Министерство образования и науки Российской Федерации
Алтайский государственный технический
университет им. И.И.Ползунова

Е.Н. КРЮЧКОВА

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Барнаул 2014

УДК 681.142.2:517

Крючкова Е.Н. Теория языков программирования и методы трансляции. Учебное пособие / Алт. госуд. технич. ун-т им. И.И.Ползунова. Барнаул, 2014. — 111с.

Учебное пособие предназначено для студентов вуза, специализирующихся в области программирования, в частности, для студентов направления 231000 — "Программная инженерия".

Рекомендовано на заседании кафедры прикладной математики
протокол N 1 от 15 января 2013 г.

Рецензент: Зав.кафедрой информатики С.И. Жилин (АлтГУ)

ВВЕДЕНИЕ

Обучение компиляторам студентов, специализирующихся в области информационных технологий, должно носить фундаментальный характер. Данная книга представляет собой учебное пособие по курсу теории языков программирования и методам трансляции. Мы рассмотрим фундаментальные понятия, алгоритмы и методы, лежащие в основе теории и практики построения интерпретаторов и компиляторов языков программирования. Весь теоретический материал сопровождается практическими методами программирования соответствующих блоков компилятора или интерпретатора.

Предполагается, что читатель хорошо знаком с основными понятиями теории формальных грамматик и языков, математической логики, теории графов, теории автоматов, методами синтаксического анализа, знает основные структуры данных, используемых в программировании, имеет хорошие навыки программирования на языке C++ и опыт разработки алгоритмов, имеет опыт в написании лексических и снтаксических анализаторов языков программирования, хорощо понимает смысл контекстной зависимости языков программирования и знает методы их реализации.

Имеются две категории программистов, непосредственно связанных с языками программирования. К первой из них относятся все программисты без исключения — лица, использующие в профессиональной деятельности языки программирования для описания алгоритмов решения задач на ЭВМ. Ко второй категории относятся специалисты по системному программному обеспечению ЭВМ, которые занимаются вопросами разработки компиляторов. В связи с этим существуют три основных цели изучения курса.

Цель базовая. В современном мире существует достаточно много фирм, которые занимаются разработкой компиляторов, и выпускник должен быть готов к работе в такой фирме. Однако, очевидно, что эту работу получают очень немногие выпускники вузов.

Цель побочная, но полезная. Теоретические методы и практические навыки, применяемые при проектировании компиляторов, значительно шире задачи разработки программ обработки языков программирования.

Цель главная и принципиально важная. Прикладной программист, хорошо представляющий процессы, происходящие на уровне компиляции его программы, имеет более высокий уровень профессионализма и обладает знаниями, необходимыми как для более быстрого и качественного написания программ, так и для их отладки. Он хорошо понимает причины, которые привели к выдаче тех или иных сообщений компилятора, он четко представляет различия в процессах выполнения операций над разными типами данных, он понимает ассемблерный код, сгенерированный компилятором и может работать с этим кодом. Для профессиональных программистов это особенно важно.

Развитие теории и практики построения компиляторов шло параллельно с раз-

витием языков программирования. Первым компилятором, который давал эффективный объектный код, был компилятор с Фортрана (Бэкус и др., 1957 год). С тех пор были написаны многочисленные компиляторы, интерпретаторы, генераторы и другие программы, обрабатывающие текстовую информацию определенной синтаксической структуры. Задачей компилятора является перевод программы с языка программирования в последовательность машинных команд, выполняющую те действия, которые предполагал программист. Структура языка программирования и способ его описания оказывают основополагающее влияние на способ проектирования компилятора. Процесс компиляции можно рассматривать как взаимодействие нескольких процессов, которые определяются структурой исходного языка. Современные методы проектирования языковых процессоров различных типов базируются на методах теории формальных грамматик, языков и автоматов. Фактически все методы построения компиляторов основаны на использовании автоматных и контекстно-свободных грамматик, а синтаксический анализатор, выполняющий грамматический разбор, является ядром любого компилятора. Предлагаемое учебное пособие "Теория языков программирования и методы трансляции" следующие основные разделы:

1) интерпретируемые языки и методы однопроходной интерпретации;

2) формы представления промежуточного кода и синтаксически управляемый перевод;

3) методы оптимизации промежуточного кода;

4) генерация ассемблерного кода;

5) методы нейтрализации ошибок;

6) автоматизация проектирования трансляторов.

В соответствии со структурой курса студенты выполняют лабораторные работы, представляющие в совокупности реализацию на основе разных методов синтаксического анализа интерпретатора и компилятора очень простого языка программирования - как правило, очень усеченного подмножества языка C++ или языка Java. В частности, такое подмножество должно содержать как минимум два базовых типа данных (для реализации приведений при выполнении операций над данными разных типов), несколько операций разных приоритетов, два - три оператора (например, оператор цикла и оператор присваивания), один сложный элемент (например, описание класса или структуры, описание и использование функций).

В связи с ограниченностью объема пособия, примеры, иллюстрирующие основные понятия предмета, вынесены большей частью в упражнения и задачи. Поэтому для более полного усвоения материала необходимо уделить время реализации помещенных в конце каждой главы заданий.

Каждая глава заканчивается тестами для самостоятельной оценки знаний студентами. Среди перечисленных вариантов возможных ответов или утверждений необходимо выбрать один правильный ответ на вопрос или верное утверждение.

Студентам, желающим более подробно познакомиться с теорией алгоритмических языков и методов компиляции, можно порекомендовать книги, список которых приведен в конце пособия.

Введем обозначения, которые будем использовать в книге. Одно из основных математических понятий, которое используется для описания языков, — это контекстно–свободные грамматики (КС–грамматики).

Порождающей грамматикой называется упорядоченная четверка

$$G = (V_T, V_N, P, S), \text{ где}$$

V_T — конечный алфавит, определяющий множество терминальных символов;

V_N — конечный алфавит, опредедяющий множество нетерминальных символов;

P — конечное множество правил вывода — множество пар вида $u \to v$, где $u, v \in (V_T \cup V_N)^*$;

S — начальный нетерминальный символ — аксиома грамматики, $S \in V_N$.

Бесконтекстные или контекстно–свободные грамматики (КС–грамматики) — это грамматики, правила вывода которых имеют вид $A \to \phi$, где $\phi \in (V_T \cup V_N)^*$, $A \in V_N$.

Грамматика, не содержащая правил с пустой правой частью, называется неукорачивающей грамматикой.

Как правило, если не оговорено особо, прописные буквы латинского алфавита обозначают нетерминальные символы грамматики, а строчные буквы латинского алфавита — терминальные символы. Стандартный символ S обычно обозначает аксиому КС–грамматики.

Алфавит — непустое конечное множество. Элементы алфавита называются символами. Цепочка над алфавитом $\Sigma = \{a_1, a_2, ..., a_n\}$ есть конечная последовательность элементов a_i. Длина цепочки x — число ее элементов, обозначается $|x|$. Цепочка нулевой длины называется пустой цепочкой и обычно обозначается ε. Непустой называется цепочка ненулевой длины.

Пусть L и M — языки над некоторым алфавитом. Произведение языков есть множество $LM = \{xy | x \in L, y \in M\}$. В частности, $\{\varepsilon\}L = L\{\varepsilon\} = L$. Используя понятие произведения, определим итерацию L^* и усеченную итерацию L^+ множества L:

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

где степени языка L можно рекурсивно определить следующим образом:

$$L^0 = \{\varepsilon\},\ L^1 = L,\ L^{n+1} = L^n L.$$

Например, пусть $L = \{a\}$, тогда

$$L^* = \{\varepsilon, a, aa, aaa, \dots\},$$

$$L^+ = \{a, aa, aaa, ...\}\ .$$

В частном случае в качестве языка может выступать сам алфавит. Множество всех цепочек (включая пустую цепочку ε) над алфавитом Σ обозначается через Σ^* . Множество всех цепочек кроме пустой цепочки ε над алфавитом Σ обозначается через Σ^+ . Например, $\Sigma = \{a, b\}$, тогда $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$, $\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, ...\}$.

Конечным автоматом называется автомат без рабочей ленты, имеющий входную ленту, с которой за один такт может быть прочитан один входной символ. Конечный автомат — это шестерка вида

$$A = (K, \Sigma, \delta, p_0, F),\ \text{где}$$

K — конечное множество состояний,

Σ — алфавит,

δ — функция переходов, в общем случае — недетерминированное отображение $\delta : K \times \Sigma \to 2^K$,

p_0 — начальное состояние, $p_0 \in K$,

F — множество заключительных состояний, $F \subseteq K$.

Частным случаем конечных автоматов являются детерминированные конечные автоматы с функцией переходов $\delta : K \times \Sigma \to K$.

Известно, что для произвольного недетерминированного конечного автомата можно построить эквивалентный детерминированный.

В отличие от конечного автомата автомат с магазинной памятью (МП–автомат) имеет рабочую ленту — магазин. МП–автомат — это семерка вида

$$M = (K, \Sigma, \Gamma, \delta, p_0, F, B_0), \text{ где}$$

K — конечное множество состояний,

Σ — алфавит,

Γ — алфавит магазина,

δ — функция переходов, $\delta : K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \to 2^{K \times \Gamma^*}$,

p_0 — начальное состояние,

F — множество заключительных состояний,

B_0 — символ из Γ для обозначения маркера дна магазина.

В общем случае это определение соответствует недетерминированному автомату. В отличие от конечного автомата для произвольного недетерминированного МП–автомата нельзя построить эквивалентный детерминированный МП–автомат.

Будем использовать следующие теоретико-множественные обозначения: если A — некоторое множество объектов, то $\overline{A}$ обозначает дополнение A до N, т.е. $\overline{A} = N \setminus A$. $A = B$ означает, что A и B одинаковы как множества, т.е. A и B состоят из одних и тех же элементов; $x \in A$ означает, что x — элемент множества A. Обозначение $\{|\}$ указывает на образование множества: $\{x|...x...\}$ — это множество всех таких x, что выражение "$...x...$" верно для всех элементов этого множества.

Для заданных элементов x и y будем рассматривать упорядоченные пары $\langle x, y \rangle$, состоящие из элементов x и y, взятых именно в таком порядке. Аналогично будем использовать обозначение $\langle x_1, x_2, ..., x_n \rangle$ — для упорядоченной n–ки или кортежа длины n, состоящего из элементов x_1, x_2,..., x_n и именно в этом порядке. Через $A \times B$ обозначим декартово произведение множеств A и B, т.е. $A \times B = \{\langle x, y \rangle | x \in A, y \in B\}$. Аналогично $A_1 \times A_2 \times ... \times A_n = \{\langle x_1, x_2, ..., x_n \rangle | x_1 \in A_1, x_2, A_2, ..., x_n \in A_n\}$. Декартово произведение множества A на себя n раз обозначается A^n.

Другие общие и специальные обозначения будут вводиться по мере необходимости.

Глава 1

ИНТЕРПРЕТАТОРЫ

1.1 Принципы интерпретации

Компилятор языка программирования генерирует ассемблерный код программы, которая в дальнейшем будет выполняться. Интерпретатор, в отличие от компилятора, не генерирует код, а сразу выполняет программу. Таким образом, преимуществом процесса интерпретации является независимость программы от платформы. Реализация может осуществляться двумя способами:

• интерпретация выполняется одновременно с процессом синтаксического анализа (однопроходной метод интерпретации),

• интерпретация выполняется после синтаксического анализа, в процессе которого исходный модуль преобразуется в некоторую внутреннюю форму. В этом случае интерпретируется внутренний код.

Если интерпретатор предназначен для обработки очень простого языка программирования, можно использовать синтаксический анализатор, построенный методом рекурсивного спуска. Однопроходная интерпретация — это выполнение действий, совмещенных с семантическим контролем. Таким образом, для интерпретации необходимо либо дополнить семантические подпрограммы алгоритмами интерпретации, либо написать дополнительные семантические подпрограммы выполнения кода.

Кратко перечислим принципиальные изменения в семантических подпрограммах, необходимые для того, чтобы выполнялась интерпретация программы. Рассмотрим сначала основные изменения.

• Интерпретатор должен интерпретировать программу, выполняя вычисления над данными, следовательно, для каждой переменной *нужно хранить вычисленное значение*. Дополним каждый элемент таблицы полем, предназначенным для вычисленного значения переменной. Данные разных типов имеют различное представление в памяти ЭВМ, что необходимо учитывать как при записи вычисленных значений, так и при их обработке.

• Для выражений необходимо *реализовать вычисление* значений, а для оператора присваивания — *запись* соответствующих значений в семантическую таблицу.

• Дополнительные действия следует предусмотреть, если в интерпретируемой программе используются структуры или классы. В процессе компиляции при обработке имени типа структуры мы создавали узел в семантическом дереве. А при компиляции описания данных с именем объявленной ранеее структуры в поле типа мы просто указывали ссылку на соответствующий узел семантического дерева, в котором хранили описание структуры. При этом оказывалась, что разные идентификаторы, объявленные с одним и тем же структурным типом, ссылались на одну и ту же

вершину семантического дерева. Такая реализация семантического дерева позволяет эффективно проверить семантическую корректность всех операций над структурами, позволяет вычислить размер памяти, выделяемый для данных, дает возможность сгенерировать ассемблерный код. Однако при интерпретации такой подход неверен, поскольку не предусматривает индивидуального адресного пространства для каждого отдельного данного структурного типа. Поэтому при интерпретации описания объектов структурных типов вместо записи такой ссылки необходимо реализовать *копирование соответствующего поддерева структурного типа* в текущий узел нового объекта. Тогда в каждом узле семантического дерева будет храниться не только тип данных, но и значение, которое является собственной принадлежностью каждого объекта в программе.

• Для циклических конструкций необходимо организовать *многократные вычисления в теле цикла*. Это можно сделать только в процессе многократной обработки тела цикла. Очевидно, что однопроходной интерпретатор в этом случае имеет более низкую производительность по сравнению с компилятором или с многопроходным интерпретатором, который использует внутренний код для интерпретации.

• Появдяется необходимость пропуска некоторых фрагментов интерпретируемой программы без выполнения. Например, при интерпретации условного оператора одна из ветвей не выполняется. Тело цикла с предусловием также может быть пропущено без выполнения. Чтобы обеспечить переключение между режимами выполнения или пропуска текста ведем *флаг интерпретации* $FlInt$. Если он равен нулю, то вычисления не выполняются, и, следовательно, не производится запись вычисленных значений данных в семантическое дерево. Этот флаг не влияет на синтаксический анализ. Он только определяет необходимость вычислений.

• Если в программе есть функции, то следует обратить внимание на *различия между интерпретацией описания функции и ее вызовом*. При обработке тела каждой функции в момент ее описания интерпретация не выполняется, следовательно, флаг интерпретации $FLint$ должен быть равен нулю. Вызов функции — это переход на интерпретацию тела функции. Следовательно, чтобы выполнить функцию при обработке еч вызова, нужно в момент трансляции описания запомнить указатель лексического анализатора на текст тела функци. Тогда возврат из функции организуется восстановлением этого указателя на позицию вызова, который должен сохраняться перед вызовом функции. Кроме того, у функции и у контекста ее вызова разные области видимости, поэтому при интерпретации вызова необходимо переключать семантический контекст.

Рассмотрим теперь реализацию вышеперечисленных моментов более подробно. Поскольку процесс синтаксического анализа совмещен с процессом выполнения, то в процессе интерпретации описаний необходимо зарезервировать место для хранения значений данных. Если в языке отсутствуют массивы, то простейший вариант хранения данных заключается в следующем:

а) каждый элемент таблицы дополняется полем значения;

б) поле для хранения значений представляет собой объединение всех возможных типов данных, которые допускает язык программирования, например,

```
enum DATA_TYPE {TYPE_UNKNOWN=1, TYPE_INT, TYPE_FLOAT, TYPE_CHAR, ... };
union TDataValue  //   значение одного элемента данных
   {
   int     DataAsInt;
   float   DataAsFloat;
   double  DataAsDouble;
```

```
   bool    DataAsBool;
   ...
   };

struct TData     // тип и значение одного элемента  данных
   {
   DATA_TYPE   DataType;      // тип элементарного данного
   TDataValue  DataValue;     // значение
   };
```

Поле типа $TData$ должно не только присутствовать в каждом элементе семантического дерева, но и являться формальным параметром каждой функции, соответствующей конструкциям выражений в синтаксических диаграммах. Очевидно, что можно не описывать новый тип $TData$, а просто дополнить все типы данных, в которых участвует поле $DataType$, еще одним полем типа $TDataValue$, предназначенным для хранения вычисленных значений.

```
struct TNode    //  вершина семантического дерева
   {
   LEX id;                    // идентификатор объекта
   DATA_TYPE   DataType;      // тип элементарного данного
   TDataType   ObjectType;    // тип объекта
   TDataValue  DataValue;     // значение
   ...
   };
```

Пример внутреннего представления структуры $TNode$ в соответствии с правилами представления данных для Visual Studio C++ представлен на рисунке 1.1.

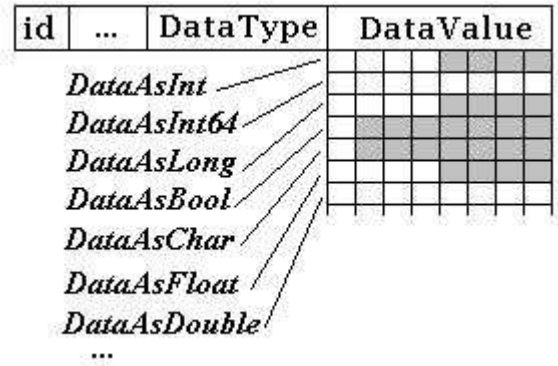


Рис. 1.1: Совмещение полей данных в семантической таблице

При наличии в интерпретируемом языке программирвоания массивов все становится сложнее из-за необходимости выделения памяти для каждого массива. Можно предложить два варианта решения этой задачи:

• Первый вариант основан на резервировании рабочей области памяти для хранения всех данных программы. В таблице данных у каждого объекта имеется поле, представляющее собой ссылку на первый элемент рабочей таблицы, начиная с которого хранятся данные объекта. Таблица реализуется с помощью резервирования области рабочей памяти и указателя на первый свободный элемент этой области. Пример представлен слева на рисунке 1.2.

• Второй и, видимо, более эффективный, но и более сложный способ можно использовать при хранении таблицы в виде динамической структуры с одновременным выделением памяти. Соответствующие действия выполняются в тех семантических подпрограммах, которые реализуют заполнение таблиц. Пример представлен справа на рисунке 1.2.

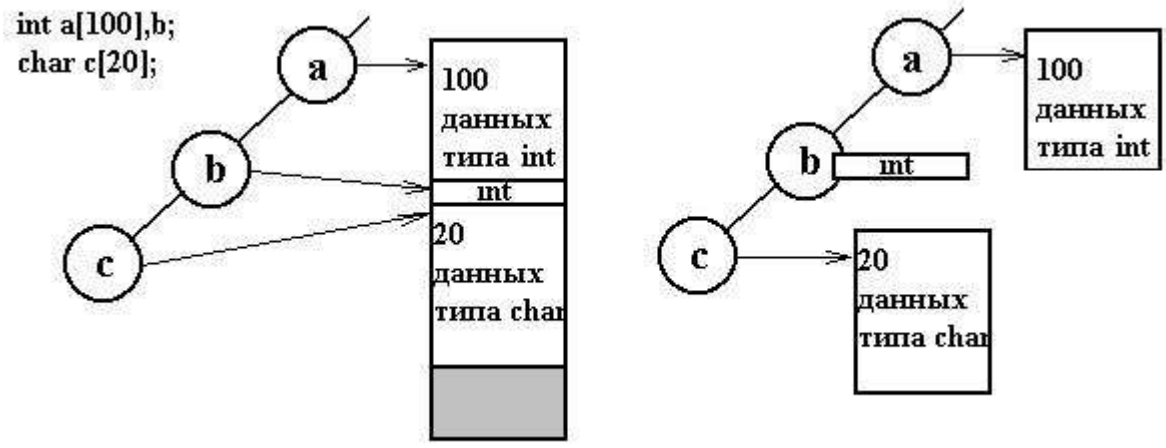


Рис. 1.2: Хранение значений в семантической таблице

1.2 Интерпретация выражений и присваиваний

1.2.1 Оператор присваивания

Выполнение оператора присваивания $a = V$ означает, что сначала нужно вычислить значение выражения V, а затем вычисленное значение сохранить в поле, предназначенном для хранения данных того типа, который определяется типом переменной в левой части оператора присваивания. Очевидно, что при этом может потребоваться приведение типов. Таким образом, функция, реализующая вычисление V, должна вернуть значение вычисленного выражения и его тип. Для этого необходимо предусмотреть следующие действия (см. рисунок 1.3):

• в точке 1 запомнить адрес переменной a в семантической таблице,

• в точке 2 получить тип и значение выражения V (обозначим эту конструкцию t),

• в точке 3 запомнить вычисленное значение для переменной a.

