МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Чувашский государственный университет имени И.Н. Ульянова»

Л.А. Павлов

СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ТРАНСЛЯЦИЯ

Учебное пособие

Чебоксары
2017

УДК 004.4’422(075.8)
ББК 3973.2р30(2)-25
П12

Рецензенты:
д-р техн наук, профессор *Г.П. Охоткин*;
канд. техн. наук *М.Ю. Харитонов*

Павлов Л.А.
П12 Синтаксически управляемая трансляция: учеб. пособие / Л.А. Павлов. Чебоксары: Изд-во Чуваш. ун-та, 2017. 60 с.

ISBN 978-5-7677-2472-7

Даны основы теории синтаксически управляемой трансляции языков программирования, рассмотрены вопросы построения атрибутной транслирующей грамматики для реализации семантического анализа и генерации промежуточного кода.

Для студентов IV курса факультета информатики и вычислительной техники направления подготовки бакалавров 09.03.01 «Информатика и вычислительная техника», а также других инженерных направлений для более глубокого изучения проблем информатики и вычислительной техники.

Ответственный редактор канд. техн. наук, доцент А.Л. Симаков

Утверждено Учебно-методическим советом университета

ISBN 978-5-7677-2472-7

УДК 004.4’422(075.8)
ББК 3973.2р30(2)-25

ПРЕДИСЛОВИЕ

Методы синтаксически управляемой трансляции рассматриваются в рамках дисциплины «Теория языков программирования и методы трансляции», изучаемой студентами, обучающимися по направлению подготовки бакалавров 09.03.01 «Информатика и вычислительная техника» (профиль «Программное обеспечение средств вычислительной техники и автоматизированных систем»). Целью изучения этих методов является получение теоретических знаний и практических навыков реализации таких фаз компиляции, как семантический анализ и генерация промежуточного кода.

Данный раздел изучается в 7 семестре и базируется на знаниях, полученных в процессе изучения предыдущих разделов дисциплины (основы теории формальных языков и грамматик, лексический и синтаксический анализ), а также таких дисциплин, как «Информатика», «Программирование», «Математическая логика и теория алгоритмов», «Дискретная математика», «Структуры и алгоритмы обработки данных» и др.

Знания, умения и навыки, полученные студентом в ходе изучения дисциплины «Теория языков программирования и методы трансляции», вполне достаточны для реализации трансляции простых языков программирования. Полученные знания будут полезны также для понимания особенностей разработки языков программирования, получения навыков работы со сложными структурами данных и алгоритмами.

В теоретической части учебного пособия в качестве основных моделей синтаксически управляемой трансляции рассматриваются синтаксически управляемые определения и схемы трансляции, особенности их применения в процессе нисходящего и восходящего синтаксического анализа. В прикладной части основное внимание уделяется реализации этих моделей для решения задач проверки типов и генерации промежуточного кода.

Излагаемый материал в основном опирается на терминологию и систему обозначений из фундаментальных работ А. Ахо, М. Лам, Р. Сети и Д. Ульмана [1; 2].

1. СИНТАКСИЧЕСКИ УПРАВЛЯЕМЫЕ ОПРЕДЕЛЕНИЯ

Синтаксически управляемое определение (СУО) представляет собой контекстно-свободную грамматику, дополненную *атрибутами* и *семантическими правилами*. Атрибуты связываются с символами грамматики (терминалами и нетерминалами) и могут иметь любой вид: число, тип данных, строку, ссылку на запись таблицы символов, адрес памяти и т.п. Атрибут a символа X грамматики обычно записывается в виде $X.a$. Семантические правила связываются с продукциями грамматики и описывают способ вычисления атрибутов. Обычно семантические правила записывают в виде $b := f(c_1, c_2, \ldots, c_k)$, где f – некоторая функция, записываемая часто как выражение; $b, c_1, c_2, \ldots, c_k$ – атрибуты, что говорит о том, что значение атрибута b зависит от значений атрибутов $c_1, c_2, \ldots, c_k$.

Если продукция в СУО содержит рекурсию, обычно нетерминал в левой части продукции записывают без индекса, а в правой части этот же нетерминал записывают с индексом, например, $R \rightarrow +TR_1$. Аналогично, если имеется несколько вхождений одного и того же символа грамматики в правую часть продукции, они записываются с различными индексами, например, $E \rightarrow E_1 + E_2$. Это связано с тем, что в дереве разбора таким символом грамматики помечены разные вершины, в которых один и тот же атрибут может иметь различные значения.

1.1. Типы атрибутов

Для нетерминалов выделяют два типа атрибутов: синтезируемые и наследуемые.

Синтезируемый атрибут определяется связанным с продукцией семантическим правилом для нетерминала A в левой части продукции. Значение синтезируемого атрибута вычисляется только с использованием атрибутов символов правой части продукции и атрибутов нетерминала A. Другими словами, если в дереве разбора нетерминалу A соответствует узел N, значение синтезируемого атрибута $A.s$ в узле N определяется только с использованием значений атрибутов в дочерних по отношению к N узлах и в самом узле N.

Наследуемый атрибут определяется связанным с продукцией семантическим правилом для нетерминала B в правой час-

ти продукции. Значение наследуемого атрибута вычисляется только с использованием атрибутов нетерминала A в левой части продукции и атрибутов символов правой части продукции (включая и атрибуты нетерминала B). Другими словами, если в дереве разбора нетерминалу B соответствует узел M, значение наследуемого атрибута $B.i$ в узле M определяется с использованием значений атрибутов в родительском по отношению к M узле, в самом узле M и братьях узла M.

Терминал может иметь только синтезируемый атрибут, который является атрибутом токена, передаваемого лексическим анализатором. Поэтому в СУО не должно быть семантических правил для вычисления значений атрибутов терминалов.

Иногда семантическое правило записывается как вызов процедуры или выполнение фрагмента программы (в таких случаях говорят, что правило имеет побочное действие). Побочное действие называется *контролируемым*, если оно не изменяет порядок вычисления атрибутов и не препятствует их вычислению. Правила с контролируемыми побочными действиями можно рассматривать как правила, определяющие значения фиктивных синтезируемых атрибутов нетерминала в левой части связанной продукции. СУО, в котором семантические правила не имеют побочных действий, называется *атрибутной* грамматикой.

СУО, которые включают только синтезируемые атрибуты, называются *S-атрибутными* СУО. В таком СУО семантическое правило вычисляет значение синтезируемого атрибута нетерминала в левой части продукции, используя значения синтезируемых атрибутов символов грамматики в правой части продукции.

Пример *S*-атрибутного СУО вычисления значения арифметического выражения представлен в табл. 1.

Таблица 1

S-атрибутное СУО вычисления арифметического выражения

Продукция	Семантические правила
1) $S \to E\perp$	$S.val := E.val$
2) $E \to E_1 + T$	$E.val := E_1.val + T.val$
3) $E \to T$	$E.val := T.val$
4) $T \to T_1 * F$	$T.val := T_1.val * F.val$
5) $T \to F$	$T.val := F.val$
6) $F \to (E)$	$F.val := E.val$
7) $F \to$ **num**	$F.val := GetVal$ (**num**.ns)

Каждый нетерминал имеет по одному синтезируемому атрибуту *val*, терминал **num** имеет синтезируемый атрибут *ns* (номер строки в таблице символов, предоставляемый лексическим анализатором в качестве атрибута токена числовой константы **num**). Функция *GetVal* (**num**.*ns*) возвращает значение числовой константы из соответствующей строки таблицы символов. Выполняемые правила для продукций очевидны и не требуют дополнительных пояснений. Результатом вычисления значения арифметического выражения является значение атрибута *S.val*, устанавливаемое в правиле *S.val* := *E.val* для первой продукции. Это правило можно заменить правилом с побочным действием, например, вида *Print*(*E.val*) для печати результатов вычислений.

Наглядное представление о процессе вычисления атрибутов дает дерево разбора, в котором в каждом узле показаны значения атрибутов. Такое дерево разбора называется *аннотированным*, а сам процесс вычисления значений атрибутов в узлах дерева разбора – *аннотированием* дерева разбора.

Аннотированное дерево разбора входной строки 1 + 2 * 3, построенное с применением правил из СУО (см. табл. 1), представлено на рис. 1.

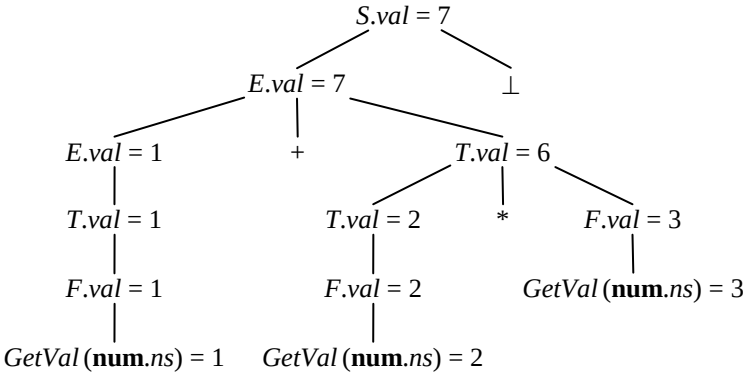


Рис. 1. Аннотированное дерево разбора выражения 1 + 2 * 3

Аннотирование осуществляется выполнением семантических правил в соответствии с обратным прохождением дерева

разбора и легко может быть реализовано в процессе восходящего синтаксического анализа, основанного на *LR*-грамматике.

Наследуемые атрибуты могут быть полезными, если возникает необходимость распространения информации от предков к потомкам. Несмотря на то, что всегда можно преобразовать СУО так, чтобы в нем использовались только синтезируемые атрибуты, тем не менее, более естественным часто является использование наследуемых атрибутов. Пример СУО с наследуемым атрибутом представлен в табл. 2, а аннотированное дерево разбора строки **char id$_1$, id$_2$** – на рис. 2.

Таблица 2

СУО с наследуемым атрибутом *L.inh*

Продукция	Семантические правила
1) $D \to T\ L$	*L.inh* := *T.type*
2) $T \to$ **int**	*T.type* := integer
3) $T \to$ **char**	*T.type* := char
4) $L \to L_1$ **, id**	L_1*.inh* := *L.inh* *AddType* (**id**.*ns*, *L.inh*)
5) $L \to$ **id**	*AddType* (**id**.*ns*, *L.inh*)

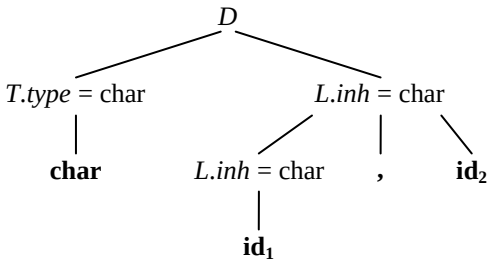


Рис. 2. Аннотированное дерево разбора строки **char id$_1$, id$_2$**

В этом СУО нетерминал *T* имеет синтезируемый атрибут *type*, значением которого является тип объявляемых данных, нетерминал *L* имеет наследуемый атрибут *inh* (от inherit – наследовать), процедура *AddType* записывает тип идентификатора в таблицу символов. После чтения ключевого слова **char** по правилу *T.type* := char, связанному с продукцией 3, устанавливается

значение атрибута *T.type* равным char. В правиле *L.inh* := *T.type*, связанному с продукцией 1, наследуемому атрибуту *L.inh* устанавливается значение char, что говорит о том, что все идентификаторы списка *L* будут иметь данный тип. Затем в нисходящем порядке это значение передается в два узла с нетерминалом *L* правого поддерева. В каждом таком узле выполняется также процедура *AddType*, которая записывает тип соответствующего идентификатора в таблицу символов.

1.2. Порядок выполнения семантических правил

Наиболее универсальной и мощной группой методов для определения порядка выполнения семантических правил являются так называемые *методы дерева разбора* [2]. В этих методах порядок выполнения определяется во время компиляции с помощью топологической сортировки [7] графа зависимостей, построенного по дереву разбора для входной строки. Методы не работают только в том случае, если построенный граф зависимостей будет иметь цикл (невозможно выполнить топологическую сортировку).

Граф зависимостей для дерева разбора представляет собой ориентированный граф, показывающий информационные связи между экземплярами атрибутов. Вершинам графа сопоставляются все экземпляры атрибутов во всех узлах дерева разбора. Если атрибут *a* зависит от атрибута *b*, то от вершины *b* ведет дуга к вершине *a*. Если в СУО имеются правила с побочными действиями, их необходимо привести к виду $b := f(c_1, c_2, \ldots, c_k)$ введением фиктивного синтезируемого атрибута *b* (при этом ни один атрибут не должен зависеть от *b*).

Граф зависимостей для дерева разбора (см. рис. 2) приведен на рис. 3 (ребра дерева разбора показаны пунктирными линиями). Вершины графа помечены числами, показывающими результаты одной из его топологических сортировок. Вершины 6 и 7 графа соответствуют фиктивным синтезируемым атрибутам, введенным для правила с побочными действиями *AddType* (**id**.*ns*, *L.inh*). Результатом топологической сортировки является следующий порядок выполнения семантических правил (a_i означает атрибут для вершины с номером *i*):

a_3 := char;
$a_4 := a_3$;
$a_5 := a_4$;
AddType (**id**$_1$.*ns*, a_5);
AddType (**id**$_2$.*ns*, a_4).

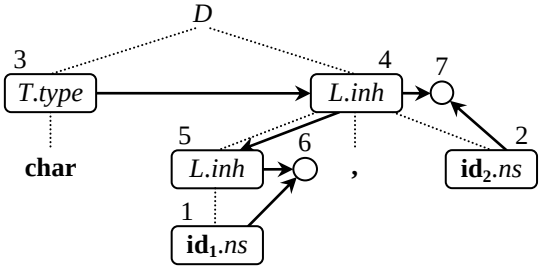


Рис. 3. Граф зависимостей для дерева разбора на рис. 2

Вторая группа методов для определения порядка выполнения семантических правил основана на анализе семантических правил, связанных с продукциями. По результатам такого анализа предопределяется порядок вычисления атрибутов в процессе разработки компилятора.

