK – семейство передаточных функций вида gen-kill,
Def – множество определений переменных.

2. Структура потока данных для анализа **живых переменных**:
$\langle Backward, LV, Var, \cup \rangle$,
$$f(x) = use \cup (x - def)$$

где LV – семейство передаточных функций вида,
Var – множество переменных программы.

Определение. Семейство передаточных функций F называется *замкнутым*, если:
F содержит тождественную функцию I: $\forall x \in L$: $I(x) = x$.
F замкнуто относительно операции композиции:

Можно доказать, что семейства передаточных функций, используемых для анализа достигающих определений и живых переменных, замкнуты.

Определение 1. Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной,* если $\forall x, y \in L, \forall f \in F\ (x \le y) \Rightarrow f(x) \le f(y)$.

Определение 2. Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *монотонной,* если $\forall x, y \in L, \forall f \in F\ \ f(x \wedge y) \le f(x) \wedge f(y)$.

Можно доказать, что эти определения эквивалентны.

Определение 3. Структура потока данных $\langle D, F, L, \wedge \rangle$ называется *дистрибутивной,* если $\forall x, y \in L, \forall f \in F: f(x \wedge y) = f(x) \wedge f(y)$.

Можно доказать, что если структура потока данных $\langle D, F, L, \wedge \rangle$ дистрибутивна, то она монотонна. Обратное утверждение неверно.

Можно доказать, что структуры потока данных для анализа достигающих определений и живых переменных дистрибутивны.

5. Обобщенный итеративный алгоритм.

Структура потока данных дает возможность сформулировать 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 ГПУ.

Метод:

```
if (D = Forward) {
    Out[Entry] = l;
    for (each B ≠ Entry) Out[B] = ⊤;
```

```
    while (внесены изменения в In[B]) {
        for (each B ≠ Entry) {
```

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

```
        }
    }
}
if (D = backward) {
    Out[Exit] = 1;
    for (each B ≠ Exit) Out[B] = T;
    while (внесены изменения в Out[B]) {
        for (each B ≠ Exit) {
    }
```

$$Out[B] = \bigwedge_{S \in Succ(B)} f_S(Out[S])$$

```
        }
    }
}
```

Можно доказать, что если обобщенный итеративный алгоритм сходится, то полученный результат является решением системы уравнений потока данных. Более того, можно доказать, что решение, полученное с помощью итеративного алгоритма, является максимальной фиксированной точкой системы уравнений. *Максимальная фиксированная точка* системы уравнений по определению представляет собой решение $\{In[B_i]^{max}\}$ этой системы, обладающее тем свойством, что для любого другого решения $\{In[B_i]\}$ выполняются условия $In\,[B_i] \le In[B_i]^{max}$ 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 ГПУ 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 при прямом обходе (последователей при обратном обходе) и тогда используется значение потока данных, соответствующее предыдущей итерации.

(4) для процесса итерации *необходимо граничное условие*, так как к блоку *Entry* передаточная функция неприменима.

(5) в качестве «нулевой итерации» все *In*[*B*] (*Out*[*B*] при обратном обходе) инициализируются значением ⊤, которое, по определению, «не меньше» всех значений потока, и, следовательно, того значения, которое оно заменяет; при этом монотонность передаточных функций обеспечивает получение результата, «не меньшего», чем искомое решение.

6. Смысл решения уравнений потока данных

Без потери общности будем считать, что рассматривается прямая задача (обход графа потока от *Entry* к *Exit*). Обратная задача (обход графа потока от *Exit* к *Entry*) рассматривается аналогично.

Пусть f_{B_k} – передаточная функция блока B_k в ГПУ. Рассмотрим путь
$P = Entry \to B_1 \to B_2 \to \ldots \to B_{k-1} \to B_k$

(Путь *P* может содержать циклы: базовые блоки могут встречаться в нем по нескольку раз).