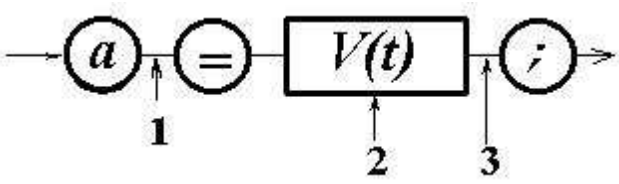


Рис. 1.3: Интерпретация присваивания

Для выполнения этих действий семантические подпрограммы в точке 1 должны вернуть тип и адрес элемента в левой части оператора присваивания. В точке

3 семантические подпрограммы выполняют операцию приведения типов, в результате чего преобразуется значение $t.DataValue$ типа $t.DataType$ к типу переменной a, полученному из семантической таблицы. Реализация точки 3 представляет собой конструкцию $switch - case$ по типам данных $DataType$ переменной a и вычисленной в V структуры t.

1.2.2 Элементарное выражение

Все данные, используемые в программе, имеют конкретные типы. Это относится как к константам, так и к переменным, в том числе к элементам массивов, к полям структур и т.п. Например, константа 100 относится к типу данных *int*, а 3.14 — к типу данных *double*. Применяемые разработчиком значения должны укладываться в допустимый диапазон значений. Например, для языка C++ Visual Studio действуют следующие ограничения, связанные с тем, что для данных разного типа в памяти существует соответствующее представление:

• int, то же самое, что signed, 4 байта, диапазон от -2,147,483,648 до 2,147,483,647;

• unsigned int, эквивалентно unsigned, 4 байта, диапазон от 0 до 4,294,967,295;

• _int8, то же самое, что char, 1 байт, диапазон от -128 до 127;

• unsigned _int8, эквивалентно unsigned char, 1 байт, диапазон от 0 до 255;

• _int16, эквивалентно short, short int и signed short int, 2 байта, диапазон от -32,768 до 32,767;

• unsigned _int16, эквивалентно unsigned short и unsigned short int, 2 байта, диапазон от 0 до 65,535;

• _int32, эквивалентно signed, signed int и int, 4 байта, диапазон от -2,147,483,648 до 2,147,483,647;

• unsigned _int32, эквивалентно unsigned и unsigned int, 4 байта, диапазон от 0 до 4,294,967,295;

• _int64, эквивалентно long long и signed long long, 8 байт, диапазон целых чисел со знаком от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807;

• unsigned _int64, эквивалентно unsigned long long, 8 байт, диапазон от 0 до 18,446,744,073,709,551,615;

• bool, 1 байт, значения false или true;

• char, 1 байт, диапазон от -128 до 127 по умолчанию или от 0 до 255 при компиляции с ключом /J;

• signed char, 1 байт, диапазон от -128 до 127;

• unsigned char, 1 байт, диапазон от 0 до 255;

• short, эквивалентно short int и signed short int, 2 байта, диапазон от -32,768 до 32,767;

• unsigned short, эквивалентно unsigned short int, 2 байта, диапазон от 0 до 65,535;

• long, эквивалентно long int и signed long int, 4 байта, диапазон целых чисел от -2,147,483,648 до 2,147,483,647;

• unsigned long, эквивалентно unsigned long int, 4 байта, диапазон целых чисел без знака от 0 до 4,294,967,295;

• long long, эквивалентно _int64, 8 байт, диапазон целых чисел со знаком от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807;

• unsigned long long, эквивалентно _int64, 8 байт, диапазон целых чисел без знака от 0 до 18,446,744,073,709,551,615;

• enum, целое число, длина в зависимости от описания типа;

• float, 4 байта, диапазон по абсолютной ведичине от 3.4E - 38 до 3.4E + 38 (7 цифр), числа со знаком;

• double, 8 байт, диапазон по абсолютной ведичине от 1.7E - 308 до 1.7E + 308 (15 цифр), числа со знаком;

• long double, эквивалентно double;

• wchar_t, эквивалентно _wchar_t, 2 байта, диапазон от 0 до 65,535.

При интерпретации необходимо отводить память для данных в соответствии с принятым стандартом и выполнять вычисления в соответствии с типами. Процессор выполняет разные команды над разными данными, именно по этой причине в языках программирования существует понятие приведения типов. Например, если разделить целое число 11 на целое число 2, то получим 5, а не 5.5. Более того, из-за ограниченности разрядной сетки в компьютере невозможно представить вещественные числа: числа с плавающей точкой не являются вещественными числами. Для демонстрации этого факта рассмотрим программу

```
int intData = 2147483647;
float floatData = intData;
double doubleData  = intData;

void iteration(int k){
    printf("After %d iterations:\n",k);
    for (int i=1; i <= k; i++)  {
       intData = intData-1;
       floatData = floatData-1;
       doubleData = doubleData-1;
    }
    printf ("intData-%d =    %d   \n",  k, intData);
    printf ("floatData-%d =  %f   \n",  k, floatData);
    printf ("doubleData-%d = %llf\n\n", k, doubleData);
}

int main(){
    printf ("intData =    %d \n",   intData);
    printf ("floatData =  %f \n", floatData);
    printf ("doubleData = %llf  \n\n", doubleData);

    printf ("intData-64 =    %d  \n",  intData-64);
    printf ("floatData-64 =  %f \n", floatData-64);
    printf ("doubleData-64 = %llf \n\n", doubleData-64);

    printf ("intData+1 =   %d     \n",  intData+1);
    printf ("floatData+1 =  %f   \n",  floatData+1);
    printf ("doubleData+1 = %llf\n\n",  doubleData+1);

    printf ("intData+2 =  %d     \n",  intData+2);
    printf ("floatData+2 = %f   \n",  floatData+2);
    printf ("doubleData+2 = %llf\n\n",  doubleData+2);

    iteration(483647);
```

```
    return 0;
}
```

В результате выполнения программы получим следующий вывод:

```
intData =    2147483647
floatData =  2147483648.000000
doubleData = 2147483647.000000

intData-64 =    2147483583
floatData-64 =  2147483584.000000
doubleData-64 = 2147483583.000000

intData+1 =   -2147483648
floatData+1 =  2147483649.000000
doubleData+1 = 2147483648.000000

intData+2 =  -2147483647
floatData+2 = 2147483650.000000
doubleData+2 = 2147483649.000000

After 483647 iterations:
intData-483647 = 2147000000
floatData-483647 = 2147483648.000000
doubleData-483647 = 2147000000.000000
```

Например, во второй строчке выведено число типа $float$, которое не равно исходному целому числу, так как мантисса из семи цифр не может точно представить число с десятью знаками. Проанализируйте все выведенные значения и объясните, почему выведено то или иное число.

При реализации интерпретаторы Вы должны помнить, что числа с плавающей точкой — это приближения вещественных чисел, при их использовании неизбежно появление ошибки. Эта ошибка тем меньше, чем большая длина отведена для хранения мантиссы числа. Эта ошибка является ошибкой округления и может привести к результатам, вычисления которых проходили с большой погрешностью. Числа с плавающей точкой имеют фиксированную точность — 24 двоичных разряда для чисел $float$ и 53 двоичных оазряда для чисел $double$. Это означает, что существует интервал между двумя последовательными числами типа $float$ или $double$. Зная интервал между двумя последовательными числами в окрестности некоторого числа с плавающей точкой можно избежать классических ошибок в вычислениях.

Перейдем к реализации интерпретатора выражений. Интерпретация выполняется как для элементарных выражений, так и для выражения с выполнением бинарных и унарных операций. Если элементарное выражение содержит только простые переменные и константы (рисунок 1.4), интерпретация выполняется просто.

• В точке 4 дополнительно к стандартным действиям по контролю описания переменной выполняется выборка значения $t.DataValue$ и типа $t.DataType$ из семантической таблицы.

• В точке 5 реализуется преобразование константы из символьного представления во внутреннее представление, а затем запись этого значения в нужное поле $t.DataValue$ в соответствии с типом этой константы $t.DataType$. Конвертирование числа из символьной формы в числовую можно с помощью функций $atoi(x)$, $atof(x)$, ${}_a todbl(x)$ и других аналогичных функций из $< stdlib.h >$.

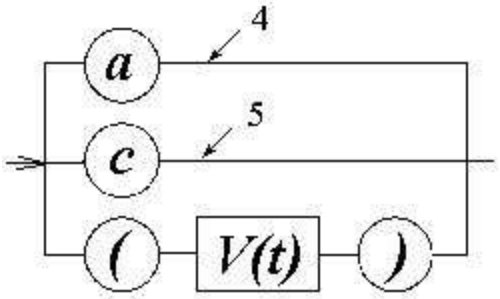


Рис. 1.4: Интерпретация элементарного выражения

1.2.3 Бинарная и унарная операция

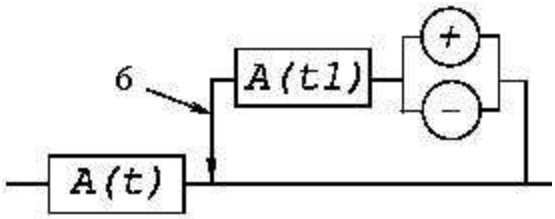


Рис. 1.5: Интерпретация бинарной операции

Если перед первым выражением A имеется унарная операция, например, "плюс" или "минус", то достаточно сохранить знак операции, а затем после вычисления A реализовать действия, соответствующие унарной операции. При этом, как всегда, проверяется допустимость указанной операции для данного вычисленного типа.

Бинарная операция выполняется несколько сложнее: фактически, например, для операции "плюс" надо вычислить значение выражения

$$t.DataValue = t.DataValue + t1.DataValue$$

с одновременным приведением типов. Для этого нужно выполнить следующую последовательность действий:

а) вычислить тип результата операции и преобразовать (как правило, только одно из значений) $t.DataValue$ типа $t.DataType$ или $t1.DataValue$ типа $t1.DataType$ к общему типу;

б) выполнить операцию над преобразованными данными.

Ниже приведен простейший пример реализации соответствующих действий, основанный на использовании совокупности операторов *switch* для выбора требуемой операции, которая должна выполняться над данными соответствующих типов:

```
switch (t.DataType)          // тип первого операнда
```

```
{
case TYPE_INT:                // первый операнд int
  switch (t1.DataType)        // тип второго операнда
    {
    case TYPE_INT:
         if (ll == TPlus)             // int += int
            t.DataValue.DataAsInt += t1.DataValue.DataAsInt;
         else  if (ll == TMinus)      // int -= int
            t.DataValue.DataAsInt -= t1.DataValue.DataAsInt;
         else    ...
         break;
    case TYPE_FLOAT:
         t.DataType = t1.DataType;    // тип результата изменился
         if (ll == TPlus)             // int += float
            t.DataValue.DataAsInt += t1.DataValue.DataAsFloat;
         else  if (ll == TMinus)      // int -= float
            t.DataValue.DataAsInt -= t1.DataValue.DataAsFloat;
         else    ...
         break;
    ...
    }
case TYPE_FLOAT:            // первый операнд float
  switch (t1.DataType)      // тип второго операнда
    {
    case TYPE_INT:
         if (ll == TPlus)           // float += int
            t.DataValue.DataAsFloat += t1.DataValue.DataAsInt;
         else  if (ll == TMinus)    // float += int
            t.DataValue.DataAsFloat -= t1.DataValue.DataAsInt;
         else    ...
         break;
    case TYPE_FLOAT:
         if (ll == TPlus)           // float += float
            t.DataValue.DataAsFloat += t1.DataValue.DataAsFloat;
         else  if (ll == TMinus)    // float -= float
            t.DataValue.DataAsFloat -= t1.DataValue.DataAsFloat;
         else    ...
         break;
    ...
    }
 ...
}
```

1.3 Интерпретация условных операторов

Интерпретация условных операторов построена на основе управления флагом интерпретации $FlInt$, который определяет режим выполнения: при значении $FlInt = 1$ выполняются операторы, а при значении $FlInt = 0$ осуществляется только синтакси-

ческий анализ. При вычислении значения этого флага необходимо помнить о рекурсивной структуре программы: внутри оператора if могут быть вложенные операторы if или любые другие операторы, которые выполняются или не выполняются в зависимости от некоторых условий, вычисляемых в процессе интерпретации. Поэтому следует использовать локальную переменную, которая сохранит значение $FlInt$ перед началом интерпретации условного оператора и затем будет использована для восстановления исходного значения при завершении интерпретации. Реализация интерпретации оператора if представлена на рисунке 1.6.

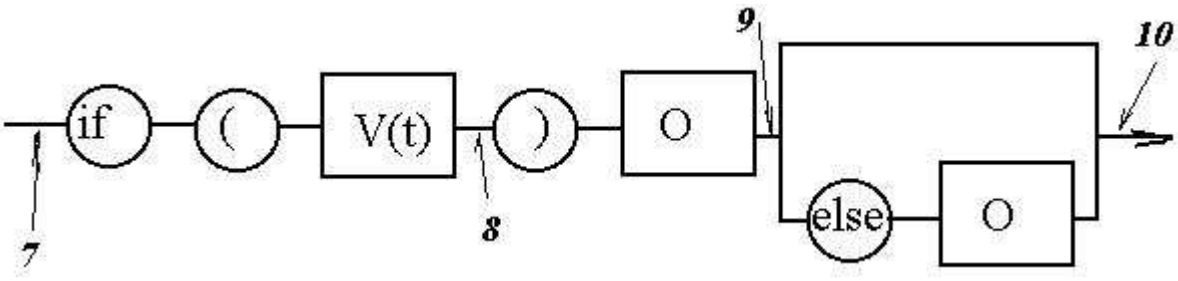


Рис. 1.6: Интерпретация условного оператора

Рассмотрим подробнее выполняемые действия:

```
// Точка 7:  сохраняем текущее значение флага интерпретации
int LocalFlInt = FlInt; // локальный флаг интерпретации

// Точка 8: вычисляем новое значение флага интерпретации
if (FlInt  &&  (t.DataValue.DataAsInt != 0) ) FlInt =1;
    else FlInt = 0;

// Точка 9: инвертируем значение флага интерпретации,
//          если исходное значение флага интерпретации равнялось 1
if ( LocalFlInt )   FlInt = 1- FlInt

// Точка 10: восстанавливаем исходное значение флага интерпретации
FlInt = LocalFlInt;
```

Аналогичные принципы переключения флага интерпретации используются при интерпретации оператора *switch*.

1.4 Интерпретация операторов цикла

Как известно, в языках программирования циклы бывают различных типов: с предусловием или с постусловием, с указанием параметров цикла или без таковых. Рассмотрим сначала обычный оператор цикла с предусловием. Реализация интерпретации оператора *while* представлена на рисунке 1.7.

Фактически, интерпретация цикла заключается в управлении флагом интерпретации и в организации перехода на повторное выполнение тела цикла. Рассмотрим выполняемые действия на примере цикла с предусловием:

```
// Точка 11:  сохраняем текущее значение флага интерпретации
```

```
int LocalFlInt = FlInt; // локальный флаг интерпретации

// Точка 12: ставим метку начала вычисления выражения
//           и запоминаем положение указателя в тексте исходного модуля
int uk1 = GetUK();
Start:

// Точка 13: Вычисляем значение флага интерпретации
//           в соответствии с исходным значением  FlInt и
//           с вычисленным значением выражения.
if (FlInt  &&  (t.DataValue.DataAsInt != 0) ) FlInt = 1;
    else FlInt = 0;

// Точка 14: Организуем повторное выполнение цикла при
//           установленном флаге интерпретации: восстанавливаем UK
//           и переходим к началу  на метку Start.
//           При завершении цикла
//           восстанавливаем значение флага интерпретации.
if ( FlInt )
     {  PutUK(uk1); goto Start; }
FlInt = LocalFlInt;
```

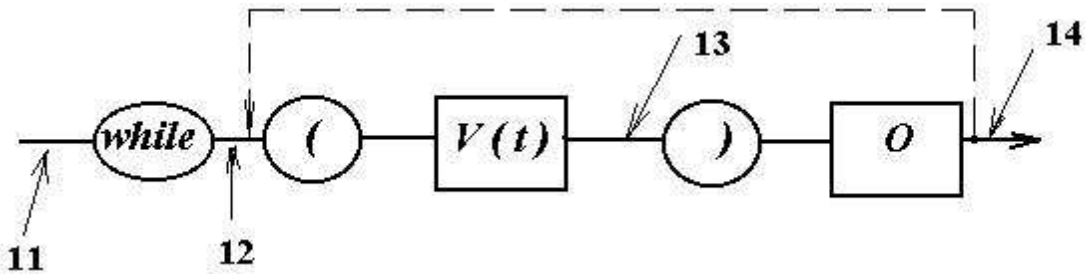


Рис. 1.7: Интерпретация циклического оператора

Аналогично интерпретируются другие операторы цикла $do - while$, for и т.п. Видимо, наибольшую сложность представляет интерпретация оператора for языков C++ или Java в силу того, что необходимо организовать довольно сложную цепочку переходов между выражениями внути круглых скобок. Одним из вариантов реализации требуемой последовательности переходов является преобразование синтаксической диаграммы к виду, соответствующему требуемой последовательности вычислений. На рисунке 1.8 представлена последовательность вычислений, а на рисунке 1.9 — соответствующий этой последовательности вариант преобразованной диаграммы.

1.5 Интерпретация функций

Если язык программирования допускает использование функций, то при реализации интерпретатора нужно решить следующие задачи:

- выбрать способ выделения памяти для внутрених данных функций,
- исключить интерпретацию тел функций при обработке описания функции,

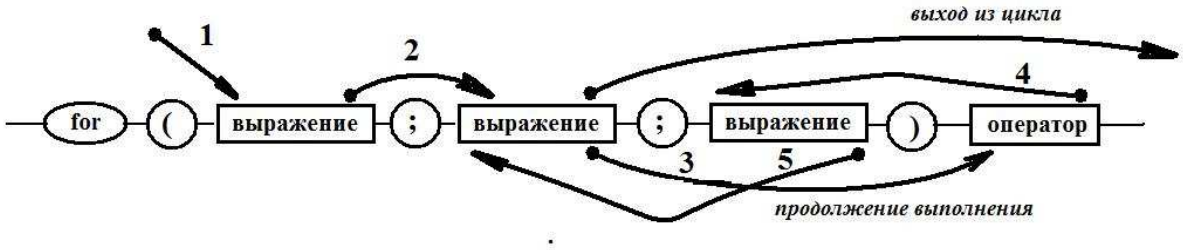


Рис. 1.8: Последовательность операций при интерпретации оператора for

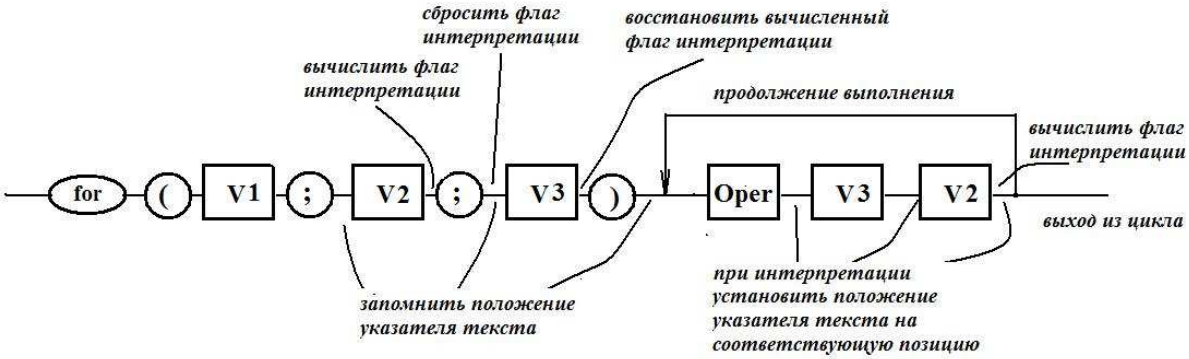


Рис. 1.9: Интерпретация оператора for

• выбрать способ передачи фактических параметров при вызове функции,

• установить область видимости, доступную для тела функции, а затем вернуться к исходной области видимости;

• перейти к выполнению тела функции при вызове функции, а затем вернуться в точку вызова.