Третью группу методов для определения порядка выполнения семантических правил составляют так называемые *игнорирующие методы*. В этих методах выбор порядка производится без рассмотрения семантических правил. В частности, если трансляция реализуется в процессе синтаксического анализа, то порядок выполнения правил определяется методом синтаксического анализа независимо от семантических правил. Очевидно, что эти методы применимы для ограниченных классов СУО. К таким классам относятся *S*-атрибутные и более общие *L*-атрибутные СУО.

В отличие от методов дерева разбора, где в процессе компиляции требуется явное построение дерева разбора и на его основе – графа зависимостей, методы, основанные на правилах, и игнорирующие методы этого не требуют. Поэтому они могут быть более эффективными как с точки зрения времени выполнения, так и объема используемой памяти.

1.3. *L*-атрибутные СУО

СУО называется *L-атрибутным*, если каждый атрибут является либо синтезируемым, либо наследуемым. При этом наследуемый атрибут символа X_j, $1 \le j \le n$ из правой части продукции $A \to X_1X_2...X_n$, вычисляемый с помощью правила, связанного с данной продукцией, зависит только от наследуемых атрибутов нетерминала A, наследуемых или синтезируемых атрибутов символов $X_1, X_2, ..., X_{j-1}$, расположенных слева от X_j, наследуемых атрибутов самого X_j (при условии, что зависимости не являются циклическими). Поскольку ограничения касаются только наследуемых атрибутов, каждое *S*-атрибутное СУО является истинным подмножеством *L*-атрибутных СУО. *L*-атрибутное СУО легко согласуется с нисходящими (*LL*-разбор) и восходящими (*LR*-разбор) методами синтаксического анализа.

Все рассмотренные выше в качестве примеров СУО (см. табл. 1 и 2) являются *L*-атрибутными. Пример СУО, не являющегося *L*-атрибутным, представлен в табл. 3.

Таблица 3

Не *L*-атрибутное СУО

Продукция	Семантические правила
1) $S \to A\ B$	$B.inh := f(A.inh, A.syn)$ $S.syn := g(A.syn, B.syn)$
2) $S \to C\ D$	$C.inh := h(S.inh, D.inh)$ $S.syn := g(C.syn, D.syn)$

Для продукции 1 правила являются корректными, поскольку наследуемый атрибут *B.inh* зависит от атрибутов символа *A*, расположенного слева от *B*. Первое правило продукции 2 не позволяет СУО быть *L*-атрибутным, поскольку атрибут *C.inh* зависит от атрибута *D.inh* символа *D*, находящегося справа от *C*. Второе правило этой продукции вполне корректно.

Таким образом, в *L*-атрибутных СУО все атрибуты могут быть вычислены в соответствии со следующим прохождением дерева разбора (само дерево явно не строится). При прохожде-

нии вниз по дереву при первом посещении узла вычисляются наследуемые атрибуты этого узла. Для продукции $A \to X_1X_2...X_n$ поддеревья с корнями X_1, X_2, …, X_n обходят слева направо. После завершения обхода всех этих поддеревьев при возврате на уровень вверх, непосредственно перед тем как будет покинут узел A, вычисляются его синтезируемые атрибуты. Другими словами, наследуемые атрибуты вычисляются в соответствии с прямым прохождением дерева разбора, а синтезируемые атрибуты – в соответствии с обратным прохождением. Иногда эти прохождения объединяют в одно так называемое *прохождение в глубину* (тогда прямое и обратное прохождения рассматриваются как частные случаи этого общего прохождения). Тогда можно сказать, что все атрибуты L-атрибутного СУО могут быть вычислены в соответствии с прохождением в глубину дерева разбора, которое определяет порядок вычисления атрибутов согласно рекурсивной процедуре *DepthFirst*, где v и w – узлы дерева разбора, *Sons* (v) – множество сыновей узла v.

```
procedure DepthFirst (v)
begin
    for w ∈ Sons (v) слева направо do
    begin
        Вычислить наследуемые атрибуты w
        DepthFirst (w)
    end
    Вычислить синтезируемые атрибуты v
end
```

2. СХЕМЫ ТРАНСЛЯЦИИ

Синтаксически управляемая схема трансляции (СУТ) представляет собой контекстно-свободную грамматику, дополненную программными фрагментами (*семантическими действиями*), вставленными в правые части продукций. Действия обычно заключаются в фигурные скобки и могут располагаться в любой позиции правой части продукции.

Отличие СУТ от СУО заключается в том, что в СУТ явно определен порядок вычисления семантических правил, задаваемый порядком обхода дерева разбора, который, в свою очередь, определяется методом синтаксического анализа. Кроме того, семантические действия в СУТ обычно более детализированы, чем семантические правила в СУО.

2.1. Преобразование *L*-атрибутного СУО в СУТ

Преобразование *L*-атрибутного СУО в СУТ для его реализации в процессе синтаксического анализа выполняется в соответствии со следующими правилами [1]:

а) действие, которое вычисляет наследуемый атрибут нетерминала *A*, необходимо поместить непосредственно перед вхождением *A* в правую часть продукции;

б) действие, вычисляющее синтезируемый атрибут нетерминала в левой части продукции, следует разместить в конце правой части продукции.

Построим СУТ для *L*-атрибутного СУО (см. табл. 2):

1) $D \to T$ {*L.inh* := *T.type*} L

2) $T \to$ **int** {*T.type* := integer}

3) $T \to$ **char** {*T.type* := char}

4) $L \to$ {L_1.*inh* := *L.inh*} L_1 **,** **id** {*AddType* (**id**.*ns, L.inh*)}

5) $L \to$ **id** {*AddType* (**id**.*ns, L.inh*)}.

В продукции 1 действие {*L.inh* := *T.type*} размещено непосредственно перед *L*, поскольку в нем вычисляется наследуемый атрибут *L.inh* нетерминала *L*. По аналогичной причине в продукции 4 действие {L_1.*inh* := *L.inh*} помещено перед L_1. Остальные действия вычисляют синтезируемый атрибут *T.type* или яв-

ляются контролируемым побочным действием *AddType*, поэтому они размещены в конце правых частей продукций.

Когда семантические действия включают в себя множество различных операций, можно в СУТ для компактности записи использовать вместо действий их обозначения с соответствующей расшифровкой в виде алгоритмов выполнения действий. Обозначим действия в рассматриваемой СУТ следующим образом:

*A*1: *L.inh* := *T.type*

*A*2: *T.type* := integer

*A*3: *T.type* := char

*A*4: L_1*.inh* := *L.inh*

*A*5: *AddType* (**id**.*ns*, *L.inh*).

Тогда СУТ можно записать в виде

1) $D \to T$ {*A*1} L

2) $T \to$ **int** {*A*2}

3) $T \to$ **char** {*A*3}

4) $L \to$ {*A*4} L_1 **, id** {*A*5}

5) $L \to$ **id** {*A*5}.

Для СУТ символы действий включаются в дерево разбора как его узлы. Действия выполняются в порядке, соответствующем прохождению дерева в глубину. При необходимости дерево разбора можно аннотировать. Для рассматриваемой СУТ дерево разбора строки **char id$_1$, id$_2$** представлено на рис. 4.

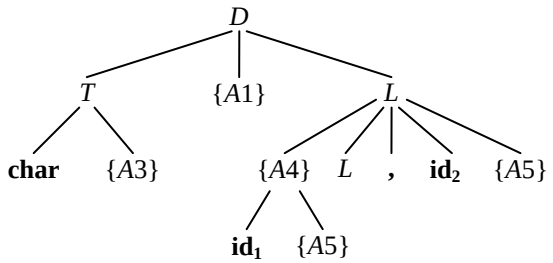


Рис. 4. Дерево разбора строки **char id$_1$, id$_2$** для СУТ

2.2. Память для хранения атрибутов

Важным является назначение памяти для хранения значений атрибутов в процессе трансляции. При заданном порядке вычисления атрибутов (зависит от порядка обхода дерева разбора) *время жизни* атрибута начинается, когда атрибут впервые вычисляется, и заканчивается, когда вычислены все атрибуты, зависящие от него. Для экономии памяти значения атрибутов сохраняются только на протяжении их времени жизни. Значения атрибутов помещаются в стек. Число и размер атрибутов символов грамматики зафиксировано, поэтому на каждом шаге процесса синтаксического анализа известно, в какой позиции стека находится интересующий атрибут. Можно разместить атрибуты в стеке синтаксического анализатора (расширив соответствующим образом структуру элемента стека) или использовать специальный стек или несколько стеков (например, отдельные стеки для синтезируемых и наследуемых атрибутов) для хранения значений атрибутов в течение времени их жизни.

Время жизни атрибута достаточно просто определяется по грамматике. Пусть имеется специальный стек для хранения атрибутов. Рассмотрим прохождение в глубину дерева разбора в соответствии с процедурой *DepthFirst* (см. подразд. 1.3). Обозначим наследуемые и синтезируемые атрибуты символа X грамматики как $I(X)$ и $S(X)$ соответственно. Для продукции $A \to BC$ процесс обхода начинается в узле A. К этому моменту в родительском по отношению к A узле вычислены наследуемые атрибуты $I(A)$, которые находятся в верхней части стека. Вычисляются и заносятся в стек значения наследуемых атрибутов $I(B)$. После завершения обхода поддерева с корнем B в стеке над наследуемыми атрибутами $I(B)$ будут находиться синтезируемые атрибуты $S(B)$. Аналогично процесс повторяется для поддерева с корнем C, т. е. в стек заносятся наследуемые атрибуты $I(C)$ и после завершения обхода поддерева с корнем C в стек будут занесены синтезируемые атрибуты $S(C)$. Таким образом, к моменту возврата к узлу A в стеке будут находиться значения атрибутов $I(A)$, $I(B)$, $S(B)$, $I(C)$, $S(C)$. Все атрибуты, необходимые для

вычисления синтезируемых атрибутов A, в данный момент находятся в верхней части стека, их число и позиции в стеке известны. После вычисления синтезируемых атрибутов A время жизни атрибутов $I(B)$, $S(B)$, $I(C)$, $S(C)$ заканчивается, поэтому обход поддерева с корнем A завершается со стеком, содержащим в верхней части $I(A)$, $S(A)$.

Рассмотрим хранение атрибутов при обходе поддерева для продукции $L \to \{L_1.inh := L.inh\}\ L_1$ **,** **id** $\{AddType(\mathbf{id}.ns, L.inh)\}$ из СУТ из подразд. 2.1. Изменение содержимого стека атрибутов в процессе обхода узлов дерева разбора представлено на рис. 5. Слева от узла показано содержимое стека до посещения узла, а справа – после посещения.

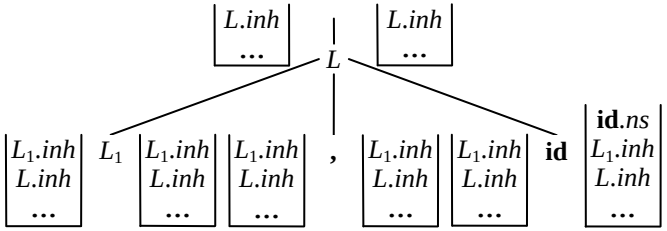


Рис. 5. Содержимое стека атрибутов
в процессе обхода узлов

Непосредственно перед посещением узла L в вершине стека находится значение его наследуемого атрибута $L.inh$. До посещения узла L_1 вычисляется его наследуемый атрибут $L_1.inh$ согласно действию $L_1.inh := L.inh$ и заносится в стек. После обхода поддерева, корнем которого является L_1, следовало бы занести значения его синтезируемых атрибутов, но, поскольку их нет, содержимое стека остается неизменным. Терминал *","* не имеет атрибутов, поэтому содержимое стека не изменяется. Терминал **id** имеет только синтезируемый атрибут **id**.*ns*, поэтому он добавляется в стек после посещения этого узла. После обхода поддерева с корнем L выполняется действие $AddType(\mathbf{id}.ns, L.inh)$, при этом время жизни атрибутов $L_1.inh$ и **id**.*ns* завершается (они исключаются из стека). Поскольку у L нет синтезируемых атрибутов, в вершине стека остается наследуемый атрибут $L.inh$.

В СУТ время жизни некоторых атрибутов может завершиться на более ранних этапах. Поэтому нет необходимости в их сохранении в стеке после вычисления значений зависящих от них атрибутов. В нашем примере время жизни атрибута $L_1.inh$ завершается после обхода поддерева с корнем L_1. Это объясняется тем, что от атрибута $L_1.inh$ не зависят значения атрибутов символов продукции $L \rightarrow L_1$ **, id**, стоящих справа от L_1. Поэтому нет необходимости в его сохранении в стеке после обхода поддерева с корнем L_1.

Заметим, что атрибут **id**.*ns* заносится в стек и сразу же удаляется из стека. Поэтому можно обойтись без занесения этого атрибута в стек.

Если некоторый атрибут b определяется правилом копирования вида $b := c$, а значение c находится в вершине стека, то в ряде случаев можно обойтись без размещения в стеке копии c. В нашем примере таким правилом является $L_1.inh := L.inh$, т.е. можно обойтись без занесения в стек атрибута $L_1.inh$.

Изменение содержимого стека атрибутов в процессе обхода узлов дерева разбора, учитывающее досрочное завершение времени жизни атрибутов и наличие правил копирования для той же продукции, что и на рис. 5, представлено на рис. 6.

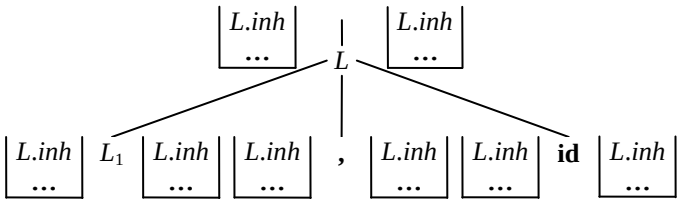


Рис. 6. Содержимое стека атрибутов с учетом досрочного завершения времени жизни и наличия правил копирования

Проведя подобный анализ для всех продукций СУТ, можно детализировать действия, связанные с вычислением или использованием атрибутов, соответствующими операциями со стеком.