По определению *передаточная функция пути P*:

$$f_P = f_{B_1} \circ f_{B_2} \circ \ldots \circ f_{B_{k-1}}$$

(f_{B_k} не входит в композицию, так как путь достигает начала блока B_k, но не его конца). Значение потока данных, создаваемое этим путем, представляет собой $f_P(l_{Entry})$, где l_{Entry} – граничное условие.

Пусть $\mathcal{P} = \{P_1, P_2, \ldots\}$ – множество *всех выполнимых* путей от $Entry$ до B_k

Путь P является *выполнимым* только тогда, когда известно выполнение программы (начальные данные), которое следует в точности по этому пути.

Идеальным решением системы уравнений потока данных для $In[B_k]$ будет:

$$Ideal[B_k] = \bigwedge_{F \in P} f_P(l_{Entry})$$

Решение $Ideal[B_k]$ названо идеальным, так как
(1) оно наиболее точное и
(2) вычислить его почти никогда не удается, так как поиск всех возможных путей выполнения – задача неразрешимая.

Следовательно, требуется поиск приближенного решения.

Добавление еще одного пути в сбор делает решение «меньше» в смысле частичного порядка полурешетки ≤

Если просмотрены все выполнимые пути $\bigwedge_{F \in P}$ и ни одного лишнего, получается идеальное решение $Ideal$.

Решение Sol_1, такое, что $Ideal \leq Sol_1$, получается, когда *просмотрены не все выполнимые пути*. Использовать решение Sol_1 опасно, так как для не просмотренных путей преобразования программы могут оказаться неверными (будет нарушена консервативность).

Решение Sol_2, такое, что $Sol_2 \leq Ideal$ обладает следующими свойствами:

(1) Sol_2 консервативно: оно содержит все выполнимые пути

(2) Sol_2 неточно, так как в нем не отсеяны «лишние» пути, т.е. либо пути, не существующие в ГПУ, либо существующие, но такие, по которым программа никогда не проследует.

В результате:
(1) Sol_2 может запрещать некоторые из преобразований, разрешенных решением $Ideal$.
(2) все преобразования, которые разрешает Sol_2, корректны
Решение сбором по всем путям (*MOP-решение*) от *Entry* до входа в B_k определяется соотношением

$$MOP[B_k] = \bigwedge_{P \in Q} f_P(l_{Entry})$$

где $Q = \{P_1, P_2, \ldots\}$ – множество *всех* путей от *Entry* до входа в блок B_k

Пути, рассматриваемые в *MOP*-решении, – это надмножество всех выполнимых путей: *MOP*-решение собирает значения потоков данных как для всех выполнимых путей, так и для путей, которые не могут быть выполнены.
Следовательно, для всех B_k выполняется соотношение $MOP[B_k] \leq Ideal[B_k]$.

Если в анализируемой программе есть циклы, количество всех путей, рассматриваемых при построении *MOP*-решения, становится бесконечным. Поэтому построить *MOP*-решение, как правило, не удается (невозможно учесть бесконечное множество путей).

Можно доказать, что решение, получаемое итеративным алгоритмом, или *MFP*-решение удовлетворяет условию $MFP \leq MOP$. Можно также доказать, что если структура потока данных дистрибутивна, то для всех B $MFP[B] = MOP[B]$.

Таким образом, в случае дистрибутивной передаточной функции метод итераций позволяет с помощью сравнительно небольшого числа итераций получить решение сбором по всем путям.

Литература.

1. **В. А. Серебряков, М. П. Галочкин**. **Теория и реализация языков программирования,** учебное пособие, 2-е изд., доп. и испр. Москва : МЗ Пресс, 2006. - 350 с.

В Интернете оно есть в формате pdf.
http://trpl7.ru/t-books/_TRYAPBOOK_pdf.pdf

2. А. В. Ахо, М. С. Лам, Р. Сети, Дж. Д. Ульман. Компиляторы: принципы, технологии и инструменты, 2-е издание. М.: «И.Д. Вильямс», 2008(в конце 2014 года был выпущен дополнительный тираж)