Решение первых двух задач легко организовать, если использовать флаг интерпретации $FlInt$. Алгоритм занесения в семантическое дерево имени функции и ее параметров был уже реализован при решении задачи семантического контроля. Следовательно, *при трансляции заголовка* функции нужно установить флаг $FlInt$, чтобы было построено семантическое дерево всех параметров функции. *При трансляции тела* функции нужно сбросить флаг $FlInt$, тогда операторы выполняться не будут.

Следующий вопрос, на который следует дать ответ — когда и как заносить в семантическое дерево локальные данные тела функции. Реализация существенно зависит от структуры языка программирования. В языке Паскаль и ему подобных в теле функции нет описаний данных, поэтому интерпретация всех локальных описаний может быть выполнена в процессе обрабоки функции. В результате будет построено семантическое дерево всех локальных данных (переменных, типов, функций и т.п.), которое можно использовать при интерпретации тела функции. Синтаксически описания в языке Псакаль отделены от операторов, поэтому описания могут быть обработаны только один раз и многократно использоваться.

При интерпретации языка C++, Java и им подобных все гораздо сдожнее. Описания и операторы синтаксически не разделены, поэтому наиболее простой (но, конечно, не единственный) подход к реализации функций основан на использовании флага $FlInt$ так, чтобы при сброшенном $FlInt$ все семантические подпрограммы не выполняли бы никаких действий. Достаточно сбросить в нуль флаг $FlInt$ при обработке тела функции, и тогда все локальные данные в семантическое дерево не попадут (рисунок 1.10). В семантической таблице необходимо запомнить указатель на на-

чало тела функции, которое будет интерпретироваться при выполнении оператора вызова.

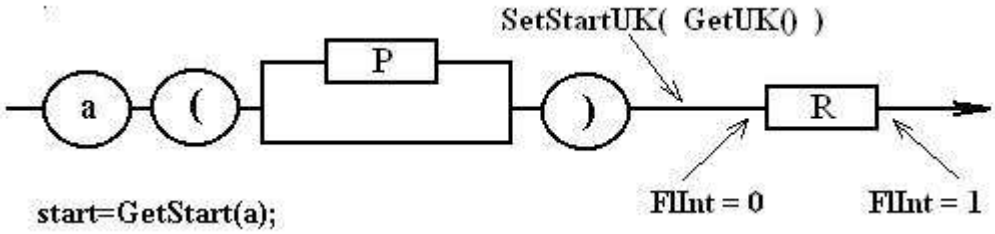


Рис. 1.10: Интерпретация описания функции

Рассмотрим теперь интерпретацию вызова функции. Здесь следует решить три проблемы:

- как передать параметры,
- как выполнить тело функции,
- как вернуть результат вычисления функции.

Самым простым является решение последнего вопроса: в семантическом дереве вершина, соответствующая идентификатору функции, имеет поле $DataValue$, в которое можно записывать возвращаемое значение.

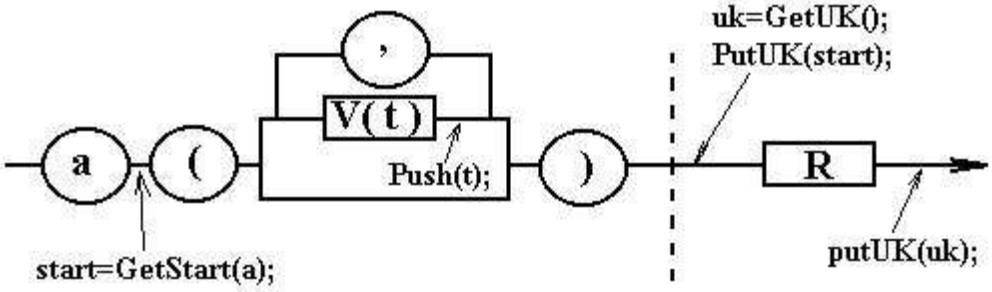


Рис. 1.11: Интерпретация вызова функции

Вопрос о передаче параметров решается по-разному в зависимости от языка программирования. Если не допускаются рекурсивные вызовы функций, то передача параметров осуществляется простым копированием вычисленных значений фактических параметров в последовательные элементы семантического дерева, которые в семантическом дереве расположены по правой ссылке от имени вызываемой функции. Эти вершины дерева как раз и являются формальными параметрами функции. Если же язык программирования допускает рекурсивные вызовы, то с необходимостью должен моделироваться стек. Сделать это можно разными способами. Прежде, чем обсуждать реализацию рекурсивных вызовов, рассмотрим еще одну проблему, связанную с реализацией вызова функции — это проблема корректной области видимости. Допустим, нам нужно интерпретировать следующий код:

```
int a = 0, n = 5;
double  b = 1, c = 2;

double func(double a, int n){
   double b = 1;
       // здесь глобальная переменная c = 2,
```

```
        // a и n - параметры функции
        // b  - локальная в  func  переменная
    for (int i=0; i < n; i++)
        b *= a + c;
    return b;
}

int main(){
    double a = 0;
    int b = 10, c = 20;
        // здесь a, b, c - локальные в  main переменные, c = 20
        // n -  глобальная переменная
    a = func (b-c/b, n);
    printf ("a = %f\n", a);
    return 0;
}
```

Программа должна вывести значение переменной a, равное 100000.00000. Однако, если Вы неправильно устновите область видимости в момент интерпретации тела тела функции, программа может вывести другое значение. На рисунке 1.12 показано, какие именно переменные видимы в той или иной точке программы. Это означает, что наряду с позицией в тексте интерпретируемой программы необходимо устанавливать текущий указатель семантического дерева. Таким образом, контекст точки вызова и интерпретируемого тела функции определяется двумя указателями — исходного кода и семантического дерева.

Перейдем теперь к реализиции рекурсивного вызова. При интерпретации рекурсии мы должны предусмотреть стековую природу как передаваемых функции параметров, так и всех локальных данных функции. Кроме того, необходимо обеспечить правильную область видимости для каждого вызова. Сделать это можно, если при реализации очередного вызова создавать новую копию заголовка после предшествующего заголовка этой же функции (рисунок 1.13), а после выхода из функции уничтожать созданное поддерево.

Рассмотрим теперь процесс выполнения тела функции. Синтаксически тело функции — это составной оператор. Таким образом, выполнить функцию — это выполнить тот составной оператор, который физически находится в интерпретируемой программе после заголовка функции. Это значит, что к синтаксической диаграмме вызова функции нужно добавить составной оператор, выполнение которого и означает выполнение вызова функции. Резюмируя все вышесказанное можно перечислить требуемые действия:

1) при интерпретации описания каждая процедура и функция в семантической таблице должна хранить значение указателя на тело функции;

2) при вызове копируется поддерево функции вместе с поддеревом формальных параметров;

3) текущий указатель текста и текущий указатель семантического дерева устанавливается на позиции, соответствующие функции;

4) выполняется составной оператор;

5) после завершения тела функции следует вернуть указатели в точку вызова;

6) и, наконец, при завершении интерпретации вызова следует удалить созданную копию семантического поддерева функции.

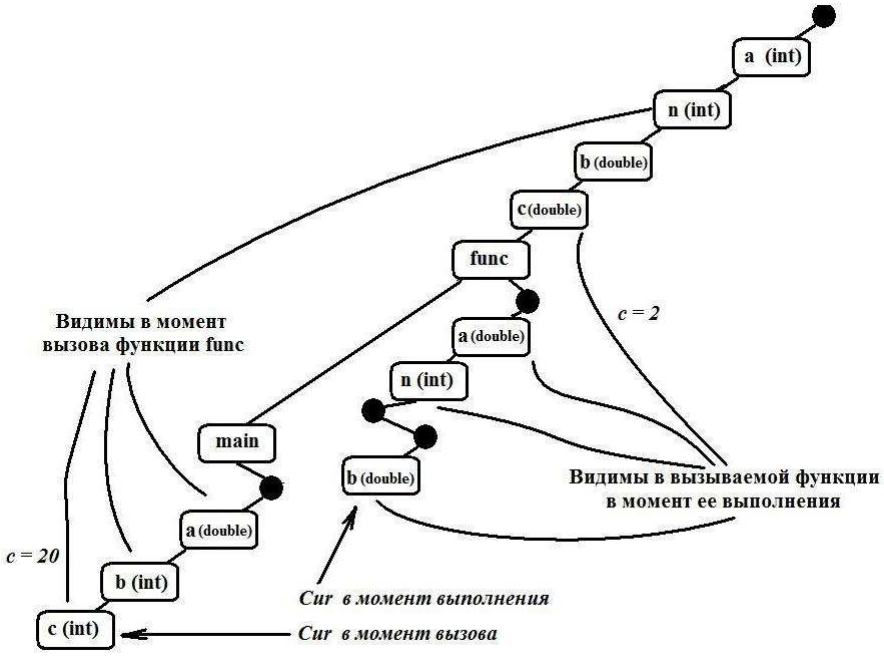


Рис. 1.12: Восстановление контекста при вызове функции

1.6 Интерпретация составных операторов

В теле любого блока (или составного оператора), в частности тела функции, могут объявляться локальные даные. Память для размещения этих данных нужно выделить в начале блока и освободить при его завершении. Существуют разные варианты выделения памяти. Трансляторы, как правило, встраивают в начало тела любой функции некоторую стандартную последовательность команд или вызов специальной подпрограммы, которая называется *прологом блока* и предназначена для выделения памяти. Перед командой выхода *ret* выделенная память освобождается. Для этой цели может быть предназначена специальная подпрограмма, которая называется *эпилогом блока*. Главная задача пролога — выделить память для локальных данных, а задача эпилога — освободить эту память. Как правило, трансляторы в эпилоге функции выделяют всю память целиком, включая и память для данных внутренних блоков. В таком случае в начале составного оператора вызов пролога не ставится, что повышает быстродействие оттранслированной программы. При трансляции это легко сделать, так как после завершения работы синтаксического анализатора в семантическом дереве имеется вся информация обо всех внутренних данных функции.

При интерпретации такое решение использовать очень сложно, так как при обработке каждого отдельного блока потребуется знать, анализируется он впервые или нет. Точнее, нужно знать, построено для блока семантическое дерево или еще нет. Гораздо проще при интерпретации каждого блока всегда выполнять создание семантического дерева этого блока, а при выходе из блока — его уничтожение. Пролог блока в этом случае — запомнить текущий указатель на семантическое дерево, эпилог — удалить все семантическое поддерево, которое было построено в блоке. Эффективность такой реализации ниже, но зато существенно упрощен алгоритм интерпретации. В качестве иллюстрации низкой эффективности можно привести пример многократного резервирования памяти в результате построения одного и того же дерева, если

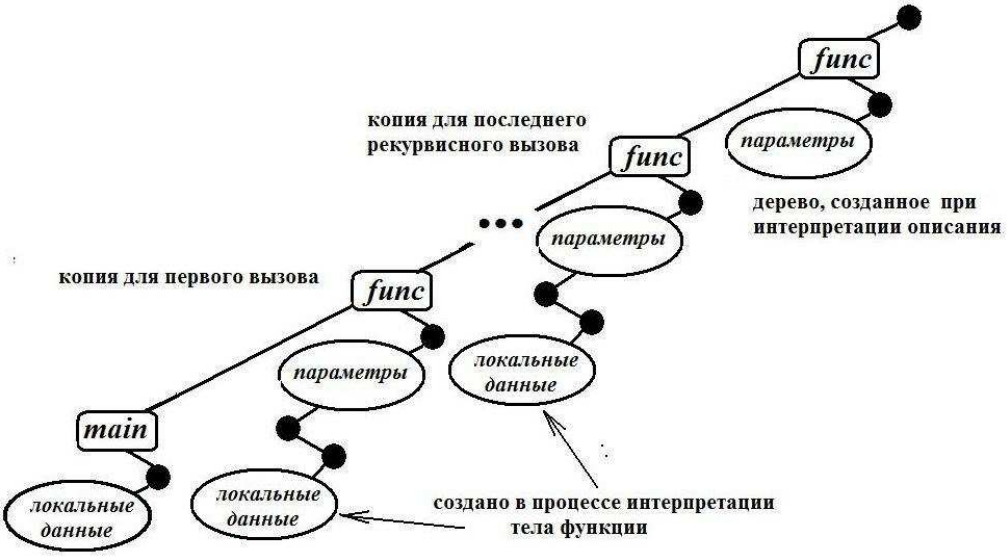


Рис. 1.13: Семантическое дерево при рекурсивном вызове функции

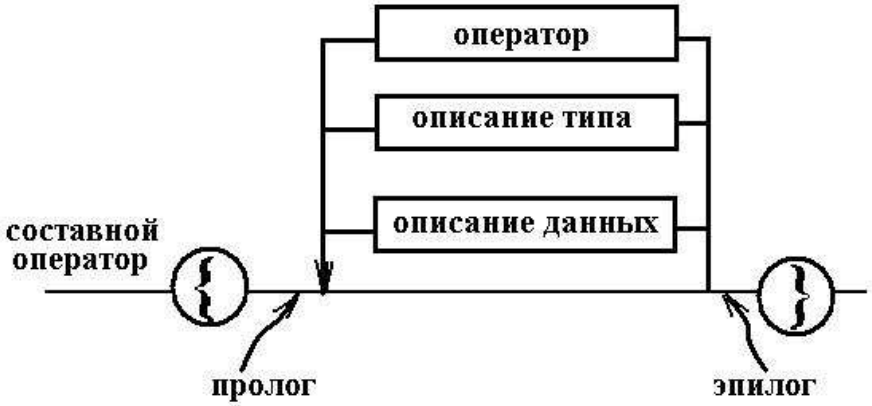


Рис. 1.14: Составной оператор языка C++

телом цикла служит составной оператор.

1.7 Интерпретация меток и операторов перехода на метки

Несмотря на то, что использование операторов *goto* по общему мнению не является признаком хорошего стиля программирования, наличие этого оператора в языке программирования обязывает нас рассмотреть алгоритм его интерпретации. Семантика метки — пометить некотрый фрагмент кода. Не на каждую метку обязан существовать переход, но каждый переход должен быть на существующую в программе метку. Это означает, что метка является семантическим объектом программы и должна быть занесена в семантическое дерево. Любая метка может использоваться в двух контекстах:

- она может помечать оператор,
- метка используется в операторе *goto*.

В зависимости от того, как расставлены метки в исходном модуле, различают

ссылку вперед (метка стоит после оператора *goto*) и ссылку назад (метка стоит в программе до того, как используется переход *goto* на эту метку). Для каждой метки в семантическом дереве должно храниться ее имя, указатель текста в исходном модуле и флаг, который означает признак обработки соответствующей метки в тексте (вместо этого флага можно установить указателю значение -1, что означает факт отсутствия в текущий момент данной метки).

Рассмотрим сначала интерпретацию оператора перехода. Если внутри текущего блока осуществляется переход назад *goto* $LabelName$, то в дереве уже имеется вершина, соответствующая $LabelName$ с установленным значением указателя. Тогда при интерпретации оператора *goto* устанавливается текущий указатель текста по указателю из таблицы на $LabelName$. При этом мы можем выйти из одного или сразу нескольких уровней вложенности составных операторов. Это означает, что необходимо удалить из семантического дерева все соответствующие этим составным операторoам поддереья.

При интерпретации перехода вперед сбрасывается $FlInt$ и при обнаружении какой-либо метки внутри текущего блока необходимо проверить ее совпадение с $LabelName$, чтобы при их совпадении устанавить $FlInt$. Это означает, что имя $LabelName$ должно где–то храниться в качестве глобального данного. Кроме того, должен существовать глобальный признак необходимости проверки метки при интерпретации оператора *goto*. При этом при переходе вперед имеется еще одна неприятная особенность: так же, как и переходе назад, переход вперед может быть из блока наружу. Это означает, что все данные, которые были внесены в дерево после метки $LabelName$, должны быть уничтожены в семантическом дереве. Мы знаем, что операция уничтожения поддерева блока всегда выполняется при достижении конца этого блока при установленном флаге $FlInt$. Но ведь в нашем случае этот флаг сброшен! Наличие метки выхода из блока усложняет логику проверки необходимости удаления дерева блока. Все эти сложности интерпретации меток еще раз подтверждают тот факт, что метки — далеко не лучшее изобретение программистов не только с точки зрения хорошего стиля программирования, но и с точки зрения интерпретации программы.

1.8 Многомерные массивы и структуры

Любые массивы представлены в оперативной памяти линейным вектором. Для реализации доступа к заданному элементу массива необходимо вычислить смещение соответствующего элемента относительно начала массива. Чтобы многомерный массив развернуть в линейный, необходимо выбрать принцип размещения элементов. Существуют два варианта размещения массивов в памяти, которые условно называют "по строкам" или "по столбцам". Иначе говоря, выбирается способ, при котором быстрее растет последний индекс элемента массива или быстрее растет первый индекс. На рисунке 1.15 вверху представлено размещение массива по строкам, когда быстрее растет последний индекс, а внизу — по столбцам, когда быстрее растет первый индекс.

Тогда, например, для трехмерного массива $b[N1][N2][N3]$, размещенного в памяти по строкам, смещение элемента $b[i][j][k]$ равно $((i*N2)+j)*N3+k$. Леворекурсивный характер грамматики списка индексов определяет леворекурсивную формулу смещения: на каждом шаге умножается текущее выражение на соответствующее измерение

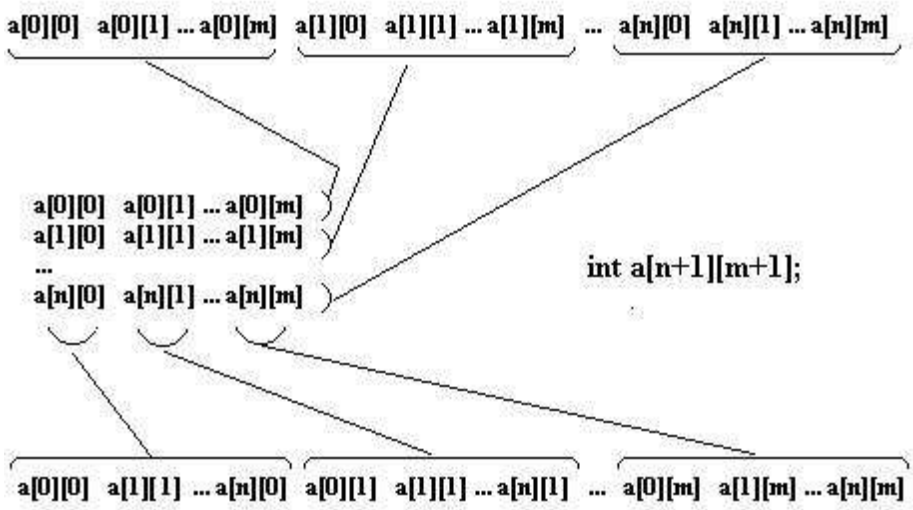


Рис. 1.15: Размещение массивов в памяти

и добавляется новый индекс:

элемент массива	смещение
$A[i_0]$	$S_0 = i_0$
...	
$A[i_0][i_1]...[i_{k+1}]$	$S_{k+1} = S_k * N_{k+1} + i_{k+1}$

Очевидно, что при вычислении реального адреса в байтах смещение S_i нужно умножить на $sizeof(A[i_0][i_1]...[i_{k+1}])$.

И в заключение рассмотрим структуры. Если в языке программирования структуры допускаются как пользовательские типы данных, то в семантическом дереве такие типы хранятся в виде вершины с соответствующим поддеревом. При интерпретации, чтобы каждый объект этого типа имел собственное значение, необходимо вместо копирования ссылки на это поддерево, скопировать в качестве правого потомка все поддерево целиком. В результате все объеты будут иметь собственные уникальные поля для хранения данных.

1.9 Влияние принципа интерпретации на архитектуру языка программирования

1.9.1 Язык *JavaScript* как пример интерпретируемого языка

Широкое распространение сети Интернет приводит к необходимости создания страниц с содержимым, которое выполняется на стороне клиента. *JavaScript* — это кросс–платформенный объектно–ориентированный язык создания скриптов. На клиентской стороне *JavaScript* расширяет возможности поддержки интерактивных элементов в окне браузера, позволяет работать с объектной моделью документа (DOM) и обрабатывать события. Браузер может интерпретировать операторы языка *JavaScript*, вставленные на страницу $HTML$. Когда клиент запрашивает такую

страницу, сервер посылает клиенту полное содержимое документа, включая $HTML$ и операторы $JavaScript$. Браузер читает страницу сверху вниз, отображая результаты интерпретации $HTML$ и операторов $JavaScript$. Результатом этого является информация в окне просмотра браузера. Сценарий на языке $JavaScript$ может изменять содержимое документа в процессе его загрузки или при возникновении некоторого события.