После детализации действий соответствующими операциями со стеком (используется отдельный стек атрибутов) СУТ из подразд. 2.1 примет следующий вид:

1) $D \rightarrow T\ L$ {*Pop* (*St*)}

2) $T \rightarrow$ **int** {*Push* (integer, *St*)}

3) $T \rightarrow$ **char** {*Push* (char, *St*)}

4) $L \rightarrow L_1$ **, id** {*AddType* (**id**.*ns*, *StackTop* (*St*))}

5) $L \rightarrow$ **id** {*AddType* (**id**.*ns*, *StackTop* (*St*))}.

Здесь операция *Push* (*x*, *St*) размещает значение *x* в стеке *St*, функция *Pop* (*St*) исключает элемент из вершины стека и возвращает его значение, функция *StackTop* (*St*) возвращает значение элемента из вершины стека *St* без его исключения.

В конце правой части первой продукции добавлено действие *Pop*(*St*), которое после завершения разбора входной строки исключает из стека единственный оставшийся элемент и делает стек пустым. Заметим, что в исходной СУТ такого действия не было. Если нет требования, чтобы после разбора входной строки стек атрибутов должен быть пустым, это действие можно не добавлять. Это действие не нужно также в случае использования для хранения атрибутов стека синтаксического анализатора, использующего метод синтаксического анализа, для которого одним из критериев успешного завершения разбора является пустота стека анализатора.

Использование СУТ для реализации СУО в процессе синтаксического анализа имеет ряд особенностей, зависящих от применяемого метода синтаксического анализа.

3. ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

3.1. Реализация *S*-атрибутных СУО

Наиболее простым является использование СУТ для реализации *S*-атрибутного СУО в процессе восходящего синтаксического анализа. В этом случае лежащая в основе СУО грамматика должна принадлежать классу *LR*. Преобразование *S*-атрибутного СУО в СУТ заключается в размещении действий в конце продукции. Эти действия выполняются в процессе свертки правой части продукции в нетерминал из левой части. СУТ со всеми действиями, расположенными в конце правой части продукции, называются *постфиксными*.

Пусть дано *S*-атрибутное СУО вычисления простого арифметического выражения, представленное в табл. 4.

Таблица 4

S-атрибутное СУО вычисления простого арифметического выражения

Продукция	Семантические правила
1) $S \to E\perp$	*Print* (*E.val*)
2) $E \to E_1 + T$	$E.val := E_1.val + T.val$
3) $E \to T$	$E.val := T.val$
4) $T \to T_1 * F$	$T.val := T_1.val * F.val$
5) $T \to F$	$T.val := F.val$
6) $F \to (E)$	$F.val := E.val$
7) $F \to$ **num**	*F.val* := *GetVal* (**num**.*ns*)

Каждый нетерминал имеет по одному синтезируемому атрибуту *val*, терминал **num** имеет синтезируемый атрибут *ns* (номер строки в таблице символов, предоставляемый лексическим анализатором в качестве атрибута токена числовой константы **num**). Функция *GetVal* (**num**.*ns*) возвращает значение числовой константы из соответствующей строки таблицы символов. Семантическое правило *Print*(*E.val*) первой продукции выводит результат вычисления выражения.

Постфиксная СУТ, реализующая данное СУО, имеет следующий вид:

$S \to E\bot$ {*Print* (*E.val*)}

$E \to E_1 + T$ {*E.val* := E_1.*val* + *T.val*}

$E \to T$ {*E.val* := *T.val*}

$T \to T_1 * F$ {*T.val* := T_1.*val* * *F.val*}

$T \to F$ {*T.val* := *F.val*}

$F \to (E)$ {*F.val* := *E.val*}

$F \to$ **num** {*F.val* := *GetVal* (**num**.*ns*)}.

Атрибут символа грамматики можно разместить в стеке состояний *LR*-анализатора вместе с состоянием, представляющим этот символ (модифицировав соответствующим образом структуру элемента стека), или использовать специальный стек для хранения значений атрибутов.

После детализации действий соответствующими операциями со стеком (используется отдельный стек атрибутов) постфиксная СУТ примет следующий вид:

$S \to E\bot$ {*Print* (*Pop* (*St*))}

$E \to E_1 + T$ {t_2 := *Pop* (*St*); t_1 := *Pop* (*St*); *Push* (t_1 + t_2, *St*)}

$E \to T$

$T \to T_1 * F$ {t_2 := *Pop* (*St*); t_1 := *Pop* (*St*); *Push* (t_1 * t_2, *St*)}

$T \to F$

$F \to (E)$

$F \to$ **num** {*Push* (*GetVal* (**num**.*ns*), *St*)}.

Здесь операция *Push* (*x*, *St*) размещает значение *x* в стеке *St*, функция *Pop* (*St*) исключает элемент из вершины стека и возвращает его значение; t_1 и t_2 – промежуточные переменные.

Примеры реализации постфиксных СУТ с использованием стека синтаксического анализатора можно посмотреть в [1; 2].

3.2. *L*-атрибутные СУО на основе *LL*-грамматики

L-атрибутные СУО на основе *LL*-грамматики обычно ориентированы на нисходящий синтаксический анализ. Однако их можно адаптировать и для реализации в процессе восходящего

синтаксического анализа, поскольку *LL*-грамматики являются истинным подмножеством *LR*-грамматик.

Поскольку для восходящей трансляции все действия должны находиться в конце продукции, построение СУТ для реализации *L*-атрибутного СУО на основе *LL*-грамматики в процессе *LR*-разбора предполагает следующее [1]:

1. Строится СУТ в соответствии с правилами из подразд. 2.1, согласно которым действия по вычислению наследуемых атрибутов нетерминала вставляются перед этим нетерминалом, а действия по вычислению синтезируемых атрибутов размещаются в конце продукции.

2. В грамматику добавляются нетерминалы-маркеры вместо каждого вставленного действия. Каждое вставленное действие заменяется отдельным маркером *M*; для каждого маркера *M* существует только одна продукция вида $M \to \varepsilon$.

3. Модифицируется действие *a*, заменяемое маркером *M* в некоторой продукции $A \to \alpha\{a\}\beta$, и с продукцией $M \to \varepsilon$ связывается действие *a*':

а) копирующее в качестве наследуемых атрибутов *M* любые атрибуты *A* или символов из α, которые требуются действию *a*;

б) вычисляющее атрибуты таким же способом, что и *a*, но делающее эти атрибуты синтезируемыми атрибутами *M*.

Заметим, что вместо наследуемых атрибутов *M* можно использовать промежуточные переменные, включив в действие *a*' побочные действия, копирующие в эти переменные необходимые атрибуты. Это связано с тем, что время их жизни завершается сразу же после вычисления синтезируемых атрибутов *M*.

Пусть имеется продукция СУТ на основе *LL*-грамматики: $A \to B\ \{C.inh := f(A.inh, B.syn)\}\ C$.

В действии наследуемый атрибут *C.inh* вычисляется как функция наследуемого атрибута *A.inh* и синтезируемого атрибута *B.syn*. Добавим маркер *M* с синтезируемым атрибутом *M.syn* (копия *C.inh*). Тогда фрагмент СУТ для этой продукции примет следующий вид (t_1 и t_2 – промежуточные переменные):

$A \to B\ M\ C$

$M \to \varepsilon\ \{t_1 := A.inh;\ t_2 := B.syn;\ M.syn := f(t_1, t_2)\}$.

На первый взгляд подобное преобразование некорректно, поскольку действие в продукции $M \to \varepsilon$ должно иметь доступ к атрибутам, связанным с символами грамматики, которые отсутствуют в данной продукции. Однако если реализовать действия с использованием стека, необходимые атрибуты всегда будут доступны с помощью известных позиций в стеке. При этом не понадобятся наследуемые атрибуты M (или промежуточные переменные). Поэтому предварительно лучше детализировать действия с помощью стековых операций (этим как бы снимается зависимость атрибутов от символов продукций), а затем выполнить рассмотренное преобразование.

Рассмотрим СУТ для вычисления арифметического выражения, основанную на $LL(1)$-грамматике (о преобразовании грамматики СУО в $LL(1)$-форму см. разд. 4):

$S \to E\perp\ \{Print\,(E.val)\}$

$E \to T\ \{X.inh := T.val\}\ X\ \{E.val := X.syn\}$

$X \to +\ T\ \{X_1.inh := X.inh + T.val\}\ X_1\ \{X.syn := X_1.syn\}$

$X \to \varepsilon\ \{X.syn := X.inh\}$

$T \to F\ \{Y.inh := F.val\}\ Y\ \{T.val := Y.syn\}$

$Y \to *\ F\ \{Y_1.inh := Y.inh + F.val\}\ Y_1\ \{Y.syn := Y_1.syn\}$

$Y \to \varepsilon\ \{Y.syn := Y.inh\}$

$F \to (E)\ \{F.val := E.val\}$

$F \to \mathbf{num}\ \{F.val := GetVal\,(\mathbf{num}.ns)\}$.

После детализации действий соответствующими операциями со стеком (используется стек атрибутов) с учетом предотвращения хранения в стеке копий значений атрибутов для действий копирования СУТ примет следующий вид:

$S \to E\perp\ \{Print\,(Pop\,(St))\}$

$E \to T\ X$

$X \to +\ T\ \{t_2 := Pop\,(St);\ t_1 := Pop\,(St);\ Push\,(t_1 + t_2, St)\}\ X_1$

$X \to \varepsilon$

$T \to F\ Y$

$Y \to *\ F\ \{t_2 := Pop\,(St);\ t_1 := Pop\,(St);\ Push\,(t_1 * t_2, St)\}\ Y_1$

$Y \to \varepsilon$

$F \to (E)$

$F \to \mathbf{num}\ \{Push\,(GetVal\,(\mathbf{num}.ns),\ St)\}$.

Для вставленных действий добавим маркеры M и N и продукции $M \to \varepsilon$ и $N \to \varepsilon$ с соответствующими действиями:

$S \to E\perp\ \{Print\,(Pop\,(St))\}$

$E \to T\ X$

$X \to +\ T\ M\ X_1$

$X \to \varepsilon$

$T \to F\ Y$

$Y \to *\ F\ N\ Y_1$

$Y \to \varepsilon$

$F \to (E)$

$F \to \mathbf{num}\ \{Push\,(GetVal\,(\mathbf{num}.ns),\ St)\}$.

$M \to \varepsilon\ \{t_2 := Pop\,(St);\ t_1 := Pop\,(St);\ Push\,(t_1 + t_2,\ St)\}$

$N \to \varepsilon\ \{t_2 := Pop\,(St);\ t_1 := Pop\,(St);\ Push\,(t_1 * t_2,\ St)\}$.

В результате полученная СУТ может быть использована для реализации *L*-атрибутного СУО в процессе восходящего синтаксического анализа.

3.3. *L*-атрибутные СУО на основе *LR*-грамматики

Очень важно обратить внимание на то, что в общем случае *L*-атрибутное СУО на основе *LR*-грамматики, имеющее наследуемые атрибуты, нельзя реализовать в процессе восходящего синтаксического анализа [1]. Это не затрагивает *S*-атрибутные СУО (хотя они и являются истинным подмножеством *L*-атрибутных СУО), поскольку у них нет наследуемых атрибутов. Неформально это объясняется следующим образом.

Пусть в *LR*-грамматике имеется продукция $A \to BC$, в которой необходимо вычислить наследуемый атрибут *B.inh*, зависящий от наследуемого атрибута *A.inh*. К моменту свертки к *B* (после обхода всех сыновей *B*) во входном потоке еще не обработана подстрока, порождаемая символом *C*. Поэтому еще нет гарантии, что основой для свертки будет именно продукция $A \to B\ C$. Следовательно, и нет гарантии, что для вычисления атрибута *B.inh* следует использовать правило, свя-

занное именно с данной продукцией, а не с какой-то другой. Даже если подождать свертки к *C*, чтобы убедиться в том, что будет выполнена свертка *BC* к *A*, то все равно неизвестно значение атрибута *A.inh* – в какой именно правой части какой продукции вычисляется *A.inh*. Такой процесс без возможности принять решение может быть продолжен до тех пор, пока не будет выполнен анализ всей входной строки. Таким образом, сначала придется построить дерево разбора, а на следующем проходе выполнить семантические правила для трансляции (возможно, через построение графа зависимостей). А это уже не является реализацией СУТ именно в процессе синтаксического анализа.

Даже при отсутствии наследуемых атрибутов не все СУТ могут быть реализованы в процессе синтаксического анализа. Для примера рассмотрим СУТ для преобразования инфиксных выражений в префиксную форму:

$S \to E\perp$

$E \to \{Print('+')\}\ E_1 + T$

$E \to T$

$T \to \{Print('*')\}\ T_1 * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{num}\ \{Print(GetLex(\mathbf{num}.ns))\}$.

Здесь функция $GetLex(\mathbf{num}.ns)$ возвращает лексему числовой константы из соответствующей строки таблицы символов.

В этой СУТ нет атрибутов, используются только побочные действия. Использование нетерминалов-маркеров M и N для действий соответственно $\{Print('+')\}$ и $\{Print('*')\}$ и добавление продукций $M \to \varepsilon\ \{Print('+')\}$ и $N \to \varepsilon\ \{Print('*')\}$ приведет к тому, что при чтении входного символа **num** возникнет неразрешимый конфликт между сверткой по $M \to \varepsilon$ и по $N \to \varepsilon$.

Это связано с тем, что, например, для выполнения $Print('+')$ надо знать, что будет выполнена сверка $E_1 + T$ к E. Таким образом, имеет место неконтролируемое побочное действие, по-

скольку оно зависит от символов, стоящих справа от него, т. е. данная СУТ по сути не является *L*-атрибутной.

Для решения задачи преобразования инфиксной формы выражения в префиксную можно построить *S*-атрибутное СУО (с последующим преобразованием в постфиксную СУТ) на основе той же *LR*-грамматики, добавив соответствующие синтезируемые атрибуты и семантические правила их вычисления (табл. 5).