Сценарий заключается между тегами $< SCRIPT >$ и $< /SCRIPT >$. Если браузер не поддерживает сценарии, он должен проигнорировать скрипт. Поэтому для исключения обработки непонятного браузеру сценария текст сценария заключается в комментарии. Комментарий — это произвольный текст между $<! --$ и $-- >$.

Язык $JavaScript$ по своему синтаксису напоминает языки $C++$ и $Java$, содержит обычные операторы if, $switch$, $while$, $do...while$,, for; допускает использование простых переменных, массивов, структур, функций; позволяет описывать классы и объекты различных классов; в выражениях разрешаются все обычные операции.

1.9.2 Типы данных языка $JavaScript$

Как любой достаточно развитой язык программирования язык $JavaScript$ имеет блочную структуру программы. Глобальные переменные, описанные вне какой–либо функции, доступны из любого места документа. Переменные объявляются либо неявно, либо явно оператором

$$var < \text{список переменных} >;$$

Неявное объявление реализуется при интерпретации присвоения вычисленного значения такой переменной, имя которой явно не объявлено. Интерпретируемый характер языка $JavaScript$ позволяет не объявлять типы переменных. Как мы знаем, в момент выполнения операции присваивания уже известен тип вычисленного выражения. Этот тип и присваивается переменной. Фактически тип переменной в каждый конкретный момент времени определяется как эквивалент типа ее значения. Таким образом, в процессе интерпретации одна и та же переменная может принимать значения разных типов. Этот же принцип работает как для формальных параметров функций, так и для типов значений, которые возаращают функции. Следовательно, параметры функций объявляются без типов и у функции нет описания возвращаемого типа. Функции объявляются конструкцией

```
function(<список имен формальных параметров>) {
 <операторы и описания>
}
```

Например,

```
function calculationString(x) {
    var y= 5;
    var result = x + y + y + (x +"  Hello!");
    return result;
}
```

Операции (аналог операций языков C++ или Java) выполняются слева направо с приведением типов

$$int \rightarrow float \rightarrow string,$$

поэтому, например, результатом выполнения функции

$$calculationString(12)$$

будет строка со значением "2212 Hello!", а результатом выполнения функции

$$calculationString("Hi!")$$

— строка "Hi! 55Hi! Hello!".

Очевидно, что при таких правилах приведения типов нельзя автоматически преобразовать строковое значение в числовое. Для этого необходимо использовать явное преобразование с помощью функций $parseInt(x)$ и $parseFloat(x)$.

1.9.3 Классы и объекты пользователя языка *JavaScript*

Рассмотрим описание классов на языке *JavaScript*. В силу своего предназначения для описания простых сценариев язык *JavaScript* не должен обладать сложным синтаксисом, поэтому в основу выбора конструкции для описания классов создатели языка взяли за основу функции. Как строится семантическое дерево для класса? Корнем дерева является вершина с именем класса, а все переменные и методы — потомками этой вершина. Но точно такой же вид имеет и вершина для функции: имя функции — в корне дерева, а параметры и внутренние данные — ее потомки. Причем нет ограничений на тип внутренних объектов функции, они могут являться как данными, так и функциями. Следовательно, есть возможность объявить класс в форме функции. Более того, аргументы функции можно рассматривать как фактические параметры конструтора при создании нового объекта заданного класса. Ниже приведен пример описания класса с именем $TMyClass$, конструктор которого имеет два формальных параметра. Значения этих параметров конструктор прсвиавает двум переменным x и y. Класс имеет три метода $Funct1$, $Funct2$, $Funct3$, которым динамически присваиваются конкретные реализации в виде функций соответственно Sum, Mul, $SumDub$.

```
<script language="JavaScript">
<!-- hide

function Sum(){  // будущий метод некоторого класса
   return this.x+this.y;
}

function SumDub(){  // будущий метод некоторого класса
   return this.x*this.x+this.y*this.y;
}

function Mul(a,b)M  // тоже будущий метод
   return this.x * this.y;
}

function TMyClass(a,b)  // класс
{
   this.x= a;
```

```
   this.y= b;
   this.Funct1 = Sum;
   this.Funct2 = Mul;
   this.Funct3 = SumDub;
}

   //  объявим объекты класса TMyClass
var a1 = new TMyClass(2,3);
var a2 = new TMyClass(10,-3);

alert(a2.Funct1());  // выполнение метода класса
// -->
</script>
```

1.9.4 Регулярные выражения языка *JavaScript*

В языке *JavaScript* определены предопределенные классы с именами *Array*, *Boolean*, *Math*, *Number*, *String*, *Date*, *RegExp*.

Наиболее интересным с точки зрения практического применения является класс регулярных выражений *RegExp*. Создать регулярное выражение можно двумя способами:

• используя принцип присваивания объекту типа при интерпретации оператора присваивания, например, выполняя инициализацию объекта с помощью оператора $reg2 = /12 * 3 * /$;

• выполняя явный вызов конструктора регулярных выражений, например, $reg1 = newRegExp(”1 + 2 * ”)$;.

Как регулярное выражение в операторе инициализации, так и конструктор регулярных выражений содержат две строки в качестве параметров:

$$/pattern/[g|i|gi],$$

где *pattern* — изображение регулярного выражения, а второй параметр может отсутствовать и представляет собой набор дополнительных флагов, которые используются при поиске на основе регулярных выражений. Флаг g указывает глобальный характер поиска, а флаг i — поиск без учета регистра. Эти флаги могут использоваться отдельно или вместе в призвольном порядке. Заметим, что флаги, i и g — неотъемлемая часть регулярного выражения. Они не могут быть добавлены или удалены позже. В записи регулярного выражения допускаются обычные операции над языками (объединение, итерация, усеченная итерация), а для обозначения символов используются либо их изображения, либо специальные обозначения:

```
  // конструктор регулярных выражений:
reg1 = new RegExp("1+2*","g");
reg2 = /12*3*/g;
alert(reg2);
       // специальные символы
              // + операция усеченной итерации
              // * операция итерации
              // | операция объединения
reg4 = /^A*/;
```

```
    // ^ означает начало строки (например, в Abcd,  но не в dcdA)
reg4 = /A*$/;
    // $ означает конец строки (например, в bcdA,  но не в Adcd)
reg4 = /\+?1\+/ ;
    // ? - символ встречается один или ноль раз
reg4 = /...12.22/ ;
    // точка означает любой символ, кроме символа перехода на новую строку
reg4 = /{1|2|3|4|5|6|7|0}*/;
    // разделитель | означает "или"
reg4 = /[a-f]/;
    // символы  " от ... до": a-f = a|b|c|d|e|f
reg4 = /[0-9]/;
    // \d - любая цифра    \D - любой символ, но не цифра
reg4 = /[0-9]*/;
    // иначе reg4=/\d*/;
reg4 = /[abcdef]/;
    // перечисленные символы
reg4 = /[^abc]/;
    // НЕ перечисленные символы
reg4 = //;
    // [\b] пробел
    //  \b - граница между словами (пробел или перевод строки)
    // [ \f\n\r\t\v] - пропуски (соответственно, конец файла,
    //                 конец строки, перевод каретки, табуляция)
    // \s - любой из вышеперечисленных символов пропуска
    // \S - любой НЕ из вышеперечисленных символов
    // \w - любой алфавитно-цифровой символ
    // \W - любой НЕ алфавитно-цифровой символ
      // операции:
reg4 = /a{3}/;    // степень: a(3)=aaa
reg4 = /a{3,}/;   // степень "не менее": a(3,)=aaa|aaaa|aaaaa|...
reg4 = /a{3,5}/;  // степень " от ... до" a(3,6)=aaa|aaaa|aaaaa
```

Регулярные выражения широко используются для обработки строк в соответствии с шаблонами: для поиска подстроки в строке (функция *exec*), для проверки строки на соответствие регулярному выражению (функция *test*), для замены одной подстроки на другую (функция *replace*), для разбиения строки на подстроки в соответствии с выражением (функция *split*). Например, при выполнении следующей функции получим последовательный вывод строк *Piter* и 20:

```
function funExec(){
   b=" ****   Piter  ()%$     20 *** Kate 100";
   a1 = /\w+/;
   a2 = /\d+/;
   ttt= a1.exec(b);   alert(ttt);
   ttt= a2.exec(b);   alert(ttt);
}
```

Следующий пример показывает проверку строки на соответсвие шаблону. В качестве шаблона используется IP-адрес и E-mail:

```
function funTest(){
```

```
myRe=/\d+.\d+.\d+.\d/;
res =  myRe.test("212.192.2.20");      alert(res);  // верно
myRe=/\w+@(\w+.)*\w+/;
res =  myRe.test("ken@agtu.secna.ru"); alert(res);   // верно
res =  myRe.test("ken.agtu.secna.ru"); alert(res);   // ошибка
}
```

В качестве последнего примера, демонстрирующего применение регулярных выражений, покажем разделение строки на подстроки. Пусть исходной информациеЙ является строка, представляющая собой последовательность триад. Эта последовательность неупорядочена, а номера триад указаны в скобках независимо от того, операнд это или индекс триады. Разобьем каждую триаду на отдельные поля, удалим скобки из индекса триады и упорядочим триады по возрастанию этого индекса.

```
function funSplit()
{
data = new String ( "(3),=, (2)    ,d;(1),+,a    , b;(2),*, (1),  c");
document.write ("Заданная строка:<BR>"+ data + "<P>");
byComandList = new Array;

document.write ("После разделения на триады:<BR>");
pattern = /\s*;\s*/;      // точкой с запятой и возможными пробелами
patternCom = /\s*,\s*/;   // запятая
dataList = data.split (pattern);   // отдельные триады
patternNumber = /\d+/;             // шаблон целого числа
for ( i = 0; i < dataList.length; i++)
   {
   document.write (dataList[i] + "<BR>");
        // разделим на 4 поля:
   byComandList[i] = dataList[i].split (patternCom);
   if (byComandList[i].length != 4)
      alert("Число полей ("+byComandList[i].length+") в "+i);
   byComandList[i][0] = patternNumber.exec(byComandList[i][0]);
   }
byComandList.sort(); // Сортируем по номеру триады
document.write ("После сортировки:<BR>");
for ( i = 0; i < byComandList.length; i++)
   document.write (byComandList[i] + "<BR>")
}
```

Рассмотрим этот пример подробнее. Задана строка:

```
(3),=, (2) ,d;(1),+,a , b;(2),*, (1), c
```

Заданная строка разделяется на триады:

```
(3),=, (2) ,d
(1),+,a , b
(2),*, (1), c
```

В индексе триады удаляются все символы, кроме числа, затем результат разделения сортируется по номеру триады:

```
1,+,a,b
2,*,(1),c
3,=,(2),d
```

1.10 Макрогенерация как пример интерпретации

Многие системы программирования содержат препроцессор, сканирующий исходный код до начала работы компилятора. Препроцессор — это интерпретатор сканируемого текста, предоставляющий программистам большую мощность и гибкость в следующих случаях:

• Использование макроопределений уменьшают длину программного кода и делают его более простым и понятным. Некоторые макроопредеоения могут сократить число вызовов функций.

• Макроопределения позволяют включать текст из других файлов (например, файлы заголовков, содержащихся в стандартной библиотеке, прототипы пользовательских функций, константы).

• На основе операторов препроцессорного уровня реализуется условная компиляция, предназначенной для улучшения переносимости и отладки.

1.10.1 Понятие макрогенерации

Макропрограммирование — это гибкий механизм, позволяющий повысить наглядность программы и сократить длину кода. Использование макросредств часто позволяет сконструировать гибкий механизм, пригодный для создания кода, содержащего готовые настраиваемые элементы проектных решений. Макросредства языков программирования основаны на понятиях макроопределения, макровызова и макроподстановки.

• Макроопределение или макро — конструкция, которая позволяет обозначать одним идентификатором некоторый фрагмент кода.

• Макроопределение может иметь формальные параметры. Параметры могут быть ключевымим, которые указываются в качестве идентификатора макроопределения. Например, в C++ фрагмент программы

```
#define min(a, b)  (((a) < (b)) ? (a) : (b))
```

определяет код, который обеспечивает механизм для замены лексем с использованием формальных параметров, подобным параметрам функции.

Существует вариант использования позиционных формальных параметров, например, в командных файлах можно написать

```
copy %1 %2
```

• Макровызов — использование идентификатора макроопределения для указания той позиции в программе, в которой требуется выполнить макроподстановку, т.е. вставку текста макроопределения с заменой формальных параметров на фактические. Каждое появление идентификатора макро в исходном коде, следующее за этой строкой управления, будет замещаться последовательностью лексем (возможно пустой). Такое замещение называется макрорасширением. Как правило, после каждого макрорасширения производится сканирование для следующих расширений

макро. Это реализует вложенные макро: расширяемый текст может содержать идентификаторы макро, которые так же замещаются.

Вызов макро приводит к двум замещениям. Во-первых, идентификатор макро и аргументы в скобках замещаются последовательностью лексем. Затем все формальные аргументы, встретившиеся в последовательности лексем, заменяются соответствующими аргументами из списка действительных аргументов.

• Специальные операторы препроцессорного уровня (условные, циклические, операторы вставки файлов и т.д.) Например, можно написать

```
#ifndef    __TREE
#define    __TRREE
class TTree{
...
};
#endif
```

Анализ этих возможностей показывает, что к реализации макрогенерации можно подойти со следующих позиций:

• макрогенерация — это обработка текста программы с одновременным вычислением нового текста, что соответствует принципу интерпретации;

• для простоты отделения элементов программы, подлежащих обработке на препроцессорном уровне необходимо на каком–то из уровней языка (лексическом или синтаксическом) просто выделять препроцессорный уровень.

• Синтаксис языка макрогенератора не должен включать в себя синтаксис языка программирования целевого уровня.

Идея метода основана на том, что в тексте синтаксической структуры нужно обрабатывать только некоторые конструктивные элементы, считая остальной текст при этом только прозрачным обрамлением этой конструкции. Как правило, грамматика обрабатываемой части рассматривает анализируемый код как итерацию понятия "один элемент":

$$S- > SE|e$$

$$E- >< element_1 > | < element_2 > |...,$$

где S — весь текст программы, а E — обрабатываемый препроцессором фрагмент. Если выделение обрабатываемых и необрабатываемых фрагментов установлено не только на лексическом уровне, но и на синтаксическом уровне, то в этом случае полезным для реализации является лексема типа "нечто", которая выделяется сканером, если очередная лексема не принадлежит списку известных сканеру препроцессорного уровня лексем (некоторый аналог ошибочного символа, но и не совсем ошибка — просто непонятный препроцессору символ).

Итак, макроподстановка — это интерпретация текста программы. Если обрабатываемый текст плохо отделяется от прозрачного текста, то на лексическом уровне вносятся дополнения, позволяющие однозначно определить обрабатываемые элементы. Например, язык ассемблера имеет простую структуру (один оператор — одна строка), поэтому достаточно обычных ключевых слов, специальные символы не требуются. На языке Си препроцессорный уровень сложно отделить без синтаксического анализа в полном объеме, поэтому используется специальный знак # для выделения ключевых слов препроцессорного уровня.

1.10.2 Синтаксис и семантика языка макрогенератора

Построим синтаксис обрабатываемого языка так, чтобы не учитывались правила формирования операторов целевого языка. Синтаксис препроцессорного уровня в виде набора синтаксических диаграмм представлен на рисунке 1.16.

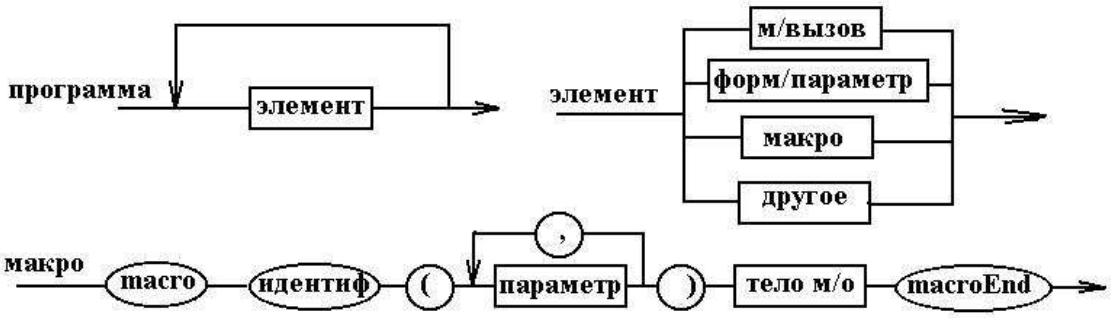


Рис. 1.16: Синтаксические диаграммы препроцессорного уровня

Работа макрогенератора заключается в том, что сканируемый текст превращается в другой текст. Рассмотрим необходимые семантические подпрограммы для реализации процесса интерпретации. Предварительно примем соглашение, что результирующий модуль будем формировать в новом файле, а семантическая таблица содержит информацию о макроопределниях, причем для каждого макроопределния в ней содержится количество параметров и ссылка на позицию в исходном модуле, начиная с которой находится тело макроопределения. Получаем прямой аналог семантического дерева для функций языка программирования. Ести язык не допускает использование вложенных макроопределений, то семантическая теблица превращается в список имен макроопрелеоений, правые ссылки которых указывают на собственные списки формальных параметров. Вложенные макроопределения, если таковые допускает язык программирования, находятся внутри тела макроопределения, и, следовательно, на этапе обработки определений верхнего уровня не обрабатываются, а, следовательно, и не заносятся в семантическое дерево. Тогда следующий набор семантическич подпрограмм полностью реализует процесс вычисления нового текста:

1 — перед началом работы выполняется настройка сканера и очистка библиотеки макроопрелелений,

2 — при сканировании началаа очередного макроопределения выполняется стандартная процедура формирования очередного фрагмента семантического дерева (имя определения и список его параметров),

3 — после заголовка макроопределения в дерево заносится текущее положения указателя текста и сбрасывается флаг интерпретации,

4 — при достижении конца макроопределения востанавливается флаг интерпретации,

5 — при сканировании каждого идентификатора проверяется его наличие в семантическом дереве: если идентификатор найден, то обнаруживается макровызов и начинается процедура интерпретации тела макроопределения,

6 — текст фактических параметров записывается в узлы формальных параметров в семантическом дереве; теперь текущий указатель дерева указывает на последний формальный параметр и их идентификаторы становятся видимы в семантическом дереве,

7 — интерпретация тела макроопределения — это обработка текста программы от начала соответствующего макроопределения до его конца (специального знака),

8 — если отсканированный идентификатор совпал с именем формального параметра (напомним, он теперь виден в семантическом дереве), выполныется копирование текста фактического параметра в генерируемый результирующий текст,

9 — если отсканировано ключевое слово, которое начинает оператор препроцессорного уровня, начинается интерпретация этого оператора по обычным правилам,

10 — во всех остальных случаях отсканированная лесема (идентификатор или "нечто") копируется в результирующий текст.

Кроме рассмотренных простейших операторов макроподстановки языки программирования допускают операторы управления вычислительным процессом. В частности классический метод записи заголовочного файла на языке C++ основан на использовании условного оператора препроцессорного уровня, например:

```
#ifndef __TREE
#define __TREE

#include "defs.hpp"

class Tree  {
private:
  Node * node;

  ...

};

#endif
```

Если язык макроуровня допускает условную или циклическую подстановку, то интерпретация выполняется по обычным правилам с учетом флага интерпретации.

1.11 Контрольные вопросы к разделу

1. Перечислите основные особенности реализации интерпретаторов.

2. Чем интерпретатор отличается от компилятора?

3. Чем двухпроходная интерпретация отличается от однопроходной?

4. В чем заключается задача создания семантической таблицы в процессе интерпретации?

5. Как интерпретируется оператор присваивания?

6. Как вычислить значение выражения в процессе интрепретации?

7. Чем определяется интерпретация условного оператора?

8. Как интерпретируется цикл?

9. Как присваивается значение элементу многомерного массива?

10. Перечислите особенности интерпретации процедур и функций.

11. Почему при интерпретации массивов можно контролировать выход индекса за границы, а в процессе трансляции — нельзя?

12. Какой способ передачи фактических параметров можно предложить при интерпретации вызова функции?

13. Как интерпретируются метки?

14. Чем интерпретации ссылки вперед отличается от интерпретации ссылки назад?

15. Какие действия должны выполняться в прологе блока?

16. Зачем нужен эпилог блока?

17. Если язык программирования не допускает рекурсивные вызовы процедур и функций, то как можно упростить передачу фактических параметров?

18. Какую роль выполняют семантические подпрограммы интерпретатора?

19. Зачем при интерпретации некоторых конструкций используется локальный флаг интерпретации?

20. Какие свойства языка программирования *JavaScript* определяются тем, что язык предназначен для интерпретации?