Таблица 5

S-атрибутное СУО преобразования инфиксной формы выражения в префиксную

Продукция	Семантические правила
1) $S \to E\perp$	*Print* (*E.str*)
2) $E \to E_1 + T$	$E.str := '+' \parallel E_1.str \parallel T.str$
3) $E \to T$	$E.str := T.str$
4) $T \to T_1 * F$	$T.str := '*' \parallel T_1.str \parallel F.str$
5) $T \to F$	$T.str := F.str$
6) $F \to (E)$	$F.str := E.str$
7) $F \to$ **num**	*F.str* := *GetLex* (**num**.*ns*)

В синтезируемом атрибуте *str* (типа строки, при реализации лучше использовать указатель на строку) формируется соответствующая строка префиксной формы. Символ || обозначает операцию конкатенации строк.

4. НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Использование СУТ для реализации *L*-атрибутного СУО в процессе нисходящего синтаксического анализа требует, чтобы лежащая в основе СУО грамматика принадлежала классу *LL*. Преобразование *L*-атрибутного СУО в СУТ для его реализации в процессе *LL*-разбора выполняется в соответствии с правилами, из подразд. 2.1, согласно которым действия по вычислению наследуемых атрибутов нетерминала вставляются перед этим нетерминалом, а действия по вычислению синтезируемых атрибутов размещаются в конце продукции.

Если грамматика не относится к классу *LL*, ее следует преобразовать. Наличие атрибутов вносит некоторые особенности в методы устранения левой рекурсии. Если СУТ содержит только действия, не связанные с вычислением атрибутов, то применим стандартный алгоритм устранения левой рекурсии [1; 6], при использовании которого действия рассматриваются так, как если бы это были терминалы. Если же действия вычисляют значения атрибутов, устранение левой рекурсии возможно, если СУТ является постфиксной.

Рассмотрим общую схему для случая одной рекурсивной продукции и одной нерекурсивной продукции [1]:

$A \to A_1\, B\ \{A.a := g(A_1.a, B.b)\}$

$A \to C\ \{A.a := f(C.c)\}$.

Согласно общему правилу преобразования (без учета атрибутов) должны получить грамматику

$A \to C\, X$

$X \to B\, X \mid \varepsilon$.

С учетом действий, связанных с вычислением атрибутов, получается следующая СУТ:

$A \to C\ \{X.i := f(C.c)\}\ X\ \{A.a := X.s\}$

$X \to B\ \{\ X_1.i := g(X.i, B.b)\}\ X_1\ \{X.s := X_1.s\}$

$X \to \varepsilon\ \{X.s := X.i\}$.

Добавленный нетерминал X имеет наследуемый атрибут $X.i$ и синтезируемый атрибут $X.s$.

Для иллюстрации устраним левую рекурсию в грамматике постфиксной СУТ:

$S \to E\perp\ \{Print(E.val)\}$

$E \to E_1 + T\ \{E.val := E_1.val + T.val\}$

$E \to T\ \{E.val := T.val\}$

$T \to (E)$ $\{T.val := E.val\}$

$T \to$ **num** $\{T.val := GetVal(\textbf{num}.ns)\}$.

В соответствии с рассмотренным выше правилом получится следующая СУТ:

$S \to E\perp$ $\{Print(E.val)\}$

$E \to T$ $\{X.i := T.val\}$ X $\{E.val := X.s\}$

$X \to +\ T$ $\{X_1.i := X.i + T.val\}$ X_1 $\{X.s := X_1.s\}$

$X \to \varepsilon$ $\{X.s := X.i\}$

$T \to (E)$ $\{T.val := E.val\}$

$T \to$ **num** $\{T.val := GetVal(\textbf{num}.ns)\}$.

Для постфиксной СУТ с детализацией операций со стеком

$S \to E\perp$ $\{Print(Pop(St))\}$

$E \to E_1 + T$ $\{t_2 := Pop(St);\ t_1 := Pop(St);\ Push(t_1 + t_2, St)\}$

$E \to T$

$T \to (E)$

$T \to$ **num** $\{Push(GetVal(\textbf{num}.ns), St)\}$

применим стандартный алгоритм устранения левой рекурсии, когда действия рассматриваются как терминалы:

$S \to E\perp$ $\{Print(Pop(St))\}$

$E \to T\ X$

$X \to +\ T$ $\{t_2 := Pop(St);\ t_1 := Pop(St);\ Push(t_1 + t_2, St)\}$ X_1

$X \to \varepsilon$

$T \to (E)$

$T \to$ **num** $\{Push(GetVal(\textbf{num}.ns), St)\}$.

Стек *LL*-анализатора наряду с элементами, представляющими терминалы и нетерминалы, должен хранить также элементы действий. Элемент действий представляет собой указатель на выполняемый код, который инициирует запуск выполнения семантического действия в момент, когда элемент действия окажется в вершине стека в процессе *LL*-разбора.

Если для хранения атрибутов используется стек синтаксического анализатора, то синтезируемые атрибуты нетерминала размещаются в стеке под элементом, представляющим этот нетерминал, а наследуемые атрибуты нетерминала хранятся в стеке вместе с этим нетерминалом. На практике можно использовать специальный стек (стеки) для хранения значений атрибутов, как это рассмотрено в подразд. 2.2. Более подробное изложение вопросов хранения атрибутов с иллюстрациями на примерах можно найти в работе А. Ахо и др. [1].

5. СЕМАНТИЧЕСКИЙ АНАЛИЗ

5.1. Основные функции семантического анализа

Фаза семантического анализа предназначена для проверки исходной транслируемой программы на соответствие семантическим соглашениям. Такая проверка называется *статической* (существует также *динамическая* проверка, которая выполняется в процессе выполнения скомпилированной целевой программы).

Типичный перечень семантических соглашений для многих языков программирования [3; 4]:

1. Никакой идентификатор объекта (переменная и т.п.) в программном блоке не должен быть объявлен более одного раза.

2. Определяющим вхождениям идентификатора должны соответствовать их использующие вхождения. Здесь под *определяющим* вхождением понимается вхождение идентификатора в конструкцию, объявляющую этот идентификатор, а *использующим* – остальные вхождения. Например, для любого использующего вхождения должно быть определяющее вхождение, причем идентификатор должен быть объявлен до его использования. Если для определяющего вхождения нет использующего вхождения (не является семантической ошибкой), то вполне естественно сформировать соответствующее предупреждение.

3. Соглашение о соответствии типов объектов, входящих в синтаксическую конструкцию языка программирования, например совместимости типов операндов операции, соответствии типов формальных и фактических параметров процедур.

4. Соглашения, определяющие различные количественные ограничения, например глубину вложенности блоков, ограниченность размерности массивов.

Первые три вида семантических соглашений связаны с *проверкой типов*.

Кроме проверки семантических соглашений в функции семантического анализа включают также *распределение памяти* для размещения объявленных объектов (простых переменных, массивов и других структур данных). По типу объекта можно определить объем памяти, необходимый для его хранения. Во время трансляции эти величины могут использоваться для назначения каждому объекту относительного адреса. Тип и относительный адрес заносятся в запись таблицы символов, соответ-

ствующую объекту. Данные переменной длины, такие как строки, или данные, размер которых невозможно определить до начала работы программы (например, динамические массивы), обрабатываются путем резервирования известного фиксированного объема памяти для указателя на данные. При размещении объектов в памяти следует учитывать также необходимость выравнивания адресов. Например, команда сложения целых чисел может требовать, чтобы числа были размещены в определенных позициях памяти, например по адресу, кратному 4.

5.2. Выражения типа

Типы имеют структуру, которую можно представить с использованием выражений типов. *Выражение типа* представляет собой либо фундаментальный (базовый) тип, либо образуется путем применения оператора, называющегося *конструктором типа*, к выражению типа. Множество фундаментальных типов и конструкторов зависит от конкретного языка. Воспользуемся следующим определением выражения типа [1]:

а) фундаментальный тип является выражением типа; типичные фундаментальные типы языка включают предопределенные типы (например, *integer*, *real*, *Boolean*, *char*) и, при необходимости, специальный тип *void*, означающий отсутствие типа;

б) имя типа является выражением типа;

в) выражение типа может быть образовано путем применения конструктора типа к выражению типа;

г) выражение типа может содержать переменные, значениями которых являются выражения типов.

Различные подтипы фундаментальных типов можно рассматривать также как фундаментальные, например тип диапазон вида 1..100 является ограниченным подтипом целого типа. Однако ничто не мешает использовать для этого специальный конструктор типа, отнеся его к создаваемым типам.

Конструктор типа предназначен для создания нового типа на основе фундаментального или другого создаваемого типа. Примеры конструкторов типа (T, T_1, T_2 – выражения типа):

а) *array* (I, T) – определяет массив элементов типа T и множество индексов I, обычно представляющих собой диапазон целых чисел;

б) *pointer* (T) – определяет указатель на объект типа T;

в) $T_1 \times T_2$ – декартово произведение типов T_1 и T_2 (считается левоассоциативным); обычно используются для представления списка типов, например параметров процедур и функций;

г) *record* $((f_1 \times T_1) \times (f_2 \times T_2))$ – определяет запись из двух полей: поля с именем f_1 типа T_1 и поля с именем f_2 типа T_2. Описание полей и их типов можно хранить в отдельной таблице символов. Тогда этот конструктор будет иметь более простую форму: *record* (t), где t –таблица символов с информацией о полях этого типа записи (по сути это указатель на таблицу);

д) $T_1 \times T_2 \to T$ – определяет функцию с входными параметрами типов T_1 и T_2 (область определения), возвращающую значение типа T (область значений). По аналогии с конструктором записи информацию о входных параметрах функции можно хранить в отдельной таблице символов. Тогда областью определения будет эта таблица символов: $t \to T$.

Набор подобных конструкторов типа формируется для всех возможных типов языка программирования. Множество правил назначения выражений типов различным конструкциям языка программирования называется *системой типов* данного языка.

Одним из способов представления выражений типов является использование графов, в частности деревьев или ориентированных ациклических графов – дагов (Directed Acyclic Graph), в которых внутренним вершинам соответствуют конструкторы типа, а листьям – фундаментальные типы. Например, тип двумерного массива, представляющего квадратную матрицу вещественных чисел размером 10×10, определяется выражением *array*(1..10, *array*(1..10, *real*)). Дерево (*а*) и даг (*б*) для этого типа представлены на рис. 7.

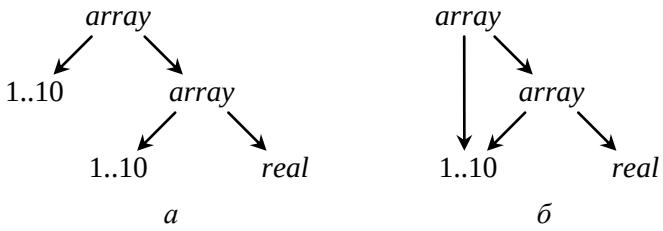


Рис. 7. Представления выражения типа *array* (1..10, *array* (1..10, *real*)): *а* – дерево; *б* – даг

Использование дагов может оказаться более предпочтительным, поскольку в этом случае возможна экономия памяти (вместо самого типа в этом случае хранится ссылка на него).

Ряд структур данных, таких, как связанные списки, могут определяться рекурсивно. Графы, представляющие такие типы, могут содержать циклы.

В некоторых случаях для выражений типов можно найти более компактную по сравнению с графом запись. Например, информацию о выражении типа можно закодировать последовательностью двоичных разрядов или в виде некоторой строки символов. При этом следует продумать систему кодирования так, чтобы различные типы не могли иметь одинаковый код. Такую запись выражений типов можно назвать линейным представлением типов, которые можно хранить в специальной таблице типов.

Определив представление типов, можно в таблице символов в качестве значения поля типа идентификатора использовать указатель на дерево или даг, либо ссылку на строку таблицы типов, хранящую линейное представление типов.

5.3. Эквивалентность и преобразование типов

При проверке типов возникает необходимость определения эквивалентности типов. Выделяют два метода определения эквивалентности типов:

1. *Структурная эквивалентность*. Два типа структурно эквивалентны, если они имеют одинаковую структуру, т.е. они либо представляют собой один и тот же фундаментальный тип, либо образованы путем применения одного и того же конструктора к структурно эквивалентным типам.

2. *Именная эквивалентность*. Два типа эквивалентны, если они объявлены с помощью одного и того же имени типа. Именная эквивалентность является более ограничивающей формой типизации, чем структурная, и обычно используется в строго типизированных языках.

Рассмотрим следующее объявление типов в синтаксисе языка Паскаль:

```
type
```

```
    TSpeed = real;
    TSize = real;
var
    Speed : TSpeed;
    Size : TSize;
```

Переменные *Speed* и *Size* структурно эквивалентны, поскольку представляют один и тот же фундаментальный тип *real*, но не эквивалентны по критериям именной эквивалентности.

Если выражения типа содержат переменные, то при определении эквивалентности выражений возникает задача подстановки выражений типа вместо этих переменных. Такая задача известна как *унификация* выражений. Алгоритм унификации выражений типа и примеры его применения для проверки структурной эквивалентности типов подробно рассмотрены в работах [1; 2].

Когда в тех или иных операциях или операторах присутствуют данные, относящиеся к различным типам, возникает вопрос о совместимости и преобразовании типов. Совместимость типов означает, что для объектов разных типов возможно приведение типа объекта к другому типу. Преобразование одного типа в другой называется *неявным*, если оно должно выполняться компилятором автоматически. Для этого в семантических действиях схемы трансляции должны быть предусмотрены соответствующие операции по преобразованию типов. Преобразование называется *явным*, если в программе специально указывается необходимость преобразования. Для этого язык должен иметь специальные средства (обычно они выглядят как применение функции или процедуры) для реализации подобных преобразований. Явное преобразование типов выполняется в процессе выполнения программы, а не в процессе компиляции, поэтому, если язык предусматривает только явные преобразования (характерно для строго типизированных языков), в схемах трансляции отсутствуют действия, связанные с преобразованием типов.

Если при преобразовании типов нет потери информации, такое преобразование называется *расширяющим*, например преобразование целого типа в вещественный тип. Преобразование типов называется *сужающим*, если оно может привести к потере информации, например при преобразовании вещественного типа в целый тип.

5.4. СУО для проверки типов

Процесс построения *L*-атрибутного СУО для проверки типов рассмотрим отдельными фрагментами: сначала объявления типов, затем проверку типов в выражениях и операторах языка.

Объявления типов различных объектов (переменных, функций и т.п.) обычно сгруппированы в декларативной части транслируемой программы. В ряде языков программирования могут быть и другие способы объявления типов (например, объявления типов рассредоточены в пределах исходной программы), вплоть до определения типа объекта по умолчанию (т.е. тип объекта явно не объявляется).

Очевидная грамматика для объявления типа для некоторого списка переменных – это грамматика со следующими продукциями:

$D \rightarrow L$ **:** T **;**

$L \rightarrow$ **id** | L **,** **id**.

Несмотря на то что это *LR*-грамматика, на ее основе нельзя получить *L*-атрибутное СУО, поскольку идентификаторы порождаются нетерминалом *L*, но тип как синтезируемый атрибут нетерминала *T* в поддереве *L* дерева разбора еще не известен. Эта проблема решается соответствующим преобразованием грамматики:

$D \rightarrow$ **id** L **;**

$L \rightarrow$ **,** **id** L | **:** T.

Теперь тип рассматривается как синтезируемый атрибут *type* нетерминалов *L* и *T*, который можно внести в таблицу символов каждого идентификатора, порождаемого *L*, как это показано в СУО в табл. 6.

Таблица 6

L-атрибутное СУО для объявления типа

Продукция	Семантические правила
$D \rightarrow$ **id** L **;**	*AddType* (**id**.*ns*, *L*.*type*)
$L \rightarrow$ **,** **id** L_1	*AddType* (**id**.*ns*, L_1.*type*); *L*.*type* := L_1.*type*
$L \rightarrow$ **:** T	*L*.*type* := *T*.*type*

В данном СУО процедура *AddType* (**id**.*ns*, *L*.*type*) сохраняет тип *L*.*type* для идентификатора **id** в записи таблицы символов, на

которую указывает атрибут терминала **id**.*ns* (*ns* – атрибут токена **id**, предоставляемый лексическим анализатором). СУО легко можно дополнить семантическими правилами для сохранения в таблице символов размера типа, диапазона значений, адреса выделенной памяти, числа измерений для массивов и тому подобного, дополнив символы грамматики соответствующими атрибутами.

Приведенный пример иллюстрирует то, что в процессе разработки СУО (даже если его основой является грамматика соответствующего класса) могут потребоваться дополнительные преобразования грамматики, чтобы СУО стало *L*-атрибутным.

СУО для проверки типов арифметических выражений будем рассматривать в предположении, что язык является строго типизированным, и нет никаких неявных преобразований типов, т.е. при выполнении арифметических операций совместимы только выражения, имеющие один и тот же тип. Тогда достаточно просто построить СУО, представленное в табл. 7.

Таблица 7

СУО для проверки типов арифметических выражений

Продукция	Семантические правила
$S \to E\perp$	
$E \to E_1 + T$	**if** E_1.*type* = *T.type* **then** *E.type* := E_1.*type* **else** *type_error*
$E \to T$	*E.type* := *T.type*
$T \to T_1 * F$	**if** T_1.*type* = *F.type* **then** *T.type* := T_1.*type* **else** *type_error*
$T \to T_1$ **mod** F	**if** T_1.*type* = *integer* **and** *F.type* = *integer* **then** *T.type* := *integer* **else** *type_error*
$T \to F$	*T.type* := *F.type*
$F \to (E)$	*F.type* := *E.type*
$F \to$ **num**	*F.type* := *GetType* (**num**.*ns*)
$F \to$ **id**	*F.type* := *GetType* (**id**.*ns*)

Если для каких-либо операций допустимы только выражения определенных типов, то семантические правила необходимо дополнить проверкой этих ограничений, как это сделано, например, в продукции для операции **mod** (вычисление остатка от целочисленного деления).

Проверка типов в логических (булевых) выражениях зависит от места их использования (синтаксического контекста).

Если логическое выражение используется в правой части оператора присваивания, то вычисляется его значение. В этом случае вычисление логического выражения можно реализовать подобно вычислению арифметического выражения и СУО для проверки типов будет похоже на приведенное выше. Можно также вместо вычисления значения использовать команды переходов, когда значение выражения устанавливается в зависимости от достигнутой программой позиции.

Если же логическое выражение используется в качестве условия в управляющих операторах (**if**, **while** и т.п.) для управления ходом выполнения программы, то вместо вычисления его значения используются команды условных и безусловных переходов, которые в зависимости от значения логического выражения передают управление в ту или иную позицию кода. Это все реализуется в процессе генерации промежуточного кода и, следовательно, не требует специальной проверки типов для логических выражений, как это делается для арифметических выражений. Для определения синтаксического контекста использования логических выражений можно использовать разные приемы [1]: применение разных нетерминалов для обозначения логических и арифметических выражений, добавление специальных наследуемых атрибутов или установка соответствующего флага в процессе синтаксического анализа.

Проверка типов для операторов языков программирования затрагивает в основном только те операторы, в которых есть ограничения на типы используемых выражений, причем эти ограничения не реализованы синтаксически. Это чаще всего управляющие операторы, где обычно используемые в качестве условий выражения должны иметь тип *Boolean*. Оператор присваивания также требует совместимости или совпадения типов левой и правой частей. В качестве примера для таких операторов можно представить следующее СУО (при условии, что нет синтаксических ограничений на тип выражения *E*) (табл. 8).

Таблица 8

СУО для проверки типов управляющих операторов

Продукция	Семантические правила
$S \rightarrow$ **id** := E	**if** *GetType* (**id**.*ns*) ≠ *E.type* **then** *type_error*
$S \rightarrow$ **if** E **then** S	**if** *E.type* ≠ *Boolean* **then** *type_error*
$E \rightarrow$ **while** E **do** S	**if** *E.type* ≠ *Boolean* **then** *type_error*

6. ГЕНЕРАЦИЯ ПРОМЕЖУТОЧНОГО КОДА

Фаза генерации промежуточного кода предназначена для преобразования исходной программы в промежуточное представление, которое можно рассматривать как программу для некоторой виртуальной машины. Промежуточный код должен относительно просто транслироваться в целевой код. Использование промежуточного представления имеет определенные преимущества [4]:

а) позволяет не учитывать особенности целевого языка, которые значительно усложняют трансляцию;

б) упрощается перенос на другую целевую машину, для чего потребуется только реализовать преобразование промежуточного кода в язык новой машины;

в) к промежуточному представлению можно применить машинно-независимую оптимизацию.

Наиболее распространенными формами представления промежуточной программы являются динамические структуры, представляющие ориентированный граф (в частности, синтаксическое дерево), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись, байт-код Java. Достаточно подробное изложение этих форм промежуточных представлений с иллюстрациями на примерах можно найти в работах [1; 2; 3]. Ограничимся рассмотрением трехадресного кода.

6.1. Трехадресный код

Трехадресный код является линеаризованным представлением синтаксического дерева и представляет собой последовательность команд вида $x := y\ op\ z$, где x, y и z – имена, константы (кроме x) или временные переменные, генерируемые компилятором; op – некоторая операция, например, арифметическая операция или операция для работы с логическими значениями. Команды могут иметь символьные метки, представляющие индекс трехадресной команды в массиве, содержащем промежуточный код. Замена меток индексами может быть выполнена в процессе отдельного прохода либо с использованием метода обратных поправок (см. подразд. 6.5).

Элементы x, y и z для удобства представляются именами или константами. При реализации это обычно указатели на соответствующие записи таблицы символов.

Выбор множества команд является важной задачей при создании промежуточного представления. Оно должно быть достаточно богатым, чтобы позволить реализовать все операции исходного языка. Небольшое множество команд легче реализуется, однако может привести к генерации длинных последовательностей команд промежуточного представления.

Пример множества основных трехадресных команд:

1) команда присваивания вида $x := y\ op\ z$, где *op* – бинарная арифметическая или логическая операция;

2) команда присваивания вида $x := op\ y$, где *op* – унарная арифметическая или логическая операция (включая и операции преобразования типов);

3) команда копирования вида $x := y$;

4) индексированные присваивания типа $x := y[i]$ и $x[i] := y$.

5) безусловный переход **goto** *L*, после этой команды будет выполнена команда с меткой *L*;

6) условный переход типа **if** *x relop y* **goto** *L*, где *relop* – операция отношения; если отношение *x relop y* истинно, следующей выполняется команда с меткой *L*, в противном случае выполняется следующая за условным переходом команда;

7) условные переходы вида **if** *x* **goto** *L* и **ifFalse** *x* **goto** *L*, в которых следующей выполняется команда с меткой *L*, если значение *x* соответственно истинно или ложно, в противном случае выполняется следующая за условным переходом команда.

Данное множество команд можно дополнить командами вызова подпрограмм и передачи им параметров, командами возврата из подпрограмм и передачи возвращаемых значений, командами работы с адресами и указателями и другими командами в зависимости от возможностей исходного языка. Примеры таких команд можно посмотреть в работах [1; 3].

Рассмотрим фрагмент программы на языке Паскаль (каждый элемент массива занимает 4 единицы памяти, диапазон индексов массива 0..100)

```
s:=0; i:=0;
while i < 100 do
   s:=s+a[i];
   i:=i+1
end
```

В результате трансляции этого фрагмента может сформироваться трехадресный код (последовательность формируемых команд полностью зависит от семантических правил, связанных с продукциями грамматики):

```
50: s:=0
51: i:=0
52: if i >= 100 goto 60
53: t1:=s
54: t2:=i*4
55: t3:=a[t2]
56: s:=t1+t3
57: t4:=i+1
58: i:=t4
59: goto 52
60:
```

В полученном фрагменте кода отсчет номеров позиций команд начинается с 50. Команда `t2:=i*4` требуется для обеспечения прямого доступа к элементу массива (каждый из элементов занимает 4 единицы памяти).

Основными структурами данных для представления трехадресных команд являются четверки (тетрады), тройки (триады) и косвенные тройки.

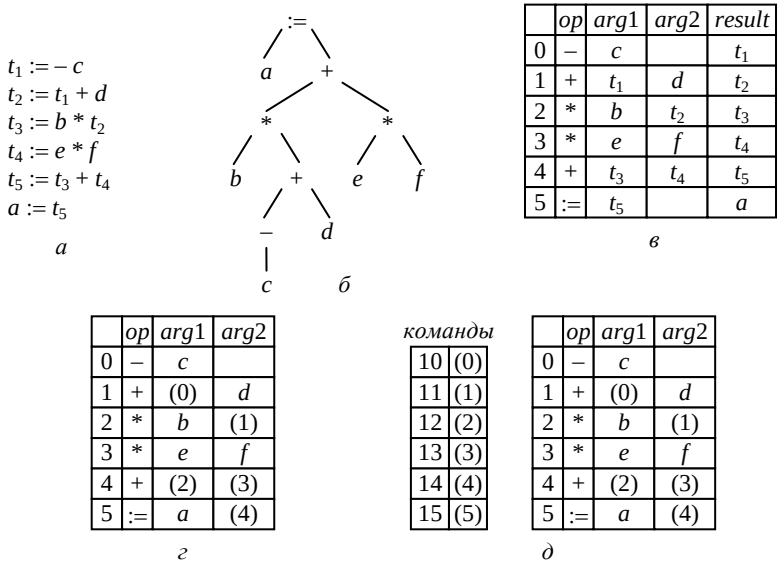
Четверка представляет собой запись с полями *op*, *arg*1, *arg*2 и *result* (рис. 8, *в*). Поле *op* содержит внутренний код операции. Поля *arg*1, *arg*2 и *result* обычно содержат указатели на соответствующие записи таблицы символов. Поэтому временные имена при их создании должны быть внесены в общую таблицу символов или в отдельную таблицу временных имен. Трехадресная команда $x := y + z$ представляется размещением + в *op*, y в *arg*1, z в *arg*2 и x в *result*. Унарные команды наподобие $x := -y$ или $x := y$ не используют *arg*2. Условные и безусловные переходы помещают в *result* целевую метку.

Тройка состоит из трех полей: *op*, *arg*1, *arg*2 (рис. 8, *г*). Они позволяют избежать вставки временных имен в таблицу символов, поскольку можно ссылаться на временное значение по номеру команды, которая вычисляет это значение. Поля *arg*1 и *arg*2 представляют собой либо указатели в таблицу символов

(для определенных программистом имен или констант), либо указатели на тройки (для временных значений). Тернарные операции наподобие $x[i] := y$ требуют двух троек, например, можно поместить x и i в одну тройку, а y – в другую. Аналогично $x := y[i]$ можно реализовать, рассматривая эту операцию так, как если бы это были две команды, $t := y[i]$ и $x := t$, где t – временная переменная, сгенерированная компилятором. Здесь временная переменная t в действительности в тройке не появляется, поскольку обращения к временным значениям выполняются с использованием их позиций в последовательности троек.

Косвенные тройки состоят не из последовательности самих троек, а из списка указателей на тройки (рис. 8, *д*). Например, можно использовать массив *команды* для перечисления указателей на тройки в требуемом порядке.