1.12 Тесты для самоконтроля к разделу

1. Зачем используется переменная $LocalFlInt$ в теле функции интерпретации условного оператора?

Варианты ответов:

а) Переменную $LocalFlInt$ можно не использовать, если в интерпретируемом языке программирования нет рекурсивных вызовов функций.

б) Переменную $LocalFlInt$ нужно использовать только при интерпретации циклов, и при интерпретации условий она не нужна.

в) Переменная $LocalFlInt$ — глобальная переменная для сохранения текущего значения FlInt.

г) В результате вычисления выражения изменяется значение FlInt, следовательно, нужно иметь возможность восстановления исходного значения флага интерпретации при завершении условного оператора.

д) В результате рекурсивной вложенности операторов может измениться значение FlInt, следовательно, нужно иметь возможность восстановления исходного значения флага интерпретации при завершении условного оператора.

Правильный ответ: д.

2. Чем однопроходной интерпретатор отличается от многопроходного?

Варианты ответов:

а) многопроходной интерпретатор выполняет синтаксический анализ исходного модуля в процессе многократного сканирования этого модуля; однопроходной интерпретатор сканирует текст только один раз;

б) однопроходной интерпретатор выполняет синтаксический анализ и выполнение программы для каждого единичного оператора в отдельности, а многопроходной интерпретатор делает это для многих операторов сразу;

в) многопроходной интерпретатор в процессе синтаксического анализа генерирует некоторый внутренний код, который затем интерпретируется;

г) многопроходной интерпретатор интерпретерует все операторы сразу, а однопроходный — только по нажатию клавиши выполняет один отдельно взятый оператор;

д) многопроходных интерпретаторв не существует.

Правильный ответ: в.

3. Чем семантические подпрограммы интерпретации отличаются от семантических подпрограмм компиляции?

Варианты ответов:

а) семантические подпрограммы интерпретатора не делают ничего дополнительного по сравнению с семантическими подпрограммами компилятора;

б) использование семантических подпрограмм интерпретатора позволяет выполнять вычисления и сохранять значения данных в памяти;

в) использование семантических подпрограмм интерпретатора позволяет выполнять вычисления;

г) семантические подпрограммы интерпретатора позволяют резервировать память для данных;

д) в процессе интерпретации семантические подпрограммы предназначены для изменения значения флага интерпретации.

Правильный ответ: б.

4. Какие из следующих утверждений истинны?

1) Однопроходной интерпретатор может оптимизировать код.

2) Если язык программирования поддерживает только операторы присваивания и оператор типа оператор $do - - - while$, то можно не использовать флаг интерпретации.

3) Многопроходная схема интерпретации характеризуется более быстрыми алгоритмами выполнения вычислений по сравнению с однопроходной;

4) Если язык программирования поддерживает только операторы присваивания и оператор типа оператор $while$, то можно не использовать флаг интерпретации.

Варианты ответов:

а) все утверждения ложны;

б) 2 и 4;

в) 3 и 4;

г) 1, 3 и 4;

д) 2 и 3;

е) 2, 3 и 4;

ж) 1, 2 и 3;

з) все утверждения истинны.

Правильный ответ: д.

5. Поясните назначение $DataAsInt$, $DataAsFloat$, $DataAsBool$.

Варианты ответов:

а) Это поля структуры, в которой хранятся значения вычисленных данных в зависимости от типа зачения.

б) Это типы данных.

в) Это локальные переменные, который последовательно вычисляются в процессе интерпретации выражений.

г) Это глобальные переменные, который последовательно вычисляются в процессе интерпретации выражений.

д) Это данные для организации вызова функции и передачи в нее фактических параметров.

Правильный ответ: а.

1.13 Упражнения к разделу

1.13.1 Задание

Цель данного задания – построить интерпретатор язака программирвония. Работу над заданием следует организовать, последовательно выполняя следующие операции.

1. Определите типы данных, которые должен вычислять Ваш интерпретатор в процессе интерпретации исходного модуля. Введите определение типа значения данных.

2. Модернизируте таблицы с тем, чтобы они поддерживали хранение значений данных, вычисляемых в процессе интерпретации. На структуру таблиц существенное влияние должны оказать:

• наличие массивов;

• наличие параметров у процедур и функций;

• возможность рекурсивного вызова процедур и функций, если это предусмотрено заданием.

3. Модернизируте программы приведения типов с тем, чтобы наряду с вычислением типа они вычисляли бы и соответствующее значение.

4. Определите дополнительные параметры процедур, соответствующие синтаксическим диаграммам выражений. Эти параметры должны обеспечивать передачу вычисленных значений выражений.

5. Постройте простейший отладчик интерпретатора. Для этого достаточно

• определить заголовок функции, предназначенной для выдачи вычисленного значения переменной, элемента массива, элемента записи (в соответствии с типами данных Вашего задания);

• написать подпрограмму выдачи отладочной информации: имени изменяемого данного, вычисленного значения.

• поставить вызов этой функции в реализации оператора присваивания.

6. Встройте в программу флаги интерпретации:

• глобальный флаг интерпретации для всей программы;

• локальные флаги (для хранения текущего значения) во всех структурных операторах, в процессе интерпретации которых может изменяться значение глобального флага.

7. Вставьте в синтаксические диаграммы операции работы с флагами интепретации и с указателем исходного модуля.

8. Внесите соответствующие изменения в программу синтаксического анализатора, построенного методом рекурсивного спуска.

9. Отладьте программу. При отладке особое внимание обратите на преобразование данных в соответствии с таблицей приведений.

1.13.2 Пример выполнения задания

Рассмотрим интерпретатцию языка программирования — простейшего варианта языка *JavaScript*, анализ которого мы выполняли в первой части нашего учебного пособия.

Определим типы данных, которые должен вычислять интерпретатор: пусть это будут данные целые, вещественные, символьные и строковые. В выражениях разре-

шим использование элементов массивов и вызовы функций. Уточним таблицу приведения типов. Пусть над строками (как массивами символов) допускаются только операции выборки элемента по индексу, а таблица приведения для любой операции, кроме операции присваивания, имеет следующий вид:

	int	float	char
int	int	float	int
float	float	float	float
char	int	float	char

Тогда описание типов имеет вид

```
enum TObjectType {
                  ObjConst=1,     // константа
                  ObjVar,         // простая переменная
                  ObjArray,       // массив
                  ObjTypeArray,   // тип массива
                  ObjStruct,      // структура
                  ObjTypeStruct,  // тип  структуры
                  ObjFunct        // функция
                  };

enum TDataType   {
                  DataInt=1,      // int
                  DataFloat,      // float
                  DataChar,       // char
                  };
union TDataValue                  // значение одного элемента данных
   {
   int DataAsInt;                 // целое значение
   int * ArrayDataAsInt;          // массив целых значений
   float DataAsFloat;             // вещественное значение
   float *ArrayDataAsFloat;       // массив вещественных значений
   char DataAsChar;               // символьное значение
   char * ArrayDataAsString;      // строковое значение

   };

struct TData     // тип и значение одного элемента  данных
   {
   TDataType   DataType;          // тип
   TDataValue  DataValue;         // значение
   };
```

Каждая функция синтаксического анализатора, соотвествующая какому–либо выражению, при реализации интерпретатора будет возвращать вычисленное значение в структуре типа $TData$. Например, для понятия "элементарное выражение" программа имеет вид:

```
void E(TData * res)
//****************************************
```

```
// элементарное выражение
// E -> c1 | c2 | c3 | c4 | N | H | (V)
//          N - элемент массива
//          H - вызов функции
//****************************************
{
TypeLex l; int t,uk1;
uk1 = GetUK(); t = Scaner(l);
if (t == TConsChar)
     {
     res -> DataType = DataChar;  // тип
     res -> DataValue.DataAsChar = l[0];  // изображение символа
     }
else
if (t == TConsInt)
     {
     res -> DataType = DataInt;  // тип
     res -> DataValue.DataAsInt = atoi(l); // целое число
     }
else
if ( t== TConsFloat)
     {
     res -> DataType = DataFloat;  // тип
     res -> DataValue.DataAsFloat = atof(l);  // вещественное число
     }
else
if (t == TConsExp)
     {
     res -> DataType = DataFloat;  // тип
     res -> DataValue.DataAsFloat = atof(l);  // вещественное число
     }
else
if (t == TLS)
    {
    V(res);  // вернется значение выражения в скобках
    t = Scaner(l);
    if (t != TPS) PrintError("ожидался символ )",l);
    }
else
if (t == TIdent)  // для определения N или  H нужно иметь first2
  {
  t = Scaner(l);  PutUK(uk1);
  if (t == TLS) H(res);       // вернется значение функции
         else   N(res);       // вернется значение элемента массива
                              // или простой переменной
  }
else PrintError("Неверное выражение ",l);
}
```

Строго говоря, вычисление элементарного выражения должно проводиться толь-

ко при установленном флаге интерпретации $FlInt$, например:

```
if (t==TConsInt)
     {
     if ( FlInt == 0 ) return;
     res -> DataType = DataInt;  // тип
     res -> DataValue.DataASInt = atoi(l);
     }
```

Однако в примере при вычислении выражения мы не проверяли флаг интерпретации. Дело в том, что мы вычисляли только значение константы, не выполняя никаких операций над данными. Значение константы всегда существует, поэтому никаких исключений при вычислении ее внутреннего представления не возникнет – это безопасный код. Совершенно иная ситуация, например, при вычислении элемента массива: индексное выражение над неопределенными данными может привести к выходу за границы массива — код становится опасным. Поэтому при $FlInt == 0$ не следует выполнять выборку из массива, а при $FlInt! = 0$ следут обязательно проверить выход индекса за границы. Аналогичная ситауция,например, при вычиполнении операций над неопределенными данными: можно разделить на ноль, получить переполнение и т.п. Поэтому при $FlInt == 0$ нельзя выполнять никакие операции над данными.

Глава 2

СИНТАКСИЧЕСКИ УПРАВЛЯЕМЫЙ ПЕРЕВОД

Транслятор выполняет синтаксический и семантический анализ всей входной цепочки, а затем начинает генерацию кода. Как правило, генерация кода происходит поэтапно с использованием вспомогательного промежуточнего кода программы. Транслятор выполняет генерацию кода на основе законченных синтаксических конструкций исходного модуля. В качестве таких конструкций естественно использовать конструкции, которые в совокупности имеют единый смысловой оттенок: это отдельные простые операторы, циклы, ветвления, функции и т.п. Одни и те же конструкции имеют место в разных языках программирования. Именно поэтому стал воможен тот подход к трансляции на общий промежуточный язык, который используется в среде Microsoft .NET Framework.

Вопрос выбора языка программирования для реализации проектов — задача очень непростая, ибо у разных языков разные возможности. Например, в неуправляемом $C/C++$ программист имеет доступ к системе на довольно, низком уровне и может распоряжаться памятью по своему усмотрению. Использование Visual Basic 6 позволяет быстро строить пользовательский интерфейс и легко управлять COM–объектами, а $C\sharp$ предназначен для эффективной разработки приложений на платформе $.NET$ для Web–сервисов, служб и т.п. При генерации кода .NET Framework позволяет разным языкам интегрироваться, поскольку компиляторы генерируют код на общем промежуточном языке CLI (Common Intermediate Language), а не традиционный объектный код, состоящий из команд процессора. Интеграция разных языков программирования означает, что на одном языке программирования можно использовать типы, созданные на других языках. Общеязыковая спецификация CLS (Common Language Specification) определяет правила, которым должны следовать разработчики компиляторов, чтобы их языки интегрировались с другими. Все эту упрощает повторное использование кода, создается общий набор готовых компонентных типов.

Вернемся к вопросу о генерации кода. Итак, транслятор в итоге должен получить или ассемблерный код ($*.asm$), или объектный код ($*.obj$), содержащий шестнадцатиричный код команд процессора, или машинно–ориентированный код на некотором промежуточном языке. Получить сразу из исходного текста программы машинно–ориентированный код очень сложно. Поэтому применяются различные формы некоторого промежуточного кода, который, во-первых, удобен для перевода из исходного текста, а, во-вторых, легко преобразуется в машинно–ориентированный код. Схема перевода в этом случае имеет вид, представленный на рисунке 2.1.

Конечно, те или иные фазы работы транслятора могут либо отсутствовать совсем, либо объединяться. В простейшем случае однопроходного транслятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

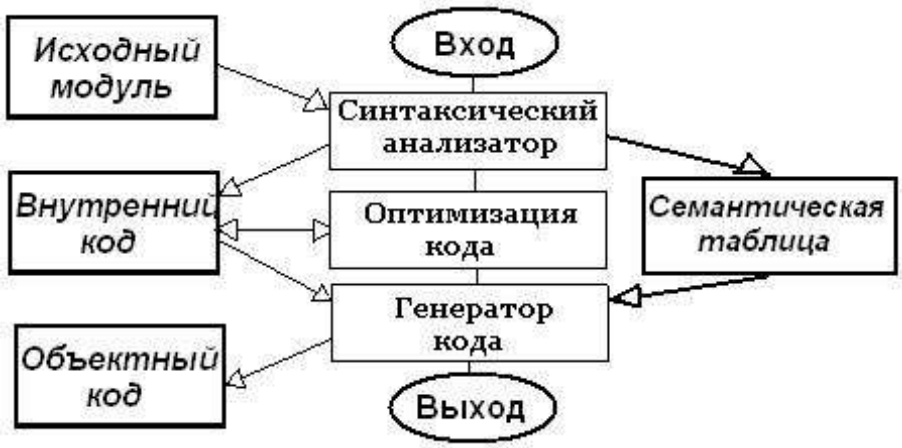


Рис. 2.1: Схема трансляции

2.1 Формы представления промежуточнего кода

В процессе трансляции программы часто используют промежуточное представление программы, предназначенное прежде всего для удобства генерации кода и проведения различных оптимизаций программы. Сама форма промежуточного (или внутреннего) кода зависит от целей его использования. промежуточний код может иметь различную структуру в зависимости от выбранной разработчиком реализации компилятора, может зависеть от архитектуры вычислительной системы или структуры транслируемого языка, от качества получаемого объектного кода и т.п. Рассмотрим сначала цель перевода во внутреннюю форму. Генератор кода выполняет преобразование синтаксического дерева и семантической информации, содержащейся в таблице идентификаторов, в машинозависимый код. Таким образом, если синтаксическое дерево и семантическая таблица являются функциями $\tau_1(x)$ и $\tau_2(x)$ от исходного кода x, то объектный код — функция $y_{ok} = \tau_3(\tau_1(x), \tau_2(x))$. Такая реализация по законам декомпозиции, безусловно, проще, чем реализации непосредственного перевода исходного модуля в объектный код без использования промежуточного кода $y_{ok} = \tau_4(x)$.

Результатом синтаксического анализа является дерево грамматического разбора, поэтому можно сказать, что это дерево является простейшей промежуточной формой. Однако, чтобы упростить алгоритм преобразования в результирующую программу, рассматривают такую структуру промежуточного кода, которая легко отображается в результирующий код (как правило, машиннозависимый). Любой способ представления промежуточнего кода в той или иной степени должен отображать древовидную структуру дерева синтаксического анализа. В то же время, не все элементы синтаксического дерева должны в обязательном порядке присутствовать во промежуточней форме. Например, для выражений нет необходимости представлять всю иерархию нетерминалов, а скобки вообще являются лишними после того, когда

синтаксическое дерево уже построено и порядок вычисления выражения уже определен. Таким образом, задача выбора архитектуры промежуточнего кода — это задача выбора такого представления, которая удовлетворяет следующим свойствам:

• во промежуточнем коде присутствуют все операнды и все операции исходного модуля,

• во промежуточнем коде отображается структура синтаксического дерева,

• некоторые подробности в структуре синтаксического дерева могут быть опущены, если это не приводит к неоднозначности в понимании структуры программы.

промежуточний код можно считать некоторой промежуточной позицией на пути от синтаксического дерева к ассемблерной программе. промежуточний код может быть более или менее приближен к машинным командам. Например, разработанный компанией Microsoft промежуточный язык MSIL (Microsoft Intermediate Language) для платформы .NET напоминает машинные команды, но не зависит ни от какой конкретной архитектуры процессора. Промежуточный язык такого типа — это не только техническое средство для построения трансляторов, но и средство для реализации платформенно независимых приложений. Другой эффект применения промежуточного кода состоит в кросс–языковой совместимости, начиная от уровня структуры программы, представленной на промежуточном языке. Например, как уже отмечалось выше, на платформе .NET могут одновременно работать модули, написанные на любых .NET–совместимых языках. Можно даже написать на одном языке класс, который будет наследовать классу, созданному на другом языке.

В зависимости от назначения промежуточного кода он может содержать или не содержать данные из семантической таблицы. В последнем случае такой язык является только средством представления синтаксической структуры и не может использоваться для реализации кросс–платформенной совместимости. Более того, компилятор может использовать промежуточные коды разного уровня для поэтапного перехода от одной стадии трансляции к другой: на различных фазах компиляции различные формы представления кода по мере перехода от одной фазы компиляции к другой последовательно преобразуются одна в другую.

Рассмотрим промежуточний код, предназначенный исключительно для целей трансляции в машинозависимый код. Как правило, рассматривают следуюшие способы представления внутреннего кода вышеуказанного уровня:

• древовидная структура,

• постфиксная или префиксная запись,

• тетрады — двухадресный код с явно именуемым результатом,

• триады — двухадресный код с неявно именуемым результатом.

Некоторые компиляторы с простых языков, не оптимизирующие генерируемый код, генерируют код по мере разбора исходной программы без использования промежуточнего кода.

2.1.1 Деревья

Дерево синтаксического анализа — это структура, отражающая конструкцию транслируемого кода в терминах правил КС–грамматики. Известно, что синтаксическое дерево отражает синтаксис конструкций входного языка и явно содержат в себе полную взаимосвязь конструкций. Рассмотрим такой способ представления промежуточнего кода, построенного на основе синтаксических деревьев, при котором сохранилась бы взаимосвязь конструкций, но упростилось бы представление дерева в целом.

Для этой цели рассмотрим сначала синтаксические деревья для выражений и операторов присваивания. После того, как синтаксическое дерево уже построено, выполним одну простую операцию преобразования дерева: поднимем каждый знак операции в дереве вверх до тех пор, пока этот знак не станет вершиной, соединяющей некоторые операнды. После этого удалим все промежуточные вершины дерева, которые имеют ровно одного потомка. Анализ полученного дерева показывает, что некоторые вершины в таком дереве являются лишними — это вершины, соответствующие скобкам. Тот факт, что скобки после построения синтаксического дерева стали лишними, вообще говоря очевиден. Дело в том, что скобки в исходном выражении использовались только для того, чтобы определить порядок выполнения операций, а стало быть, и структуру синтаксического дерева. Выполним операцию удаления поддеревьев, соответствующих скобкам. В результате получим дерево, в котором значимыми являются все конструкции. Пример данного преобразования представлен на рисунке 2.2. Полученное дерево иногда называют деревом операций, поскольку все вершины, кроме листьев, являюся операциями.

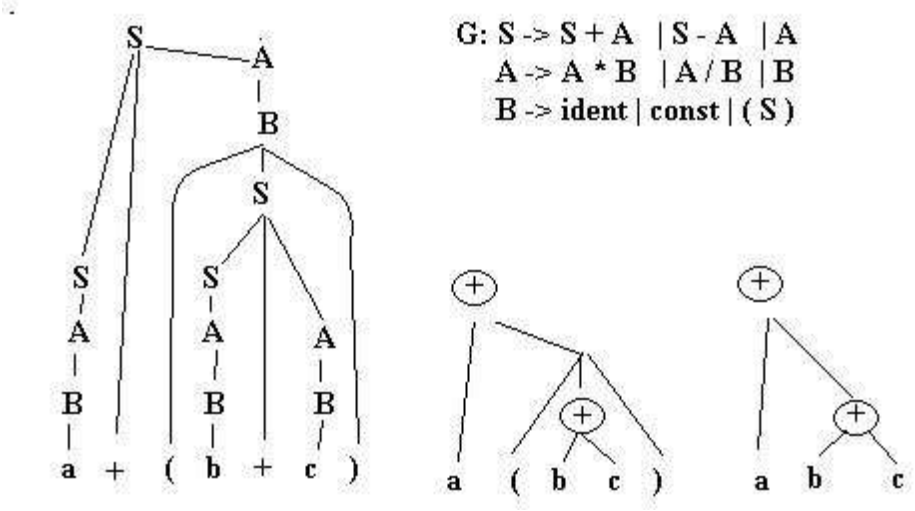


Рис. 2.2: Процесс построения промежуточнего кода в виде древовидной структуры

Метод можно распространить на все операторы. Для операторов управления ходом вычислительного процесса в качестве операций можно выбрать ключевые слова, соответствующие выполняемым действиям, а далее конструктивные элементы оператора становятся операндами операции. Например, для оператора if промежуточными вершинами–операциями являются операции if и $else$, а для оператора $while$ достаточно одной операции.