Пример трехадресного кода и его представления для присваивания $a := b * (- c + d) + e * f$ приведен на рис. 8.



	op	*arg*1	*arg*2	*result*
0	–	*c*		t_1
1	+	t_1	*d*	t_2
2	*	*b*	t_2	t_3
3	*	*e*	*f*	t_4
4	+	t_3	t_4	t_5
5	:=	t_5		*a*

в

	op	*arg*1	*arg*2
0	–	*c*	
1	+	(0)	*d*
2	*	*b*	(1)
3	*	*e*	*f*
4	+	(2)	(3)
5	:=	*a*	(4)

г

команды

10	(0)
11	(1)
12	(2)
13	(3)
14	(4)
15	(5)

	op	*arg*1	*arg*2
0	–	*c*	
1	+	(0)	*d*
2	*	*b*	(1)
3	*	*e*	*f*
4	+	(2)	(3)
5	:=	*a*	(4)

д

Рис. 8. Трехадресный код и его представления для оператора присваивания $a := b * (- c + d) + e * f$: *а* – трехадресный код; *б* – синтаксическое дерево; *в* – четверки; *г* – тройки; *д* – косвенные тройки

Рассмотренные представления трехадресных команд имеют свои достоинства и недостатки. При использовании четверок трехадресные команды, определяющие или использующие временные переменные, могут непосредственно обращаться к памяти для этих переменных с помощью таблицы символов. Другое преимущество четверок проявляется в оптимизирующем компиляторе, когда в процессе оптимизации приходится удалять или перемещать команды. При перемещении команды, вычисляющей *x*, команда, использующая это значение, не требует внесения каких-либо изменений. В случае же использования троек перемещение команды, определяющей временное значение, требует изменения всех ссылок на эту команду в полях *arg*1 и *arg*2. Эта проблема затрудняет использование троек в оптимизирующих компиляторах. При использовании косвенных троек такой проблемы не возникает – перемещение команд осуществляется простым переупорядочением списка команд, а сами тройки остаются неизменными.

Создание временных переменных при формировании трехадресного кода (например, при использовании четверок) обычно делается с помощью специальной функции, назовем ее для определенности *NewTemp*, которая создает новую временную переменную и возвращает указатель на соответствующую запись в таблице (в зависимости от реализации это может быть общая таблица символов или отдельная таблица временных переменных). Для оптимизирующего компилятора при каждом вызове *NewTemp* полезно создавать новую переменную, т.е. в трехадресном коде все присваивания выполняются для переменных с различными именами (так называемый принцип *статических единственных присваиваний* [1]). Для экономии памяти за счет уменьшения числа временных переменных можно использовать метод повторного их использования, учитывающий их время жизни [2], хотя он и имеет ряд ограничений и создает определенные трудности для оптимизации кода.

В следующих разделах, чтобы излишне не усложнять связанные с продукциями семантические правила, будем рассматривать только правила, связанные с генерацией промежуточного кода. При этом следует помнить, что наряду с правилами для трансляции в промежуточный код в СУО могут быть и правила для проверки типов (см. разд. 5).

6.2. Трансляция арифметических выражений

В процессе трансляции выражения в промежуточный код происходит его преобразование в трехадресный код, в котором каждая из команд имеет не более чем одну операцию (см. рис. 8, *а*).

СУО (табл. 9) формирует трехадресный код для оператора присваивания и арифметического выражения.

Таблица 9

СУО для трансляции оператора присваивания и арифметического выражения

Продукция	Семантические правила
$S \to \mathbf{id} := E$	*S.code* := *E.code* || *Gen* (**id**.*ns* ':=' *E.addr*)
$E \to E_1 + T$	*E.addr* := *NewTemp* () *E.code* := E_1.*code* || *T.code* || *Gen* (*E.addr* ':=' E_1.*addr* '+' *T.addr*)
$E \to T$	*E.addr* := *T.addr* *E.code* := *T.code*
$E \to - T$	*E.addr* := *NewTemp* () *E.code* := *T.code* || *Gen* (*E.addr* ':=' '–' *T.addr*)
$T \to T_1 * F$	*T.addr* := *NewTemp* () *T.code* := T_1.*code* || *F.code* || *Gen* (*T.addr* ':=' T_1.*addr* '*' *F.addr*)
$T \to F$	*T.addr* := *F.addr* *T.code* := *F.code*
$F \to (E)$	*F.addr* := *E.addr* *F.code* := *E.code*
$F \to \mathbf{id}$	*F.addr* := **id**.*ns* *F.code* := ''

Синтезируемый атрибут *addr* является указателем на запись в таблице символов, где имеется вся необходимая информация об объекте (переменная, константа или временная переменная), вплоть до адреса размещения объекта в памяти. В синтезируемом атрибуте *code* формируется трехадресный код для соответствующего нетерминала. Функция *NewTemp* () создает новую временную переменную и возвращает указатель на соответствующую запись в таблице (в зависимости от реализации это может быть

общая таблица символов или специальная таблица временных имен). Формирование трехадресной команды для удобства показывается процедурой *Gen* (*x* ′:=′ *y* ′+′ *z*), представляющей команду *x* := *y* + *z* (при реализации вместо символов операций в команду записываются их коды). В результате выполнения правила *F.addr* := **id**.*ns* для продукции *F* → **id** атрибут *F.addr* указывает на запись в таблице символов для данного экземпляра **id**. Символ || обозначает операцию конкатенации строк.

Поскольку атрибуты *code* могут быть довольно длинными строками, чаще применяют *инкрементную* трансляцию. Она заключается в том, что формируется единый поток генерируемых трехадресных команд в некотором глобальном массиве или файле. Процедура *Gen* при этом не только представляет трехадресную команду, но и инкрементно добавляет новую команду к последовательности сформированных к данному моменту команд. Реализация инкрементной трансляции СУО, генерирующей тот же код, что и СУО в табл. 9, представлена в табл. 10.

Таблица 10

Инкрементная трансляция арифметического выражения

Продукция	Семантические правила
$S \to$ **id** $:= E$	*Gen* (**id**.*ns* ′:=′ *E.addr*)
$E \to E_1 + T$	*E.addr* := *NewTemp* () *Gen* (*E.addr* ′:=′ E_1.*addr* ′+′ *T.addr*)
$E \to T$	*E.addr* := *T.addr*
$E \to - T$	*E.addr* := *NewTemp* () *Gen* (*E.addr* ′:=′ ′–′ *T.addr*)
$T \to T_1 * F$	*T.addr* := *NewTemp* () *Gen* (*T.addr* ′:=′ T_1.*addr* ′*′ *F.addr*)
$T \to F$	*T.addr* := *F.addr*
$F \to (E)$	*F.addr* := *E.addr*
$F \to$ **id**	*F.addr* := **id**.*ns*

Как видно в рассмотренном примере, преобразование СУО для применения инкрементной трансляции особых проблем не вызывает (хотя и могут быть определенные трудности). Поэтому в большинстве СУО будем использовать атрибут *code*, по-

скольку он дает более наглядное представление о последовательности формирования трехадресного кода.

Рассмотренные выше СУО легко можно приспособить для соответствующих представлений трехадресных команд. Для четверок семантические правила практически не требуют изменений (в процедуре *Gen* следует уточнить порядок заполнения полей). Для троек и косвенных троек вместо *NewTemp* в атрибуты *addr* следует записывать номер (метку) текущей команды, для тернарных операций предусмотреть формирование двух троек. Для косвенных троек, кроме формирования самих троек, дополнительно следует создавать список указателей на тройки.

6.3. Трансляция логических выражений

Логические выражения строятся с помощью логических операций (**or**, **and**, **not**), операций отношения типа $a > b$ и логических констант **true** и **false**. Важной их особенностью является возможность применения методов сокращенного вычисления. Если в выражении a **and** b (a и b логические выражения) установлено, что a = **false**, то без вычисления b можно сказать, что все выражение будет иметь значение **false**. Аналогично для a **or** b, если a = **true**, то и значение всего выражения будет **true**. Исключение составляют выражения с побочными действиями, например функция, изменяющая некоторую глобальную переменную. В этом случае может понадобиться полное вычисление выражения (это не лучший стиль программирования).

Трансляция логических выражений зависит от места их использования (синтаксического контекста).

Если логическое выражение используется в качестве условия в управляющих операторах, то обычно используются команды условных и безусловных переходов, которые в зависимости от логического условия передают управление в ту или иную позицию кода (само значение не вычисляется).

Если же логическое выражение используется в правой части оператора присваивания, то может формироваться трехадресный код для его вычисления, подобный коду для арифметиче-

ских выражений. Можно также вместо вычисления логического значения использовать команды условных и безусловных переходов, когда значение выражения устанавливается в зависимости от достигнутой программой позиции.

Для определения синтаксического контекста использования логических выражений для обозначения арифметических и логических выражений будем использовать нетеминалы *E* и *B* соответственно.

СУО в табл. 11 иллюстрирует формирование трехадресного кода для логического выражения (используется инкрементная трансляция и предполагается, что выражение вычисляется полностью) по аналогии с арифметическим выражением.

Таблица 11

СУО для формирования трехадресного кода
для логического выражения (инкрементная трансляция)

Продукция	Семантические правила
$B \to B_1$ **or** B_2	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ B_1.*addr* ′or′ B_2.*addr*)
$B \to B_1$ **and** B_2	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ B_1.*addr* ′and′ B_2.*addr*)
$B \to$ **not** B_1	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′not′ B_1.*addr*)
$B \to E_1$ **rel** E_2	*B.addr* := *NewTemp* () *Gen* (′if′ E_1.*addr* **rel**.*op* E_2.*addr* ′goto′ *next*+3) *Gen* (*B.addr* ′:=′ ′false′) *Gen* (′goto′ *next*+2) *Gen* (*B.addr* ′:=′ ′true′)
$B \to (B_1)$	*B.addr* := B_1.*addr*
$B \to$ **true**	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′true′)
$B \to$ **false**	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′false′)

В синтезируемом атрибуте *addr* устанавливается логическое значение (**true** или **false**) соответствующего выражения. Переменная *next* содержит индекс очередной трехадресной команды

в формируемом потоке. Процедура *Gen* после построения каждой команды инкрементирует значение *next*.

Для выражения $b < c$ **and not** ($d > e$ **or** $f < g$) СУО сгенерирует следующий код (условно начиная с позиции 50):

```
50: if b < c goto 53
51: t1:=false
52: goto 54
53: t1:=true
54: if d > e goto 57
55: t2:=false
56: goto 58
57: t2:=true
58: if f < g goto 61
59: t3:=false
60: goto 62
61: t3:=true
62: t4:=t2 or t3
63: t5:=not t4
64: t6:=t1 and t5.
```

Другой, более распространенный подход к вычислению логических выражений основан на использовании команд условных и безусловных переходов, которые в зависимости от логического условия передают управление в ту или иную позицию кода (само значение не вычисляется, в коде отсутствуют логические операции). Такой подход позволяет достаточно легко реализовать сокращенное вычисление выражений.

СУО для трансляции логических выражений с помощью команд условного и безусловного переходов показано в табл. 12.

В синтезируемом атрибуте *code* формируется трехадресный код для соответствующего нетерминала. Функция *NewLabel*() создает новую метку, а функция *Label*() назначает метку очередной трехадресной команде. Логическое выражение *B* имеет наследуемые атрибуты *B.true* и *B.false*. Значениями этих атрибутов являются метки, которым передается управление в случае истинности или ложности выражения *B* соответственно. Оператор присваивания *S* имеет наследуемый атрибут *S.next*, значением которого является метка, указывающая на команду, непосредственно следующую за кодом *S* (подробнее об этом атрибуте в подразд. 6.4).

Таблица 12

СУО для трансляции логических выражений

Продукция	Семантические правила
$S \to \mathbf{id} := B$	$B.true := NewLabel()$ $B.false := NewLabel()$ $S.code := B.code \parallel Label(B.true) \parallel Gen(\mathbf{id}.ns\ ':='\ 'true')$ $\parallel Gen('goto'\ S.next)$ $\parallel Label(B.false) \parallel Gen(\mathbf{id}.ns\ ':='\ 'false')$
$B \to B_1\ \mathbf{or}\ B_2$	$B_1.true := B.true$ $B_1.false := NewLabel()$ $B_2.true := B.true$ $B_2.false := B.false$ $B.code := B_1.code \parallel Label(B_1.false) \parallel B_2.code$
$B \to B_1\ \mathbf{and}\ B_2$	$B_1.true := NewLabel()$ $B_1.false := B.false$ $B_2.true := B.true$ $B_2.false := B.false$ $B.code := B_1.code \parallel Label(B_1.true) \parallel B_2.code$
$B \to \mathbf{not}\ B_1$	$B_1.true := B.false$ $B_1.false := B.true$ $B.code := B_1.code$
$B \to E_1\ \mathbf{rel}\ E_2$	$B.code := E_1.code \parallel E_2.code$ $\parallel Gen('if'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ 'goto'\ B.true)$ $\parallel Gen('goto'\ B.false)$
$B \to (B_1)$	$B_1.true := B.true$ $B_1.false := B.false$ $B.code := B_1.code$
$B \to \mathbf{true}$	$B.code := Gen('goto'\ B.true)$
$B \to \mathbf{false}$	$B.code := Gen('goto'\ B.false)$

Первая продукция специально добавлена в СУО для иллюстрации использования логического выражения в правой части оператора присваивания. Значение выражения определяется позицией в последовательности команд. Метки *B.true* и *B.false* означают позиции перехода для выражения в случае его истинности или ложности соответственно. Тогда логическое выражение истинно, если управление достигает команды с меткой *B.true*, и ложно, если достигает команды с меткой *B.false*.

Рассмотрим продукцию $B \rightarrow B_1$ **or** B_2. Если B_1 истинно, то и все выражение B истинно, поэтому $B_1.true := B.true$. Если B_1 ложно, то следует перейти к вычислению B_2, поэтому $B_1.false$ должна быть меткой первой команды кода для B_2. Метки $B_2.true$ и $B_2.false$ совпадают с соответствующими метками для B.

В продукции $B \rightarrow B_1$ **and** B_2 аналогичные правила. Отличие в том, что, если B_1 истинно, реализуется переход к вычислению B_2, в противном случае B ложно.

В продукции $B \rightarrow$ **not** B_1 не формируется никакой новый код, а просто реализуется перенаправление управления с истины на ложь и наоборот.

Семантические правила для остальных продукций очевидны и не требуют дополнительных пояснений.