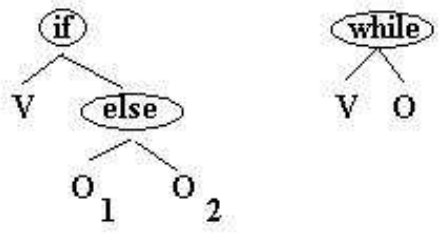


Рис. 2.3: Древовидные структуры промежуточнего кода для операторов if и $while$

Дерево операций удобно использовать при интерпретации или на этапе подготовки к генерации кода. Однако использование древовидной структуры требует наклад-

ных расходов на реализацию связей. Кроме того, многие оптимизирующие алгоритмы требуют коренной перестройки дерева вплоть до превращения дерева в сложную связную структуру. Поэтому деревья не могут использоваться при оптимизации кода, в результате которой древовидная структура может разрушиться.

2.1.2 Префиксная и постфиксная запись

Польская инверсная запись — ПОЛИЗ — была предложена польским математиком Лукашевичем для записи выражений. Это постфиксная запись операций, т.е. каждой операции предшствуют ее операнды. Например, выражение $a+b*c$ в ПОЛИЗ имеет вид

$$abc*+.$$

Форма постфиксной записи получена из дерева операций методом его укладки в линию, если на это дерево давить слева направо. Пример такой укладки представлен на рисунке 2.4.

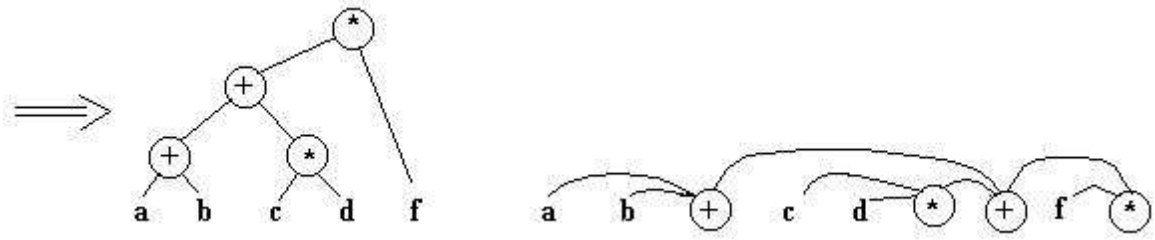


Рис. 2.4: Постфиксная запись выражения

В ПОЛИЗ все операции выполняются в том порядке, в котором они встречаются в выражении. Это условие позволяет не использовать приоритеты операций. Более того, ПОЛИЗ исключительно удобно использовать при наличии стека: например, при вычислении выражения достаточно каждый операнд записывать в стек, а при появлении в ПОЛИЗ операции выполнять ее над вершиной стека, заменяя операнды вычисленным результатом операции. Для генерации машинного кода алгоритм обработки ПОЛИЗ аналогичен алгоритму вычисления. Простейший вариант реализуется при генерации команд типа "регистр — регистр". Для этого достаточно следующих действий:

• для операнда генерировать команду загрузки этого операнда в свободный регистр, помещая в рабочий стек имя этого регистра,

• для операции генерировать соответствующую команду для регистров, находящихся в верхушке рабочего стека, замещая при этом верхушку стека на адрес регистра, в котором находится результат операции. Например, при трансляции выражения $abc+-$ генерация команд выполняется следующим образом:

вход	a	b	c	+	-
рабочий стек	ax	bx ax	cx bx ax	bx ax	ax
выход	mov ax,a	mov bx,b	mov cx,c	add bx,cx	sub ax,bx

Предполагается наличие логической шкалы занятости для всех регистров, а также программы выделения свободного регистра. В зависимости от запроса эта программа либо выделяет первый свободный регистр, или освобождает специально запрашиваемый. При возникновении нехватки регистров формируются команды временного хранения содержимого освобождаемого регистра. Более подробно данная проблема будет рассмотрена позже.

Наряду с постфиксной записью выражений существует префиксная запись, в которой операции предшествуют операндам. Например, выражение $a + b * c$ в префиксной записи имеет вид

$$+a * bc.$$

Префиксная запись выражения получается при укладке дерева операций в линию при давлении на него справа налево — пример представлен на рисунке 2.5.

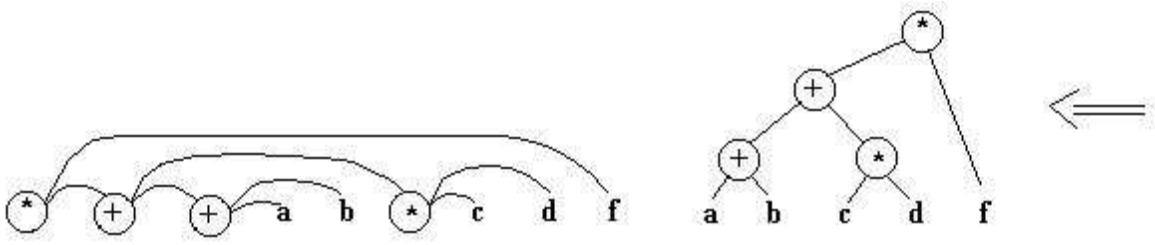


Рис. 2.5: Префиксная запись выражения

Безусловно, ПОЛИЗ удобнее префиксной записи, т.к. алгоритм преобразования ПОЛИЗ в машинный код существенно проще.

По аналогии с выражениями ПОЛИЗ используется для представления других операторов, например можно предложить следующую запись:

- $V\ if\ O_1\ else\ O_2\ then$ — для условных операторов в полной форме,
- $V\ if\ O_1\ then$ — для условных операторов в усеченной форме,
- begin V while O repeat — для оператора while V O.

Следует отметить еще один факт, который непосредственно следует из способа получения ПОЛИЗ: поскольку ПОЛИЗ представляет собой уложенное в линию дерево, то и любые алгоритмы обработки ПОЛИЗ фактически представляют собой обработку дерева. Так, при обработке операторнда надо будет сопоставить его с соответствующей операцией. Это означает, что операнд должен ждать появления относящейся к нему операции. А поскольку операнды в ПОЛИЗ упорядочены в том виде, как они упорядочены в дереве, необходима стековая структура для операндов. Отсюда алгоритм:

- операнды помещаются в стек в соответствии с порядком их появления в ПОЛИЗ,
- операция извлекает из верхущки стека необходимое количество операндов.

2.1.3 Триады и тетрады

Тетрады — это конструкции, состоящие из четырех полей:

- операция,
- два операнда,
- результат выполнения опеарции.

В силу вышеуказанного назначения полей тетрады иначе называют или трехадресным промежуточним кодом, или двухадресным кодом с явно именуемым результатом. Идея построения такого кода основана на том, что транслируемые составные выражения могут быть разбиты на подвыражения, при этом могут появиться временные имена, обозначающие местонахождение результата. Смысл термина "трехадресный код" в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код — это линеаризованное представление синтаксического дерева, в котором явные имена соответствуют промежуточним вершинам дерева синтаксического анализа. Например, выражение $a - b * c$ может быть оттранслировано в последовательность операторов

$$res_1 = b * c;\ res_2 = a - res_1;$$

где res_1 и res_2 — имена, сгенерированные компилятором. В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

```
if (a>b) S1 else S2
```

может быть представлен следующим кодом:

```
-      a      b       res_1
if   res_1  label_S2
label_S1:
   <код для S1>
go   Label_S3
label_S2:
   <код для S2>
label_S3:
```

Здесь if — двухместная операция условного перехода, не вырабатывающая результата. Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код. Например, можно оптимизировать запись выражения $b/(2-c)+b/(2-c)$:

```
-   2    c    res1          -   2      c   res1
/   b   res1  res2          /   b    res1  res2
-   2    c    res3          +  res2  res2  res5
/   b   res3  res4          :=  a    res5
+  res2 res4  res5
:=  a   res5
```

Попробуем преобразовать тетрады в более эффективный код. Заметим, что тетрады занимают в памяти массив, поэтому с каждой тетрадой можно связать ее индекс. Тогда вместо имени результата можно использовать индекс тетрады и последнее поле становится лишним. тетрада превращается в триаду. Остается только ввести различие между операндами — индексами и непосредственными операндами. В программе компилятора это различие будет представлено флагом, связанным с операндом, а

при визуальном представлении ссылку (индекс триады) будем указывать в скобках. Таким образом, триада — это структура из трех полей:

• операция,

• два операнда, каждый из которых может быть либо непосредственным операндом, либо ссылкой на другую триаду.

Для представления структур управления в виде последовательности триад необходимо ввести операции передачи управления, например,

```
if  (i)  (j)
go  (k)   -
```

Здесь *go* — операция безусловной передачи управления на триаду, указанную в качестве операнда, а if — передача управления на триаду, указанную в качестве первого или второго операнда, если предварительно был выработан признак результата 1 или 0 соответственно. Тогда, например, оператору

```
if (V)   O1 else O2
```

может соответствовать его следующее представление в виде последовательности триад:

```
...     <триады для V>
i)   if   (i+1)  (m+1)
i+1) <триады для O1>
...
m)     go    (k+1)
m+1)   <триады для O>
...
k)
```

В процессе формирования триад для структур управления, как правило, появляются недоформированные триады, которые формируются полностью на дальнейших шагах построения внутреннего кода.

2.2 Определение синтаксически управляемого перевода

Прежде, чем давать формальное определение синтаксически управляемого перевода, необходимо определить, что именно мы хотим формализовать.

Во-первых, еще раз отметим, что в результате выполнения синтаксического анализа строится дерево грамматического разбора. Генерацию кода можно считать функцией, определенной на синтаксическом дереве и на информации, содержащейся в семантическом дереве. Для того, чтобы формально описать правила, по которым для синтаксических конструкций исходного языка компилятор должен построить результирующий код, часто используется метод, называемый *синтаксически управляемым переводом*. Целью синтаксически управляемого перевода является совместная запись синтаксических правил грамматики и семантических правил в такой форме, которая пригодна для реализации интерпретации или трансляции в процессе грамматического разбора. Синтаксически управляемый перевод — это механизм, позволяющий связывать с каждой цепочкой входного языка другую цепочку, которая должна быть выходом (результатом перевода) исходной цепочки.

Чтобы яснее стала задача формализации семантики, вспомним метод рекурсивного спуска и рассмотрим синтаксическую диаграмму, на основе которой построена интерпретация какого–либо фрагмента исходной программы. Пусть это будет оператор присваивания, представленный на рисунке 2.6 вверху. Обозначим семантические подпрограммы символами Δ_1 и Δ_2 (Δ_1 – для выполнения действий по поиску типа и адреса переменной по ее идентификатору, Δ_2 – для приведения типа вычисленного выражения к типу переменной и записи вычисленного выражения по найденному адресу). Поскольку семантические подпрограммы явно вызываются в программе, то рисунок можно изменить, вставив соответствующие блоки в синтаксическую диаграмму. Получим диаграмму, представленную на том же рисунке внизу. А теперь вернемся от синтаксической диаграммы к правилу КС–грамматики, считая, что блок семантической подпрограммы — это некоторый специальный *операционный символ* грамматики. В результате правило грамматики имеет вид:

$$A \to a\Delta_1 = V\Delta_2;$$

Рассмотрим подробнее получившееся правило. Во первых, это правило описывает семантику интерпретации оператора присваивания. Во–вторых, оно имеет смысл не само по себе, а связано с синтаксическим правилом КС–грамматики $A \to a = V;$. И, наконец, в правой части этого правила стоят как символы из множества $V_T \cup V_N$, так и специальные семантические символы, которые обозначают вызов семантических подпрограмм (Δ_1 и Δ_2). Более того, оказывается, что при такой записи семантики мы формально указали не все семантические вычисления. Дело в том, что при обработке нетерминала V вычисляется значение и тип семантического выражения, которые где-то должны быть сохранены. Фактически, мы можем сказать, что с каждым нетерминалом N может быть связана некоторая семантическая функция $\tau(N)$ перевода это нетерминала. Семантические функции перевода могут быть связаны и с терминальными символами, поэтому в общем виде семантическое правило для оператора прсиваивания может быть выражено формулой

$$\tau(A) \;=\; \tau(a)\ \Delta_1\ \tau(=)\ \tau(V)\ \Delta_2\ \tau(;)$$

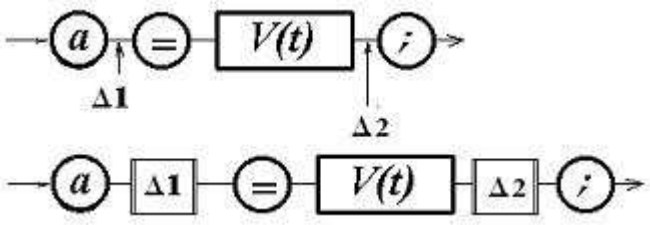


Рис. 2.6: Семантика оператора присваивания

Итак, мы пришли к пониманию того факта, что с каждым терминальным или нетерминальным символом может быть связана некоторая функция, которая представляет смысл соответствующей конструкции. В принципе, таких функций у каждого символа может быть несколько. Например, выражению можно приписать три функции:

- перевод выражения в последовательность выполняемых операций,
- тип выражения,
- местонахождение результата выражения.

Однако, как определение этих функций, так и правила их вычисления существенно зависят от способа представления промежуточнего кода программы.

Перейдем теперь к формальному определению. Существует много вариантов определения синтаксически управляемого перевода. Все они построены на основе того, что перевод должен быть задан функцией $y = \tau(x)$, которая определяет правила формирования результирующего кода y по исходному коду программы x. Поскольку внутреннее представление исходного модуля строится в процессе синтаксического анализа, необходимо с каждым правилом грамматики связать правила перевода. Правило КС–грамматики определяется для некоторого нетерминала, поэтому можно говорить о сопоставлении функции каждому нетерминальному символу грамматики.

Для того, чтобы построить перевод для программы x в целом, надо найти перевод для корня дерева грамматического разбора цепочки x. Поэтому для каждой вершины синтаксического дерева надо так выбирать результат перевода, чтобы можно было построить результирующий код для корня дерева. Пусть $A \to u$ – правило КС–грамматики. В дереве грамматического разбора сентенциальная форма u состоит из терминальных и нетерминальных символов $u = a_1 a_2 ... a_k$. Результат перевода для вершины A строится в результате конкатенации в некотором порядке результатов перевода ее прямых потомков a_1, a_2, ... a_k. В свою очередь для построения перевода для прямых потомков вершины A потребуется найти перевод для потомков второго уровня и т.д.

Пусть нам требуется перевести в ПОЛИЗ выражение, которое содержит идентификаторы и константы, знаки операций $+$, $-$, $*$, $/$, круглые скобки:

$$\begin{array}{ll} G: & V \to V + A \mid V - A \mid A \\ & A \to A * E \mid A/E \mid E \\ & E \to a \mid c \mid (V). \end{array}$$

Если, например, вершина синтаксического дерева соответствует правилу грамматики $V \to V + A$, то при условии, что вычислен перевод $\tau(V)$ и $\tau(A)$ соответственно для нетерминалов V и A, перевод для корня дерева можно построить в результате конкатенации $\tau(V)\tau(A)+$. Чтобы различать одинаковые нетерминалы в записи правила грамматики, будем ставить индекс — номер повторяющегося нетерминала в записи слева направо. Тогда правило перевода имеет вид

$$\tau(V_1) = \tau(V_2)\tau(A) + .$$

Перевод в ПОЛИЗ операнда равен самому этому операнду. Таким образом, например, для правила $E \to a$ справедливо

$$\tau(E) = a.$$

Правило $A \to E$ или $V \to A$ передают перевод от потомка к родителю, таким образом, справедливы правила перевода

$$\tau(V) = \tau(A), \tau(A) = \tau(E).$$

Схему перевода можно представить в следующей таблице:

правило грамматики	перевод	сокращенная запись перевода
$V \to V + A$	$\tau(V_1) = \tau(V_2)\tau(A)+$	$V = VA+$
$V \to V - A$	$\tau(V_1) = \tau(V_2)\tau(A)-$	$V = VA-$
$V \to A$	$\tau(V) = \tau(A)$	$V = A$
$A \to A * E$	$\tau(A_1) = \tau(A_2)\tau(E)*$	$A = AE*$
$A \to A/E$	$\tau(A_1) = \tau(A_2)\tau(E)/$	$A = AE/$
$A \to E$	$\tau(A) = \tau(E)$	$A = E$
$E \to a$	$\tau(E) = a$	$E = a$
$E \to c$	$\tau(E) = c$	$E = c$
$E \to (V)$	$\tau(E) = \tau(V)$	$E = V$

Рассмотрим простейшее определение, подсказанное рассмотренным примером.

Определение 2.1. Схемой синтаксически управляемого перевода (или схемой синтаксически ориентированного перевода, или сокращенно СУ–схемой) для КС–грамматики $G = (V_N, V_T, P, S)$ называется пятерка

$$T = (V_N, V_T, V_{res}, R, S),$$

где V_N – конечное множество нетерминальных символов;

V_T – конечный входной алфавит;

V_{res} – конечный выходной алфавит;

$S \in V_N$ – аксиома грамматики,

$R = \{A \to \alpha, \beta\}$ – конечное множество правил перевода, где $\alpha = v_1 v_2 ... v_m, A \to \alpha \in P$, удовлетворяющих следующим условиям:

• каждый символ, входящий в β, принадлежит $V_{res} \cup V_N$, либо является символом из V_N с индексом;

• каждый символ из β, представляющий собой символ из $V_{res} \cup V_N$ или из V_N с индексом, соответствует некоторому нетерминалу из $v_1 v_2 ... v_m$;

• если символ $B \in V_N$ имеет более одного вхождения в α, то каждый символ B_k в цепочке β соотнесен нижним индексом с конкретным вхождением B в цепочку α.

Правило $A \to \alpha$ называют входным правилом вывода, а β — переводом нетерминала A. Если в СУ-схеме T нет двух правил перевода с одинаковым входным правилом вывода, то ее называют *семантически однозначной.*

Практическое применение определения 2.1 иногда может вызывать затруднения в построении простой и прозрачной конструкции схемы перевода. Дело в том, что фактически эта схема определяет только одну функцию, связанную с каждой вершиной дерева синтаксического анализа. Чаще бывает проще определить перевод в терминах нескольких функций, например, для выражений имеет смысл говорить о промежуточнем коде в виде последовательности триад, о семантическом типе выражения, об обозначении результата вычисления выражения и т.п. Например, рассматривая в начале главы пример интерпретации оператора присваивания, мы ввели два параметра для выражения — тип и значение. Кроме того, опыт показывает, что при описании семантики удобно работать с семантическими подпрограммами. Определение 2.1 схемы СУ–перевода удобно несколько модифицировть, введя понятие *операционных символов.* Это специальные символы для обозначения вызовов семантических подпрограмм, которые необходимо выполнить в процессе трансляции. Поэтому на практике чаще рассматривают следующую обобщенную схему синтаксически управляемого перевода.

Определение 2.2. Схемой обобщенного синтаксически управляемого перевода для КС–грамматики $G = (V_N, V_T, P, S)$ называется шестерка

$$T = (V_N, V_T, V_{res}, \Delta, R, S),$$

где V_N, V_T, V_{res}, S имеют тот же смысл, что и в определении 2.1, $\Delta = \{\Delta_1, \Delta_2, ..., \Delta_n\}$ — конечное множество операционных символов (конктретизация понятия символов перевода из 2.1), а R – конечное множество правил перевода вида

$$A \to \alpha,\ A_1 = \beta_1,\ A_2 = \beta_2, ...,\ A_j = \beta_j,\ \text{где } \alpha = v_1 v_2 ... v_m, A \to \alpha \in P,$$

удовлетворяющих следующим условиям:

• каждый символ, входящий в β_i, принадлежит $V_{res} \cup V_N \cup \Delta$, либо является символом из V_N с индексом;

• если символ $B \in V_N$ имеет более одного вхождения в α, то каждый символ B_k соотнесен нижним индексом с конкретным вхождением B.

Символы из множества Δ в определении 2.2 являются прототипами тех функций, которые вычисляются для вершин синтаксического дерева. Заметим, что если семантических функций в СУ–переводе несколько, то одна из них представляет собой промежуточний код, а остальные являются вспомогательными.

В качестве примера рассмотрим трансляцию оператора описания простых переменных

$$\begin{array}{rll} G: & S & \to T\ P; \\ & T & \to int \mid float \\ & P & \to a \mid P, a \end{array}$$

Обозначим операционным символом Δ_0 вызов семантической подпрограммы записи переменной в семантическую таблицу с контролем повторности описания, символами S_{Code}, T_{Code}, P_{Code} — генерируемый промежуточний код, а символом T_{Type} — тип переменной. Тогда схема СУ–перевода может быть представлена в таблице:

правило грамматики	перевод	пояснения
$S \to TP;$	$S_{Code} = T_{Code} P_{Code}$	последовательное вычисление перевода для T и P
$T \to int$	$T_{Type} = TypeInt$ $T_{Code} = \varepsilon$	вычислили семантический тип
$T \to float$	$T_{type} = TypeFloat$ $T_{Code} = \varepsilon$	вычислили семантический тип
$P \to a$	$P_{Code} = \Delta_0$	занесли имя в таблицу
$P \to P, a$	$P_{Code} = P_{Code} \Delta_0$	обработали предшествующий список и занесли имя в таблицу

Рассмотренное определение 2.2 является универсальным методом описания семантики языков, однако требует существенной корректировки с учетом требования программирования реальных компиляторов. Последовательно рассмотрим соответствующие замечания.

1) Для того, чтобы ускорить процесс перевода и не выполнять перестановку фрагментов полученного результата, желательно так определить схему СУ–перевода, чтобы каждое правило схемы согласовывалась со структурой правила КС–грамматики по порядку используемых символов. Текст исходного модуля компилятор читает слева направо, поэтому действия по переводу также должны осуществляться в том же

порядке, в противном случае потребуется либо перестановка фрагментов сгенерированного кода, либо внесение изменений в уже построенное семантическое дерево. Естественно, что такие действия приведут к существенным усложнениям в коде компилятора.

2) Форма перевода 2.2 построена только на обработке правил КС–грамматики и не учитывает действия для некоторых терминальных символов. Однако на практике обработка терминала в момент его появления может существенно упростить процесс генерации кода.

3) Обозначения вычисляемых значений функций перевода для нетерминалов в форме A_i наглядны при записи схемы, но не всегда удобны при практической реализации. Дело в том, что любое вычисленное значение функции должно в программе компилятора где-то храниться. Очевидно, что вычисленные значения типов хранятся в области памяти одного типа, промежуточний код — в области памяти другого типа и т.п. Причем очевидно, что исходя из рекурсивной структуры синтаксиса анализируемой программы все вычисленные значения хранятся в рекурсивной структуре. Для организации указанных данных идеально подходит магазин с его принципом LIFO обработки данных. Но тогда, чтобы *явно указать область памяти*, куда заносятся вычисленные в процессе перевода значения, следует вместо A_i записывать функцию $i(A)$ или i_A.

Исходя из этих требований формула перевода A_i для правила КС–грамматики $A \to b_1 b_2 ... b_k$ для каждой из функций $i(A)$ принимает форму

$$i(A) = \phi_0 i(b_1)\phi_1 i(b_2)\phi_2 ... \phi_{k-1} i(b_k)\phi_k. \tag{2.1}$$

Кроме того, могут появиться правила для терминальных символов $t \in V_T$

$$i(t) = \phi_j, \tag{2.2}$$

где ϕ_0, ϕ_1, ... , ϕ_k, ϕ_j — цепочки из операционных символов и символов выходного алфавита.

d

2.3 Простейшие примеры СУ–схем

Рассмотрим перевод в триады операторов присваивания

$$\begin{aligned} G: \quad & S \to a = V; \\ & V \to V + A \mid V - A \mid A \\ & A \to A * E \mid A/E \mid E \\ & E \to a \mid c \mid (V). \end{aligned}$$

Сначала выделим функции, необходимые для построения перевода:

1) $\tau(b)$ — внутреннее представление исходного модуля, соответствующее конструкции b, $b \in \{V_T \cup V_N\}$;

2) $R(b)$ — функция, которая определяет местонахождение результата значения, соответствующего b;

3) k — текущий номер формируемой триады (для определенности будем считать, что значение k указывает на первую свободную позицию в массиве триад).

Рассмотрим правило $S \to a = V;$. Если $R(V)$ — результат, соответствующий вычислению R, то мы должны сформировать новую триаду

$$k) \ = \ a \ R(V)$$

Тогда при трансляции языка $C++$, в соответствии с правилами которого после выполнения операции присваивания результат сохраняется, мы можем формально определить перевод:

$$\begin{aligned}&\tau(S) = \tau(V)\ [k]\ =\ a\ R(V)],\\ &R(S) = R(V),\\ &k = k + 1.\end{aligned}$$

В языке Паскаль операция прсваивания не имеет результата, поэтому для Паскаля перевод изменяется для функции $R(S)$:

$$\begin{aligned}&\tau(S) = \tau(V)\ [k]\ =\ a\ R(V)],\\ &R(S) = \varepsilon,\\ &K = K + 1.\end{aligned}$$

Очевидно, что любое добавление очередной триады увеличивает значание k на количество добавляемых триад. Поэтому очевидную операцию $k = k+1$ в дальнейшем записывать не будем.

Для правила с бинарной операцией $V \to V + A$ перевод имеет вид:

$$\begin{aligned}&\tau(V_1) = \tau(V_2)\ \tau(A)\ [k]\ +\ R(V_2)\ R(A)],\\ &R(V_1) = (k).\end{aligned}$$

Рассмотрим правило $V \to A$. Очевидно, что применение такого правила ничего не добавляет к множеству триад, поэтому

$$\begin{aligned}&\tau(V) = \tau(A),\\ &R(V) = R(A).\end{aligned}$$

Аналогично определяется перевод для элементарного выражения, которое представляет собой выражение в скобках. Правила для элементарного выражения, представляющего собой простую переменную или константу, триад не формируют, зато формируют функцию $R()$:

$$\begin{array}{ll}E \to a & E \to c,\\ \tau(E) = \varepsilon, & \tau(E) = \varepsilon,\\ R(E) = a, & R(E) = c.\end{array}$$

Приведенный пример перевода не обеспечивает семантический контроль. Встроить семантический контроль очень просто, достаточно в нужных позициях вставить Δ–функции, что при реализации будет означать вызов соответстувующих семантических подпрограмм. Более того, нужно или или добавить еще одну функцию $T()$, значением которой является семантический тип выражения, или расширить функцию $R()$, рассматривая в качестве ее значений совокупность "тип — значение". Например, для правила с бинарной операцией $V \to V + A$ получим:

$$\begin{aligned}&T(V_1) = \Delta_1(T(V_2), T(A)),\\ &\tau(V_1) = \tau(V_2)\ \tau(A)\ [k]\ +\ R(V_2)\ R(A)],\\ &R(V_1) = (k).\end{aligned} \tag{2.3}$$

Здесь функция $\Delta_1(T(V_2), T(A))$ вычисляет тип результат операции над типами $T(V_2)$ и $T(A)$, а также, если нужно, генерирует триады для вызова стандартных функций преобразования данных из одного типа в другой. Сразу следует отметить, что при реализации перевода для вычисления каждой функции следует зарезервировть

определенную структуру данных. Так, множество триад — это массив, в который последовательно заносятся триады. Функция $R()$ должна быть реализована в виде магазина, из верхушки которого забираются операнды очередной триады. В виде магазина, синхронно изменяющегося вместе с $R()$, должна быть реализована функция $T()$.

Этот же перевод можно записать в соответствии с определением 2.2 и в виде единственной функции:

$$\tau(V_1) = \tau(V_2)\ \tau(A)\ \Delta_1\ \Delta_2, \tag{2.4}$$

Пусть магазин TR представляет собой массив структур, в каждом элементе которого хранится пара "тип — значение". Тогда в выражении (2.4) функция Δ_1 выполняет приведение типов над верхушкой магазина TR, генерирует при необходимости триады преобразования данных и помещает на верхушку этого магазина тип результата, не увеличивая указатель магазина. Функция Δ_2 генерирует триаду сложения над операндами в верхушке магазина TR, записывает на верхушку ссылку на результат и увеличивает указатель на TR. В результате в верхушке TR всегда создается пара "тип — значение".

Как мы увидим в дальнейшем, выбор способа представления в форме (2.3) или (2.4) иногда диктуется методом синтаксического анализа, а иногда может определяться только предпочтениями программиста.

Рассмотрим перевод этих же синтаксических конструкций в двоичное дерево. При этом для упрощения схемы не всегда будем уделять внимание семантическому контролю. Дерево для сложной синтаксической конструкции формируется из деревьев для составляющих элементов, поэтому достаточно использования единственной функции $form(a, x, y)$, первый аргумент которой — тип операции в создаваемой вершине, а оставшиеся аргументы — поддеревья, из которых создается новое дерево. Например, для правила $S \to a = V$;

$$\tau(S)\ = form(-\tau(a), \tau(V)), \tau(a)\ = form(a, NULL, NULL).$$

Для правила с бинарной операцией $V \to V + A$ перевод имеет вид:

$$\tau(V_1) = form('+', \tau(V_2), \tau(A)).$$

2.4 Согласование формы СУ–перевода со стратегией грамматического разбора

В соотвествии с определением (2.2) функции вычисления перевода для правила $A \to b_1 b_2 ... b_k$ и терминала a в общем случае имеют форму

$$\begin{array}{l} \tau(A) = \Delta_0 \tau(b_1) \Delta_1 i(b_2) \Delta_2 ... \Delta_{k-1} i(b_k) \Delta_k, \\ \tau(a) = \Delta_a. \end{array} \tag{2.5}$$

При восходящей стратегии к моменту вычисления $\tau(A)$ все $\tau_i(a_i)$ уже вычислены. Но в процессе этих вычислений не было известно, по какому правилу будет выполняться редукция, следовательно и вызов семантических подпрограмм Δ_0, Δ_1, ..., Δ_{k-1} был невозможен. Зато функция Δ_k может быть легко вызвана в момент редукции. Без проблем может быть вызвана и функция Δ_a в момент сканирования терминала a. Таким образом, может быть сделан следующий вывод:

• при восходящей стратегии разбора для нетерминалов следует использовать схему СУ–перевода в форме

$$\tau(A) = \tau(b_1)i(b_2)...i(b_k)\Delta_k,$$

• если в соответствии с логикой перевода внутрь правила необходимо вставить вызов семантических подпрограмм, для этого их нужно связать с терминалами, которые с необходимостью должны быть в указанном контексте. Более того, если транслируемый язык программирования не содержит нужных терминалов в требуемом контексте, придется изменить синтаксис этого языка, добавив нужные терминалы.

При нисходящей синтаксического стратегии анализа этих проблем нет. Форма правила (2.4) практически не отличается от формы обычного синтакисческого правила

$$A \to \Delta_0 b_1 \Delta_1 b_2 \Delta_2 ... \Delta_{k-1} i(b_k) \Delta_k,$$

если считать, что к Δ_i просто еще один тип нетерминалов. В магазин анализатора знаки Δ_i записываются наряду с символами b_i, а при их извлечении вызывается соответствующая семантическая подпрограмма.

2.5 Примеры СУ–схем сложных операторов

Пример 2.1. Перевод оператора if в триады.

$$\begin{array}{rll} G: \quad S \to & if\ (V)\ O\ else\ O \\ | & if\ (V)\ O \end{array}$$

В разделе 2.1 был рассмотрен промежуточний код для оператора if. Будем использовать триаду условного перехода, которая проверяет результат в регистре флагов и по условию $true$ переходит на триаду — первый операнд, а по условию $false$ — на второй. Запишем перевод в терминах функции τ:

$$\begin{array}{rlll} 0) & & & \\ i_1) & \tau(V) & & \\ i_1+1) & if & (i_1+2) & (i_2+2) \\ i_1+2) & ... & & \\ i_2) & \tau(O_1) & & \\ i_2+1) & go & (i_3+1) & \\ i_2+2) & ... & & \\ i_3) & \tau(O_2) & & \end{array}$$

В соответствии с ним можно предложить следующую схему СУ–перевода:

$$\begin{array}{rll} \tau(S) = & \tau(if)\tau(\text{'('})\tau(V)\tau(\text{')'})\tau(O_1)\tau(else)\tau(O_2)\Delta_{fi} \\ | & \tau(if)\tau(\text{'('})\tau(V)\tau(\text{')'})\tau(O_1)\Delta_{fi} \\ \tau(if) = & \varepsilon \\ \tau(\text{'('}) = & \varepsilon \\ \tau(\text{')'}) = & \Delta_{then} \\ \tau(else) = & \Delta_{else} \end{array}$$

Функция Δ_{then} должна сформировать недоформировнную триаду

$$i_1+1)\ if\ (i_1+2)\ \varepsilon,$$

запомнить адрес недоформированной триады в магазине недоформированных операций. Кроме того, при условии, что выражение V представляет собой элементарное выражение, нужно предварительно сформировать какую–нибудь триаду, результат трансляции которой в машинный код приведет к генерации операции, вырабатывающей признак в регистре флагов. Функция Δ_{else} должна доформировать недоформированную триаду if и сформировать новую недоформировнную триаду

$$i_2 + 1)\ go\ \varepsilon.$$

Функция Δ_{fi} доформировывает последнюю недоформированную триаду, которой может оказаться триада if или go.

Формула применима при любой стратегии разбора, но лучше согласуется с восходящей стратегией. При нисходящей стратегии разбора ее можно существенно упростить, включив вызов всех Δ–функций непосредственно в правило:

$$\begin{array}{rl} \tau(S) = & if(\tau(V))\Delta_{then}\tau(O_1)\Delta_{else}\tau(O_2)\Delta_{fi} \\ & |if(\tau(V))\Delta_{then}\tau(O_1)\Delta_{fi} \end{array}$$

Пример 2.2. Перевод процедур и функций в триады. Рассмотрим простейший пример, когда функции имеют тип $void$, а все параметры имеют тип int:

$$\begin{array}{rll} G: & S \rightarrow & void\ a(P)Q \\ & P \rightarrow & P\ ,\ int\ a \mid a \\ & D \rightarrow & a(F) \\ & F \rightarrow & F\ ,\ V \mid V \end{array}$$

Здесь нетерминал S — описание функции, P — список формальных параметров, D — вызов функции, F — список фактических параметров функции. Рассматривая реализацию интерпретаторов, мы уже говорили о выделении паияти, о прологах и эпилогах процедур и функций. Выберем вариант реализации, когда в эпилоге функции выделяется память для всех локальных данных этой функции. Тогда только после завершения анализа тела функции можно выяснить лействительную величину памяти, выделяемой в теле функции. Пусть некоторая стандартная подпрограмма $Prolog$ резервирует в стеке память, причем количество выделяемых байтов лежит в стеке. А подпрограмма $Epilog$ освобождает память в стеке. Фактически, простейший вариант реализации просто основан изменении содержимого регистров SP и BP. Но в данный момент нам проще считать, что нужные действия выполняют подпрограммы $Prolog$ и $Epilog$. Тогда вариант СУ–перевода для описания функции может иметь вид:

$$\begin{array}{rl} \tau(S) = & [k)\ proc\ a] \\ & \tau(P) \\ & \Delta_{PushEmpty} \\ & [k)\ call\ Prolog] \\ & \tau(Q) \\ & \Delta_{Push} \\ & [k)\ call\ Epilog] \\ & [k)\ endp] \end{array}$$

Семантическая подпрограмма $\Delta_{PushEmpty}$ запишет недоформированную триаду запаси в стек числа резервируемых байтов и запомнит адрес этой триады. Затем подпрограмма Δ_{Push} возьмет из семантического дерева число байтов, запишет его в недоформированную триаду, а также сформирует в текущей позиции триаду $push$ для использования ее эпилогом блока.

Для вызова функции СУ–схема еще проще:

$$\begin{array}{ll}\tau(D) = & \tau(F)\ [k]\ call\ a] \\ \tau(F_1) = & \tau(F_2)\ [k]\ push\ R(V)] \mid [k]\ push\ R(V)]\end{array}$$

2.6 Реализация синтаксически управляемого перевода

Фактически синтаксически управляемый перевод представляет собой правила вычисления одной или нескольких функций. Следовательно, первый вопрос, который следует решить — определить область памяти для вычисления этих функций. Рекурсивный принцип описания грамматики языка программирования (а, следовательно, и схемы перевода) означает, что и память для хранения значений вычисленных функций должна представлять собой соответствующую структуру. Как правило, главная функция $\tau(S)$, которая представляет собой промежуточний код конструкции S, формируется последовательно по мере обработки очередных конструкций. Поэтому для хранения значений $\tau(S)$ достаточно использовать массив с указателем на первый свободный элемент. Значения функций, соответствующих семантическим типам обработанных выражений, хранятся в магазине, так как при обработке очередной операции над данными, например, для правила с бинарной операцией $V \to V + A$ по формуле

$$\begin{array}{l}T(V_1) = \Delta_1(T(V_2), T(A)), \\ \tau(V_1) = \tau(V_2)\ \tau(A)\ [k]\ +\ R(V_2)\ R(A)], \\ R(V_1) = (k).\end{array} \tag{2.3}$$

вместо значений $T(V_2)$ и $T(A)$ в верхушке магазина, туда нужно поместить вычисленное значение типа $T(V_1)$.

Перейдем теперь к вопросу о том, куда и как встраивать Δ–функции в синтаксический анализатор. В соответствии с рассмотренными нами методами синтаксического анализа можно говорить о программировании СУ-=схемы для метода рекурсивного спуска, для магазинных методов нисходящего или восходящего анализа. Метод рекурсивного спуска не требует никаких специальных действий, достаточно встроить Δ–функции в синтаксическую диаграмму в качестве самостоятельных элементов.

Магазинные явные методы независимо от стратегии разбора требуют механизма реализации описания схемы в табличной форме. Неявные методы синтаксического анализа позволяют реализовать перевод существенно проще. В частности, при использовании нисходящих методов анализа в магазин дополнительно записываются символы, соответствующие вызлву Δ– функций. При нисходящем методе дополнительные действия встраиваются туда, где программируется редукция. Эти действия представляют собой вызов Δ–функции для правила грамматики $A \to b_1 b_2 ... b_k$ в формуле перевода

$$\tau(A) = b_1 b_2 ... b_k \Delta.$$

При восходящей стратегии для некоторых терминалов могут быть записаны функции перевода, поэтому при сканикорании таких терминалов следует вызвать соответствующие семантические функции.

2.7 Контрольные вопросы к разделу

1. Перечислите способы представления внутреннего кода.

2. Какими свойставми обладают деревья, представляющие внутренний код?

3. Чем триады отличаются от тетрад?

4. Как строится ПОЛИЗ?

5. Какой внутренний код по Вашему мнению является наиболее подходящей формой для интерпретации кода?

6. Зачем используются промежуточные языки при генерации кода? Поясните преимущества и недостатки использования промежуточных языков.

7. Дайте определение схемы синтаксически управляемого перевода.

8. Предложите представление оператора *switch* языка $++$ в форме триад.

9. Предложите представление оператора *switch* языка $++$ в форме ПОЛИЗ.

10. Чем префиксная запись выражений отличается от постфиксной?

11. Предложите представление оператора *do...while* языка $C++$ в форме бинарного дерева.

12. Как в ПОЛИЗ представить функцию?

13. Нужен ли специальный оператор, указывающий завершение ветви *then* оператора if языка $C++$, если в качестве внутреннего кода используется ПОЛИЗ? Поясните свой ответ.

14. Как Вы считаете, почему имеются трудности оптимизации выражений, представленных в форме ПОЛИЗ?

15. Есть фрагмент программы на языке $C++$:

$$a+=b+c*2; i=(a+b)/3+1; k=1;$$

Предложите представление этого фрагмента в форме ПОЛИЗ и в виде дерева. Покажите соответствие этих форм представления кода.

16. Предложите алгоритм интерпретации выражений, представленных в форме ПОЛИЗ.

17. Предложите алгоритм интерпретации условных операторов, представленных в форме ПОЛИЗ.

18. Допустим, Вы реализовали две программы интерпретатора: без использования внутреннего кода и с его использованием. Дайте краткую сравнительную характеристику этих программ.

19. Можно ли написать программу транслятора, которая работает без использования внутреннего кода?

20. Зачем нужен внутренний код?

2.8 Тесты для самоконтроля к разделу

1. Зачем при определении обощенного синтаксически управляемого перевода используется понятие операционных символов?

Варианты ответов:

а) для обозначения операций, которые будут сгенерированы в промежуточный код;

б) для обозначения вызовов семантических подпрограмм;

в) для обозначения операций, которые будут сгенерированы в ассемблерный код;

г) для обозначения правил генерации кода;

д) для обозначения функций, которые представляют собой синтаксически управляемый перевод для терминалов и нетерминалов.