Для присваивания a := $b < c$ **and not** ($d > e$ **or** $f < g$) данное СУО сгенерирует следующий код (*Snext* – метка команды, непосредственно следующей за данным оператором присваивания):

```
      if b < c goto L3
      goto L2
L3:  if d > e goto L2
      goto L4
L4:  if f < g goto L2
      goto L1
L1:  a:=true
      goto Snext
L2:  a:=false.
```

Обычно генерируемый код не оптимален. Например, можно без всяких последствий удалить четвертую и шестую команды (безусловные переходы **goto** *L*4 и **goto** *L*1), поскольку они реализуют переход на непосредственно следующие за ними команды. Можно убрать также вторую команду (**goto** *L*2), если в первой команде заменить **if** на **ifFalse**, т.е. поменяв условие на обратное. Тогда улучшенный трехадресный код будет иметь следующий вид:

```
      ifFalse b < c goto L2
      if d > e goto L2
      if f < g goto L2
L1:  a:=true
      goto Snext
L2:  a:=false.
```

Такое удаление избыточных команд перехода можно выполнить в процессе оптимизации кода.

Возможен и другой подход, связанный с соответствующим изменением семантических правил так, чтобы избыточные переходы не включались в генерируемый код. Один из таких подходов основан на использовании специальной метки, означающей отсутствие перехода. Рассмотрим детали этого метода [1].

Обозначим через *fall* специальную метку, означающую отсутствие перехода. Если логическое выражение *B* используется в управляющих операторах, то правила для соответствующих этим операторам продукций (например, СУО из подразд. 6.4) можно модифицировать так, чтобы установить *B.true* равным *fall*. Это позволит управлению проходить сквозь код *B*.

Рассмотрим правила для продукции $B \to E_1$ **rel** E_2 (табл. 12). Для их модификации необходимо рассмотреть следующие ситуации. Если *B.true* и *B.false* являются явными метками (ни одна из них не равна *fall*), то должны генерироваться две команды переходов, как это было ранее. Если *B.true* представляет собой явную метку, а *B.false* – метку *fall*, то должна генерироваться одна команда **if**. Если же *B.true* = *fall*, а *B.false* ≠ *fall*, то должна генерироваться команда **ifFalse**. В случае, когда и *B.true*, и *B.false* равны *fall*, никакие переходы не генерируются. Тогда новые семантические правила для продукции будут следующими:

test := E_1.*addr* **rel**.*op* E_2.*addr*

str := **if** *B.true* ≠ *fall* **and** *B.false* ≠ *fall* **then**
Gen (′if′ *test* ′goto′ *B.true*) || *Gen* (′goto′ *B.false*)
else if *B.true* ≠ *fall* **then** *Gen* (′if′ *test* ′goto′ *B.true*)
else if *B.false* ≠ *fall* **then** *Gen* (′ifFalse′ *test* ′goto′ *B.false*)
else ′ ′

B.code := E_1.*code* || E_2.*code* || *str*.

Новые правила для продукции $B \to B_1$ **or** B_2:

if *B.true* ≠ *fall* **then** B_1.*true* := *B.true* **else** B_1.*true* := *NewLabel* ()

B_1.*false* := *fall*

B_2.*true* := *B.true*

B_2.*false* := *B.false*

if *B.true* ≠ *fall* **then** *B.code* := B_1.*code* || B_2.*code*

else *B.code* := B_1.*code* || B_2.*code* || *Label* (B_1.*true*).

Следует заметить, что смысл метки *fall* для *B* отличается от смысла для B_1. Пусть *B.true = fall*. Это значит, что управление проходит через *B*, если значение *B* истинно. Хотя значение *B* истинно, если это же значение принимает B_1, метка B_1*.true* должна обеспечивать переход управления через код B_2 к команде, непосредственно следующей за кодом *B*. Если же вычисленное значение B_1 ложно, то истинность значения *B* определяется значением B_2. Поэтому приведенные правила гарантируют, что метка B_1*.false* обеспечивает прохождение управления от B_1 к коду B_2.

Аналогичным образом можно модифицировать семантические правила для других продукций СУО в табл. 12.

6.4. Трансляция управляющих операторов

СУО для трансляции в трехадресный код основных управляющих операторов представлено в табл. 13. Для полноты и наглядности установки начальных значений наследуемых атрибутов в СУО добавлены и другие продукции.

Таблица 13

СУО для трансляции управляющих операторов

Продукция	Семантические правила
$P \to S$	*S.next* := *NewLabel* () *P.code* := *S.code* || *Label* (*S.next*)
$S \to$ **id** := E	см. табл. 9
$S \to$ **id** := B	см. табл. 12
$S \to$ **if** B **then** S_1	*B.true* := *NewLabel* (); *B.false* := *S.next* S_1*.next* := *S.next* *S.code* := *B.code* || *Label* (*B.true*) || S_1*.code*
$S \to$**if** B **then** S_1 **else** S_2	*B.true* := *NewLabel* (); *B.false* := *NewLabel* () S_1*.next* := *S.next*; S_2*.next* := *S.next* *S.code* := *B.code* || *Label* (*B.true*) || S_1*.code* || *Gen* ('goto' *S.next*) || *Label* (*B.false*) || S_2*.code*
$S \to$ **while** B **do** S_1	*beg* := *NewLabel* () *B.true* := *NewLabel* (); *B.false* := *S.next* S_1*.next* := *beg* *S.code* := *Label* (*beg*) || *B.code* || *Label* (*B.true*) || S_1*.code* || *Gen* ('goto' *beg*)
$S \to S_1$; S_2	S_1*.next* := *NewLabel* (); S_2*.next* := *S.next* *S.code* := S_1*.code* || *Label* (S_1*.next*) || S_2*.code*

Логическое выражение *B* имеет наследуемые атрибуты *B.true* и *B.false* (см. СУО в табл. 12). С этими атрибутами связаны метки, которым передается управление в случае истинности или ложности выражения *B* соответственно. Оператор *S* имеет наследуемый атрибут *S.next*, с которым связана метка, указывающая на команду, непосредственно следующую за кодом *S*.

Структура генерируемых этим СУО кодов показана на рис. 9. Для операторов **if-then** и **while** значения меток *B.false* и *S.next* совпадают.

В правилах для продукции $S \rightarrow$ **if** *B* **then** S_1 создается новая метка *B.true*, которая назначается первой трехадресной команде, генерируемой для оператора S_1 (рис. 9, *а*). Установкой атрибуту *B.false* значения *S.next* обеспечивается пропуск кода для оператора S_1 в случае ложности значения *B*. Очевидно, что непосредственно следующая за *S* команда с меткой *S.next* является также непосредственно следующей за S_1. Поэтому наследуемому атрибуту S_1*.next* устанавливается значение (метка) *S.next*.

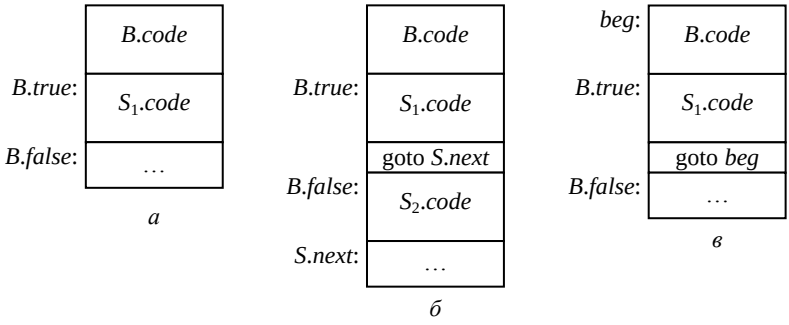


Рис. 9. Структура трехадресных кодов для управляющих операторов: *а* – **if-then** (*B.false* = *S.next*); *б* – **if-then-else**; *в* – **while** (*B.false* = *S.next*)

В правилах для оператора **if-then-else** (рис. 9, *б*) создаются новые метки *B.true* и *B.false*, которые назначаются первым командам кодов S_1 и S_2 соответственно для передачи им управления в зависимости от истинности или ложности *B*. В качестве последней команды кода S_1 формируется команда безусловного перехода **goto** *S.next* для пропуска кода S_2. После кода S_2 непосредственно следует команда с меткой *S.next*.

В операторе цикла **while** первая команда совпадает с первой командой кода для выражения *B* (рис. 9, *в*). В связанных с продукцией семантических правилах для передачи управления этой команде создается новая метка с использованием локальной переменной *beg*. Новая метка *B.true* создается для назначения первой команде кода, формируемого для оператора S_1, для передачи ей управления в случае истинности *B*. В конце кода для S_1 помещается команда **goto** *beg* для передачи управления в начало кода выражения *B*. Метка *B.false* устанавливается равной *S.next* для реализации перехода в случае ложности *B* к первой команде кода, следующего после оператора *S*. Метка S_1*.next* устанавливается равной *beg*, чтобы при наличии в коде S_1 команд переходов управление передавалось в начало цикла к команде с меткой *beg*.

Семантические правила для остальных продукций очевидны и не требуют дополнительных пояснений.

В общем случае реализация семантических правил может привести к тому, что одной трехадресной команде может быть назначено нескольких меток. Удаление лишних меток можно реализовать в процессе оптимизации кода. Другой подход основан на применении метода обратных поправок (подразд. 6.5), в котором метки создаются только тогда, когда они необходимы.

Если не определен синтаксический контекст особенностей использования логических и арифметических выражений, можно использовать для выражений атрибут, который позволит отличать типы выражений друг от друга и выбирать соответствующие способы генерации промежуточного кода. Подробнее об этом можно посмотреть в работе [2].

6.5. Метод обратных поправок

При инкрементной трансляции (когда формируется единый поток генерируемых трехадресных команд в некотором глобальном массиве или файле) логических выражений и управляющих операторов возникает проблема, которая заключается в том, что к моменту создания команды перехода позиция команды, к которой должно перейти управление, еще неизвестна. Например, в продукции $S \rightarrow$ **if** B **then** S_1 при трансляции *B* для случая его ложного значения формируется переход к первой ко-

манде кода для оператора, следующего за *S*. Однако к этому моменту, возможно, еще не завершена трансляция всего выражения *B* и еще не выполнена трансляция S_1. Поэтому целевая метка для перехода еще не известна.

Простейшим способом решения проблемы является использование двух проходов. Сначала выполняется трансляция с использованием символьных меток (см. СУО в табл. 12 и 13), а затем производится замена этих меток целевыми метками (индексами глобального массива трехадресного кода).

Другой подход, ориентированный на однопроходную трансляцию, формирует команду перехода с временно неопределенными переходами, которые доопределяются целевыми метками в момент, когда становятся известными позиции команд, к которым передается управление. Такое последовательное заполнение команд метками называется *методом обратных поправок* [1; 2].

В этом методе после генерации команды с временно неопределенным переходом эта команда (а точнее, ее индекс в массиве команд) помещается в специальный список команд, метки которых будут указаны после того, как они будут определены. Все переходы группируются в списки так, что команды из одного списка будут иметь одну и ту же целевую метку.

Пусть для определенности трехадресный код генерируется в виде массива, тогда метки представляют собой индексы этого массива.

Для работы со списками переходов используются следующие функции и процедура:

функция *MakeList*(*i*) создает новый одноэлементный список, состоящий только из *i* (индекс в массиве команд), возвращает указатель на созданный список;

функция *Merge*(p_1, p_2) объединяет списки, на которые указывают p_1 и p_2, возвращает указатель на объединенный список;

процедура *BackPatch*(*p, i*) устанавливает *i* в качестве целевой метки в каждую команду из списка, на который указывает *p*, после этого время жизни списка завершается.

Вместо наследуемых атрибутов *B.true* и *B.false*, как это было в СУО в табл. 12 и 13, используются синтезируемые атрибуты *B.truelist* и *B.falselist*. Эти атрибуты представляют собой ука-

затели на списки команд перехода, которые должны получить метку команды, которой передается управление при истинности или ложности выражения *B* соответственно. Аналогично вместо наследуемого атрибута *S.next* в СУО в табл. 13 используется синтезируемый атрибут *S.nextlist*, представляющий собой указатель на список команд переходов к команде, идущей непосредственно за кодом *S*.

СУО для инкрементной трансляции логических выражений методом обратных поправок показано в табл. 14.

Таблица 14

СУО для трансляции логических выражений методом обратных поправок

Продукция	Семантические правила
1) $S \to$ **id** $:= B$	*BackPatch* (*B.truelist*, *nextinstr*) *BackPatch* (*B.falselist*, *nextinstr* + 2) *Gen* (**id**.*ns* ′:=′ ′true′) *Gen* (′goto′ *nextinstr* + 2) *Gen* (**id**.*ns* ′:=′ ′false′) *S.nextlist* := **null**
2) $B \to B_1$ **or** $M\ B_2$	*BackPatch* (B_1.*falselist*, *M.instr*) *B.truelist* := *Merge* (B_1.*truelist*, B_2.*truelist*) *B.falselist* := B_2.*falselist*
3) $B \to B_1$ **and** $M\ B_2$	*BackPatch* (B_1.*truelist*, *M.instr*) *B.truelist* := B_2.*truelist* *B.falselist* := *Merge* (B_1.*falselist*, B_2.*falselist*)
4) $B \to$ **not** B_1	*B.truelist* := B_1.*falselist* *B.falselist* := B_1.*truelist*
5) $B \to E_1$ **rel** E_2	*B.truelist* := *MakeList* (*nextinstr*) *B.falselist* := *MakeList* (*nextinstr* + 1) *Gen* (′if′ E_1.*addr* **rel**.*op* E_2.*addr* ′goto ?′) *Gen* (′goto ?′)
6) $B \to (B_1)$	*B.truelist* := B_1.*truelist* *B.falselist* := B_1.*falselist*
7) $B \to$ **true**	*B.truelist* := *MakeList* (*nextinstr*) *Gen* (′goto ?′)
8) $B \to$ **false**	*B.falselist* := *MakeList* (*nextinstr*) *Gen* (′goto ?′)
9) $M \to \varepsilon$	*M.instr* := *nextinstr*

В продукции для операций **or** и **and** добавлен специальный нетерминал-маркер *M*, с которым связана продукция $M \to \varepsilon$. Этот маркер фиксирует момент, когда необходимо получить индекс очередной команды непосредственно перед ее генерацией. При преобразовании СУО в СУТ это действие должно выполняться непосредственно перед нетерминалом, для которого будет генерироваться код. В соответствии с правилами из подразд. 3.2 для восходящей трансляции все действия должны быть в конце правой части продукции, что и достигается добавлением нетерминалов-маркеров.

Глобальная переменная *nextinstr* выполняет функции счетчика трехадресных команд и хранит индекс очередной команды. Константа **null** служит для инициализации пустых списков.

Как и в СУО в табл. 12, первая продукция добавлена для иллюстрации использования логического выражения в правой части оператора присваивания. К моменту выполнения связанных с продукцией семантических правил код для *B* уже сформирован. Логическое выражение истинно, если управление достигает команды из списка *B.truelist*, и ложно, если достигает команды из списка *B.falselist*. Поэтому для команд перехода из этих списков устанавливаются целевые метки генерируемых команд присваивания идентификатору соответствующих значений логического выражения *B*. Поскольку из оператора присваивания нет никаких переходов, список *S.nextlist* делается пустым.

В продукции $B \to B_1$ **or** $M\ B_2$, если B_1 истинно, то и все выражение *B* истинно, поэтому список B_1*.truelist* объединяется со списком *B.truelist*. Если B_1 ложно, то следует перейти к вычислению B_2, поэтому целевой меткой переходов из B_1*.falselist* устанавливается метка первой команды кода для B_2. Эта метка получается с помощью синтезируемого атрибута *M.instr* маркера *M*. Списком *B.falselist* становится список B_2*.falselist*.

В продукции $B \to B_1$ **and** $M\ B_2$ аналогичные правила. Отличие в том, что если B_1 истинно, реализуется переход к вычислению B_2, в противном случае *B* ложно.

В продукции $B \to$ **not** B_1 для реализации перенаправления управления меняются местами списки для ложного и истинного значений выражения.

Для простоты в продукции $B \to E_1$ **rel** E_2 генерируются две команды переходов (условный и безусловный) без применения методов сокращения избыточных переходов (см. подразд. 6.3).

Семантические правила для остальных продукций очевидны и не требуют дополнительных пояснений.

Преобразование данного СУО в СУТ приведет к тому, что все действия будут расположены в конце правых частей продукций и легко могут быть выполнены при свертке в процессе восходящего разбора.

Рассмотрим присваивание a := *b* < *c* **and not** (*d* > *e* **or** *f* < *g*). Отсчет номеров позиций команд начинается с 50. На аннотированном дереве разбора (рис. 10) атрибуты для удобства обозначены: *truelist – t, falselist – f, instr – i,* значения атрибутов *truelist* и *falselist* показываются как содержимое списков.

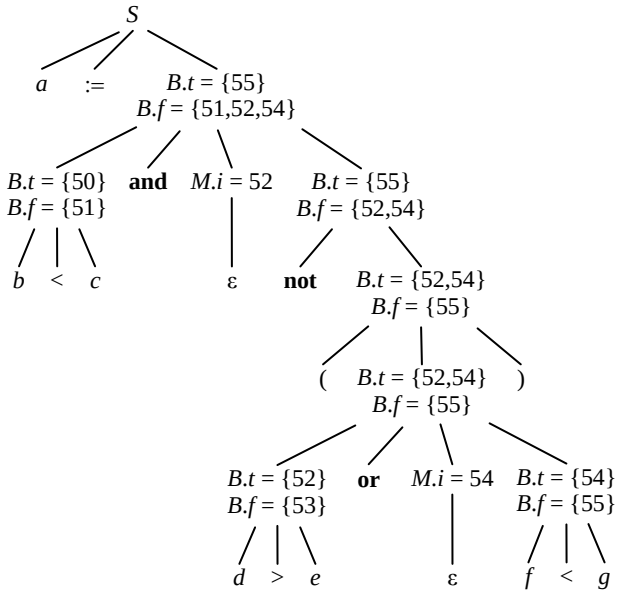


Рис. 10. Аннотированное дерево разбора
для a := *b* < *c* **and not** (*d* > *e* **or** *f* < *g*)

В соответствии с продукцией 5 формируются две команды:

```
50: if b < c goto ?
51: goto ?
```

Сгенерирован код для B_1 продукции $B \rightarrow B_1$ **and** M B_2. С помощью маркера M в атрибуте *M.instr* сохраняется текущее значение *nextinstr*, равное 52. Далее в соответствии с продукцией 5 генерируются команды:

```
52: if d > e goto ?
53: goto ?
```

Сгенерирован код для B_1 продукции $B \rightarrow B_1$ **or** M B_2. С помощью маркера M в атрибуте *M.instr* сохраняется текущее значение *nextinstr*, равное 54. Далее в соответствии с продукцией 5 генерируются команды:

```
54: if f < g goto ?
55: goto ?
```

Сгенерирован код для B_2 продукции $B \rightarrow B_1$ **or** M B_2. Поскольку в данный момент B_1.*falselist* = {53} и *M.instr* = 54, выполняется *BackPatch*({53}, 54), в результате чего команда 53 получит целевую метку 54. Список *B.truelist* = {52, 54} образуется в результате объединения *Merge*(*B*$_1$.*truelist*, *B*$_2$.*truelist*). Списком *B.falselist* становится список B_2.*falselist* = {55}.

Продукция $B \rightarrow (B_1)$ не изменяет атрибуты. В соответствии с продукцией $B \rightarrow$ **not** B_1 списки B_1.*truelist* и B_1.*falselist* меняются местами, т.е. *B.truelist* = {55} и *B.falselist* = {52, 54}.

К данному моменту завершено формирование кода для B_2 продукции $B \rightarrow B_1$ **and** M B_2. Атрибуты имеют следующие значения: B_1.*truelist* = {50}, B_1.*falselist* = {51}, B_2.*truelist* = {55}, B_2.*falselist* = {52, 54}, *M.instr* = 52. В результате выполнения *BackPatch*({50}, 52) команда 50 получит целевую метку 52. Списком *B.truelist* становится список B_2.*truelist* = {55}. Список *B.falselist* = {51, 52, 54} образуется в результате объединения *Merge*(*B*$_1$.*falselist*, *B*$_2$.*falselist*).

Завершена генерация кода для B продукции **id** := B. На данный момент *nextinstr* = 56. Поскольку *B.truelist* = {55}, в результате выполнения *BackPatch*({55}, 56) команда 55 получит целевую метку 56. Так как *B.falselist* = {51, 52, 54}, в результате выполнения *BackPatch*({51, 52, 54}, 58) команды 51, 52 и 54 получат целевую метку 58. Затем формируются команды для установки значений идентификатору **id**. В результате команды примут окончательный вид:

```
50: if b < c goto 52
51: goto 58
```

```
52: if d > e goto 58
53: goto 54
54: if f < g goto 58
55: goto 56
56: a:=true
57: goto 59
58: a:=false
59:
```

Метод обратных поправок можно использовать и для однопроходной инкрементной трансляции управляющих операторов. Соответствующее СУО представлено в табл. 15. Как и в СУО в табл. 14, логические выражения *B* имеют два списка переходов *B.truelist* и *B.falselist*, соответствующие истинным и ложным значениям *B*. Нетерминалы *L* (список операторов) и *S* (оператор) имеют атрибут *nextlist*, представляющий собой указатель на список команд переходов к команде, идущей непосредственно за кодом *L* или *S*. В эти списки группируются команды с временно неопределенными переходами.

Таблица 15

СУО для трансляции управляющих операторов методом обратных поправок

Продукция	Семантические правила
$L \to L_1$; $M\ S$	*BackPatch* (*L*$_1$.*nextlist*, *M.instr*) *L.nextlist* := *S.nextlist*
$L \to S$	*L.nextlist* := *S.nextlist*
$S \to$ **id** := B	см. табл. 14
$S \to$ **if** B **then** $M\ S_1$	*BackPatch* (*B.truelist*, *M.instr*) *S.nextlist* := *Merge* (*B.falselist*, *S*$_1$.*nextlist*)
$S \to$**if** B **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$	*BackPatch* (*B.truelist*, *M*$_1$.*instr*) *BackPatch* (*B.falselist*, *M*$_2$.*instr*) *tmp* := *Merge* (*S*$_1$.*nextlist*, *N.nextlist*) *S.nextlist* := *Merge* (*tmp*, *S*$_2$.*nextlist*)
$S \to$ **while** $M_1\ B$ **do** $M_2\ S_1$	*BackPatch* (*S*$_1$.*nextlist*, *M*$_1$.*instr*) *BackPatch* (*B.truelist*, *M*$_2$.*instr*) *S.nextlist* := *B.falselist* *Gen* (′goto′ *M*$_1$.*instr*)
$M \to \varepsilon$	*M.instr* := *nextinstr*
$N \to \varepsilon$	*N.nextlist* := *MakeList* (*nextinstr*) *Gen* (′goto ?′)

В продукции для оператора **if-then**, если выражение *B* истинно, то следует перейти к вычислению S_1, поэтому целевой меткой переходов из *B.truelist* устанавливается метка первой команды S_1. Эта метка получается с помощью синтезируемого атрибута *M.instr* маркера *M*. Если *B* ложно, необходимо обеспечить пропуск оператора S_1 и переход к первой команде, непосредственно следующей за *S* (эта команда является также непосредственно следующей за S_1). Поэтому выполняется объединение списков *B.falselist* и S_1*.nextlist* в список *S.nextlist*.

При трансляции оператора **if-then-else** выполняется обратная поправка для переходов из списка *B.truelist* установкой целевой метки M_1*.instr*, представляющей начало кода S_1 и из списка *B.falselist* установкой целевой метки M_2*.instr*, представляющей начало кода S_2. В конец кода S_1 необходимо добавить безусловный переход для пропуска кода S_2 и передачи управления команде, непосредственно следующей за *S*. Позиция этой команды в данный момент неизвестна. Поэтому с помощью маркера *N* с продукцией $N \rightarrow \varepsilon$ формируется команда безусловного перехода и создается одноэлементный список *N.nextlist* с индексом этой команды. Затем списки S_1*.nextlist, N.nextlist* и S_2*.nextlist* объединяются в список *S.nextlist* (для объединения используется локальная переменная *tmp*).

В продукции для оператора цикла **while** используются маркеры M_1 для первой команды кода *B* и M_2 для первой команды кода S_1. Необходимость маркера M_1 объясняется тем, что в конец кода S_1 добавляется безусловный переход для передачи управления на первую команду кода *B*, индекс которой должен быть известен. При трансляции выполняется обратная поправка для переходов из списка S_1*.nextlist* установкой целевой метки M_1*.instr*, представляющей начало кода *B* и из списка *B.truelist* установкой целевой метки M_2*.instr*, представляющей начало кода S_1. Если *B* ложно, необходимо обеспечить пропуск оператора S_1 и переход к первой команде, непосредственно следующей за *S*. Поэтому списком *S.nextlist* становится список *B.falselist*. Добавляется команда безусловного перехода для передачи управления первой команде кода *B*, индекс которой сохранен в M_1*.instr*.

Последовательность операторов представляется продукцией $L \to L_1$; M S. Выполняется обратная поправка для переходов из списка L_1.*nextlist* установкой метки M.*instr*, представляющей начало кода S. Списком L.*nextlist* становится список S.*nextlist*.

Для продукции $L \to S$ список L.*nextlist* совпадает S.*nextlist*.

В рассмотренном СУО трехадресная команда генерируется только в продукциях для операторов **if-then-else** (через маркер N и продукцию $N \to \varepsilon$) и **while**. Весь остальной код формируется в семантических правилах, связанных с операторами присваивания и выражениями.

В рассмотренных выше СУО, использующих метод обратных поправок, все семантические правила выполняются в конце правых частей продукций. Это означает, что это практически готовые схемы трансляции для восходящего разбора. Достаточно решить вопросы хранения атрибутов и детализировать действия соответствующими операциями со стеком.

В заключение следует отметить, что в лекциях основное внимание было уделено методам построения СУО для решения задач проверки типов и генерации промежуточного кода. Поэтому, чтобы упростить изложение этих методов, не рассматривались структурированные типы данных (массивы, записи и т.п.) и применение конструкторов типа. Варианты СУО и СУТ, обеспечивающих проверку типов, распределение памяти и генерацию промежуточного кода для различных структурированных типов данных, указателей, операторов языков программирования для самостоятельной проработки можно найти в работах [1; 2]. Рекомендации по практическому применению методов для реализации различных фаз компиляции можно посмотреть также в работе [9].

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Ахо А.* Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Д. Ульман. – 2-е изд.– М.: Вильямс, 2015. – 1184 с.

2. *Ахо А.* Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман. – М.: Вильямс, 2003. – 768 с.

3. *Гавриков М.М.* Теоретические основы разработки и реализации языков программирования: учеб. пособие / М.М. Гавриков, А.Н. Иванченко, Д.В. Гринченков. – М.: КНОРУС, 2010. – 184 с.

4. *Опалева Э.А.* Языки программирования и методы трансляции / Э.А. Опалева, В.П. Самойленко. – СПб.: БХВ-Петербург, 2005. – 480 с.

5. *Павлов Л.А.* Восходящий синтаксический анализ: конспект лекций / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2004. – 44 с.

6. *Павлов Л.А.* Нисходящий синтаксический анализ: конспект лекций / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2003. – 48 с.

7. *Павлов Л.А.* Структуры и алгоритмы обработки данных: учеб. пособие / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2008. – 252 с.

8. *Свердлов С.З.* Языки программирования и методы трансляции: учеб. пособие / С.З. Свердлов. – СПб.: Питер, 2007. – 638 с.

9. Теория языков программирования и методы трансляции: метод. указания к расчетно-графическим работам / сост. Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2015. – 48 с.

ОГЛАВЛЕНИЕ

Учебное издание

Павлов Леонид Александрович

СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ТРАНСЛЯЦИЯ

Учебное пособие

Редактор *Л.Г. Григорьева*
Компьютерная верстка и правка *Л.А. Павлова, Е.В. Ивановой*

Согласно Закону № 436-ФЗ от 29 декабря 2010 года
данная продукция не подлежит маркировке

Подписано в печать 5.05.2017. Формат 60×84/16.
Бумага газетная. Печать офсетная. Гарнитура Times New Roman.
Усл. печ. л. 3,48. Уч.-изд. л. 3,5. Тираж 100 экз. Заказ № 503.

Издательство Чувашского университета
Типография университета
428015 Чебоксары, Московский просп., 15