Правильный ответ: б.

2. Может ли произвольная схема СУ–перевода быть использована как при восходящей, так и при нисходящей стратегии синтаксического анализа? Поясните свой ответ.

Варианты ответов:

а) Да, так схема перевода определяет смысл синтаксических конструкций языка программирования, а их смысл не зависит от стратегии синтаксического анализа.

б) Да, так как операционные символы обрабатываются синтаксическим анализатором по аналогии с обычными нетерминалами. А если КС–грамматика для какого-либо метода синтаксического анализа потребует преобразования, это можно сделать, рассматривая операционные символы как обычные нетерминалы в правой части правила.

в) Нет, так как восходящий анализ предъявляет очень жесткие требования к структуре правила перевода.

г) Нет, так как нисходящий анализ предъявляет очень жесткие требования к структуре правила перевода.

Правильный ответ: в.

3. Почему при синтакисческом анализе может оказаться, что терминальные символы связаны с вычислением Δ–функций?

Варианты ответов:

а) из–за особенностей реализации перевода при восходящем синтаксическом анализе,

б) из–за особенностей реализации перевода при нисходящем синтаксическом анализе,

в) по причине неэффективно сформированнного СУ–перевода,

г) по причине неправильно сформированнного СУ–перевода.

Правильный ответ: а.

4. Какие из следующих утверждений истинны?

1) Синтаксически управляемый перевод в терминах нескольких функций менее эффективен, чем с использованием только одной функции.

2) Для некоторых языков программирования синтаксически управляемый перевод с использованием только одной функции сформировать невозможно.

3) Любой перевод, записанный в терминах нескольких функций всегда можно преобразовать в перевод с использованием только одной функции.

4) Если семантических функций в СУ–переводе несколько, то одна из них представляет собой внутренний код, а остальные являются вспомогательными.

Варианты ответов:

а) все утверждения ложны;

б) 2 и 4;

в) 3 и 4;

г) 1, 3 и 4;

д) 2 и 3;

е) 2, 3 и 4;

ж) 1, 2 и 3;

з) все утверждения истинны.

Правильный ответ: в.

5. Поясните понятие "доформировать недоформированную триаду".

Варианты ответов:

а) Такого понятия в хорошо формализованном СУ–переводе просто не может быть, так как все триады формируются последовательно по мере обработки исходной программы.

б) Это понятие означает, что СУ–перевод построен неэффективно и иногда требуется вмешательство в уже сформированный код с целью изменения операндов некоторой триады.

в) Это означает, что в процессе генерации промежуточного кода обнаружилось, что нужно вставить еще одну или несколько триад в предшествующий код.

г) Это означает дополнение некоторой сгенерированной ранее триады параметрами, которые стали известны позже, чем была сгенерирована эта триада. Правильный ответ: г.

2.9 Упражнения к разделу

Цель данного задания – построить синтаксически–управляемый перевод для КС–грамматик, синтаксические анализаторы которых Вы писали, выполняя задания из первой части данного пособия. Работа должна быть выполнена для нисходящего и восходящего анализа.

Задание для нисходящего анализа должно быть выполнено в соответствии со следующим планом работ.

1. Выписать грамматику, в соответствии с которой Вы построили LL(1)-анализатор.

2. Ввести функции СУ–перевода. Одна из функций СУ–перевода должна соответствовать представлению дерева грамматического разбора в виде последовательности триад. Остальные функции имеют вспомогательное назначение.

3. Для каждой структурной единицы написать ее представление в виде последовательности триад, используя функции СУ–перевода.

4. Для каждого правила КС-грамматики записать определение функций СУ–перевода.

5. Ввести обозначение семантических подпрограмм контроля. Вставить соответствующие Δ–функции в СУ–перевод для каждого оператора, при трансляции которого необходим семантический контроль.

6. Перейти к программированию построенного перевода. Реализовать описание данных, соответствующих функциям перевода.

7. Построить подпрограммы формирования СУ–перевода в соответствии с Δ–функциями перевода.

8. Вставить в программу нисходящего синтаксического анализа фрагменты, соответствующие СУ–переводу:

- при записи в магазин правой части правила вместе с Δ–функциями;
- при чтении Δ–функции из верхушки магазина.

9. Отладить программу. Особое внимание следует уделить правильному порядку вызова семантических функций для правил КС–грамматики, которые были получены в процессе удаления левой рекурсии. Именно обработка таких правил часто выполняется неверно.

Задание для восходящего анализатора, который выполняет перевод в ПОЛИЗ, можно выполнить следующим образом.

1. Выписать грамматику, в соответствии с которой Вы построили восходящий анализатор.

2. Ввести функции СУ–перевода. Одна из функций СУ–перевода должна соответствовать представлению дерева грамматического разбора в виде ПОЛИЗ. Остальные функции имеют вспомогательное назначение.

3. Для каждой структурной единицы написать ее представление в виде ПОЛИЗ, используя функции СУ–перевода.

4. Для каждого правила КС–грамматики записать определение функций СУ–перевода.

5. Ввести функции перевода для терминальных символов, если это необходимо. Особое внимание обратить на однозначность перевода терминалов, т.к. соответствующие программные фрагменты должны быть встроены в программу синтаксического анализа в той ее части, где стоит вызов сканера для сканирования очередного терминала.

6. Ввести обозначение семантических подпрограмм контроля. Вставить соответствующие Δ–функции в СУ–перевод для каждого оператора, при трансляции которого необходим семантический контроль.

7. Перейти к программированию. Реализовать описание данных, соответствующих функциям перевода.

8. Построить подпрограммы формирования СУ–перевода в соответствии с Δ–функциями перевода.

9. Вставить в программу восходящего синтаксического анализа фрагменты, соответствующие СУ–переводу:

- для терминальных символов;
- при редукции по некоторому правилу КС-грамматики.

10. Отладить программу.

Глава 3

ОПТИМИЗАЦИЯ ВНУТРЕННЕГО КОДА

3.1 Граф управления

Сразу отметим, что оптимизации подлежит только такой внутренний код, который допускает свободное перемещение и удаление некоторых фрагментов. Из рассмотренных нами форм внутреннего кода этому требованию удовлетворяют только триады и тетрады. Будем рассматривать алгоритмы оптимизации в предположении, что внутренний код представляет собой последовательность триад.

Очевидно, что методы оптимизации линейных участков программы отличаются от оптимизации операторов, например, if, а методы оптимизации операторов ветвления принципиально отличаются от оптимизации циклов. Таким образом, возникает необходимость представить структуру программы в виде некоторой конструкции, удобной для применениия методов оптимизации. Такой структурой является граф управления. Вершины гарфа управления — эти линейные участки программы, т.е. такие последовательности триад, которые всегда последовательно выполняются от первой триады до последней, причем во всей программе отсутствуют переходы внуть линейных участков. Дуги соответствуют операциям передачи управления от одного линейного участка другому. Ориентированная дуга направлена от линейного участка a к линейному участку b, если после последней триады участка a может быть выполнен переход на первую триаду участка b. Таким образом, граф управления строится в процессе последовательной обработки триад, причем дуга может появиться только при обработке триады с операцией go или if.

Очевидно, что все линейные участки, кроме начального, для которых число входящийх дуг равно нулю, представляют собой так называемый "недостижимый код", который можно удалить из программы.

В графе управления легко можно найти ветвления и циклы, которые подлежат оптимизации:

• ветвления — это два или более путей между двумя вершинами графа,

• циклы — это такие сильносвязанные участки графа управления, которые либо не пересекаются, либо строго содержатся один в другом.

Примеры циклов и ветвлений в гарфе управления представлены на рисунке 3.1. Здесь представлены две параллельные ветви 2—3 и 4 одного ветвления, а также три цикла 2—3, 9, 6—7—8—9. Сильносвязанные участки 6—7—8 и 7—8—9 не могут оптимизироваться как отдельные циклы, так как они не являются вложенными один в другой и их пересечение не пусто.

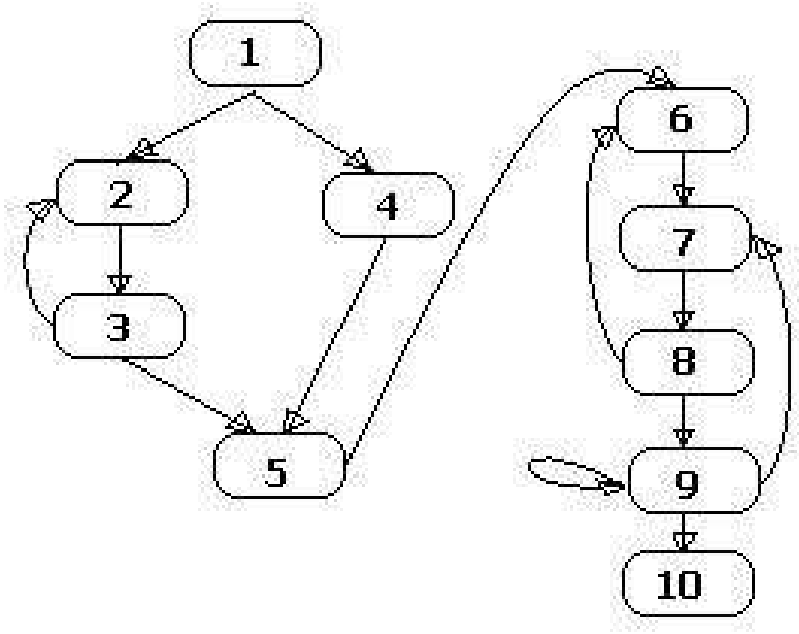


Рис. 3.1: Граф управления с ветвлением и циклами

3.2 Оптимизация линейных участков

Линейные участки соответствуют фрагменту программы, который содержит только выражения (операторы присваивания в том числе). Вычисления представляют собой основную часть выполняющегося кода, поэтому их оптимизация должна приводить к существенному повышению производительности.

Простейшая оптимизация на линейном участке — это вычисление некоторых выражений на фазе компиляции. Обычно в программе существует много констант, в том числе именованных. Константы участвуют в выражениях, в результате некоторые фрагменты выражение и даже переменные вычисляются как константные выражения. Ятобы не выполнять лишние вычисления, компиляторы выполняют вычисление константных на фазе компиляции. Для этого Ддостаточно найти триады, в которых все операнды являются константами. Операцию над константами надо выполнить, триаду удалить, а затем заменить все ссылки на удаляемую триаду на новую константу — результат выполнения операции. В результате такого алгоритма, например, код

```
#define MAX_NUM 100
#define MAX_COUNT 500
int a = (6+MAX_NUM * MAX_COUNT);
double b =a*2;
```

превратится в

```
int a = 50006;
double b = 100012.0;
```

Следующий простейший метод оптимизации — удаление повторяющихся триад. Попарно сравниваем триады, причем сравнение может быть только при условии, что между сравниваемыми триадами не изменяется значение их операндов. Здесь

следует отметить, что изменение может быть явным, когда в некоторой триаде переменной присваивается новое значение. Однако, изменение значение может быть невидимым на этапе трансляции, так как вызываемая функция может изменить значение переменной. Поэтому вызов функции также является границей, после которой недопустимо считать совпадающие по форме триады взаимозаменяемыми. Рассмотрим пример:

```
A = (B * C + D * K )* T;
D = D * K + B * C;
K = D * K + B * C;
```

исходные триады	один проход оптимизации	результат оптимизации
0) * B C	0)*BC	0)*BC
1) * D K	1) * D K	1) * D K
2) + (1)(0)	2)+ (1)(0)	2) + (1)(0)
3) * T (2)	3) * T (2)	3) * T (2)
4) = A (3)	4) = A (3)	4) = A (3)
5) * D K		
6) * B C		
7) + (5)(6)	**7) + (1)(0)**	
8) = D (7)	8) = D (7)	8) = D (2)
*9) * D K*	9) * D K	9) * D K
10)* B C		
11)+ (9)(10)	11)+ (9)(0)	11)+ (9)(0)
12)= K (11)	12)= K (11)	12)= K (11)

Анализ приведенного примера показывает большие ограничения метода, основанного на полном совпадении триад. Достаточно было в первом выражении поставить скобки или просто переставить операнды — и триады перестанут совпадать даже с учетом коммутативности операции умножения. Более сложный метод поиска совпадений основан на конструировании агрегатов коммутативных и ассоциативных операций. Назовем агрегатом множество непосредственных операндов, связанных ассоциативной и коммутативной операцией. Пример агрегата для операции умножения представлен на рисунке 3.2.

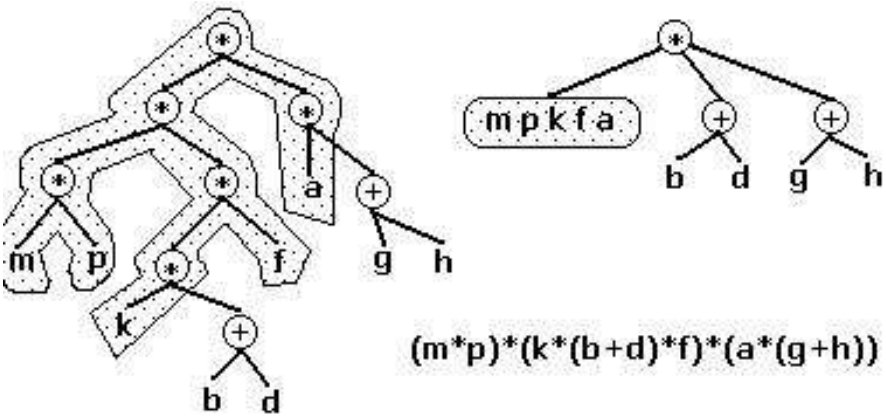


Рис. 3.2: Агрегат для операции умножения

При реализации определение агрегата можно упростить:

• если оба операнда операции ”*” элементарные операнды, то агрегат для этой триады — это множество, содержащее оба операнда;

• если оба операнда операции ”*” ссылки на триады, имеющие непустые агрегаты, то агрегат для этой триады — это множество, равное объединению агрегатов дочерних триад;

A + Алгоритм оптимизации : * * A R 1)Попарно сравниваем агрегаты для АК-опе- ВС DKT раций с учетом границы такого сравнения (на :=) 2)Если пересечение агрегатов содержит более одного элемента, реализовать представление в виде триад соответствую- щего элемента и перестроить триады с Опять же любое учетом ссылки на сформированную после- повторяется довательность.

4)Удаление лишних вычислений Если результат некоторой триады нигде не используется, ее можно удалить.

а) выполнение операций, оба операнда - константы. 1) + a 2 2) * 3 5 (2) тождественно 15-ти 3) + (2) 7 (3) тождественно 22-ум 4) * (1) (3) тождественно * (1) 22

Замечание: триады с констант-операндами появляются в программе, так как развитые ЯП позволяют использовать именованные константы и макросредства.

б) устранение лишних операций. 1) + a 2 2) * (1) b 3) + 2 a 4) / (3) 5 5) * (4) (2)

Триада называется линейной, если у неч совпадают операция и операнды с точностью до коммутативности операндов и между совпадающими триадами не должно быть триады, в которой изменяется значение одного из операндов.

Замечание. Сложность алгоритма квадратичная, но просмотр осуществляется последовательно и до первого изменения операнда.

в) оптимизация агрегатов ассоциативных и коммутативных операций. 1) + a 2 2) + (1) b 3) + b a 4) + 2 (3) 5) * (2) (4)

из (1) и (2): a+2+b из (3) и (4): b+a+2

С учетом коммутативности и ассоциативности получаем дерево.

Алгоритм оптимизации. 1) Последовательно обрабатывается каждая операция из числа коммутативных ассоциативных. Рекурсивно можно дать определение агрегата. 2) Агрегат имеет 2 непосредственных операнда. Операндами агрегата являются один или 2 агрегата, при этом оставшиеся операнды - непосредственные. 3) Все агрегаты (условие неизменности операндов такое же как в б)) оптимизируются вместе. Для этого строится граф с целевыми вершинами - элементы агрегатов и промежуточными вершинами - произвольным пересечением агрегатов. Оптимальный подграф определяет способ вычисления агрегатов.

3.3 Оптимизация ветвлений

Ветвления можно обнаружить в графе управления, так как оно является таким подграфом графа управления, который имеет один исток и один сток, причем ветви *then* и *else* могут иметь сложную структуру.

Оптимизация заключается в том, что одинаковые начала ветвей *then* и *else* выносятся в присоединяются к истоку в его конце. Аналогично выполняется оптимизация на концах ветви *then* и *else*, причем проверку на совпадение следует начинать не с последней триады ветвей, а с первых триада последних операторов ветвей *then* и *else*.

Пример: Исходный код имеет вид:

```
if (a>b) {
   c=2*d;
   one = func(2,c);
   y=x*x;
   a=x-y;
}
else {
   c=2*d;
   two = func(2,c);
   t= a+b-two;
   a=x-y;
}
```

В процессе трансляции будут построены триады:

```
 1) >    a     b
 2) if   (3)   (14)
 3) *    2     d     // c=2*d;
 4) =    c     (3)
 5) push 2           //  one = func(2,c);
 6) push c
 7) call func
 8) =    one   (7)
 9) *    x     x     // y=x*x;
10) =    y     (9)
11) -    x     y     // a=x-y
12) =    a     (11)
13) goto (24)
14) *    2     d     // c=2*d;
15) =    c    (3)
16) push 2           //  two = func(2,c);
17) push c
18) call func
19) =    two  (18)
20) +    a    b      // t= a+b-two;
21) -    (20) two
22) =    y    (21)
22) -    x    y      // a=x-y;
23) =    a    (11)
24) nop
```

Результат оптимизации без перенумерации триад:

```
 1) >    a     b
1.1)  = temp (1)
 3) *    2     d
 4) =    c     (3)
 5) push 2
 6) push c
```

```
 7) call func
1.2) cmp temp 0
 2) if   (8)   (19)
 8) =    one   (7)
 9) *    x     x
10) =    y     (9)
13) goto (24)
19) =    two   (7)
20) +    a     b
21) -    (20)  two
22) =    y     (21)
24) nop
11) -    x     y
12) =    a     (11)
```

3.4 Оптимизация циклов

Первая задача при оптимизации цикла - увидеть цикл. В графе управления нужно увидеть цикл, так как безграмотная конструкция программы может привести к тому, что построенные с помощью операторов do, while и т.д. могут оказаться не оптимальными из-за переходов. В графе управления циклом является только такой сильно связанный подграф, в котором есть один вход и один выход. Если таких фрагментов несколько, то можно рассматривать только такие подграфы, которое либо лежат один в другом, либо имеют пустое пересечение.

1,2,3 и 2,3,4 - не циклы 5 и 6 и 5,6,7 - циклы

Методы. а) Вводится понятие инвариантной триады, аргументы этой триады в цикле не изменяются. Инвариантную триаду выносят в начало и ставят перед циклом. Замечание. Цикл с постусловием оптимизируется без дополнительных ограничений; цикл с предусловием - с копированием условия и может быть выполнением только при отсутствии побочного эффекта, связанного с вызовом функции.

б) Оптимизация структуры цикла с постусловием

с // с стирается и делается переход к с в цикле. do a b c while (f);

в) Устранение мультипликативных операций.

for (i=0;i<n;i++) j=a*i+b; k=2*j-5; ...

i - аддитивная переменная, если оная изменяется в цикле с постоянным шагом, еч изменению соответствует шаблон: iнов=iстар+hi

inc dec + i h = i ()

iнов=a*iнов+b=a*(iстрар+hi)+b=a*iстар+b+астар*hi=jстар+a*hi=jстар+pj

1) Вынесение инвариантных операций for (;;) A=B*C; D=A+K*R; F=B*D; Операция называется инвариантной, если ее операнды в цикле не из- меняются. Такие операции можно вынести за пределы цикла - в начало При этом любые передачи в графе управления программы на данный цикл изменяются на передачу управления на полученный линейный участок. В результате преобразований может оказаться, что тело цикла не содержит никаких операторов, кроме изменений параметров цикла Такой цикл заменяется присвоением последнего значения параметру циклов. 2) Замена сложных операций простыми с учетом аддитивных переменных. Переменная называется аддитивной в цикле, если она изменяется с постоянным шагом. i=i+b [*

i a] i=a*i+b [+(1) b] i [= i (2)]

После того как нашли эту переменную можем найти другую j=K*i+t j,nob,= k+inob,+t = k(a*i,ct,+b)+ t = (k*i,ct)*a +k*b + t = = j,ct,*a+k*b+b-a*t = j,ct,a + h,j В результате j становится аддитивной переменной и дальнейший поиск таких переменных может продолжаться. Если a=1, то j на каждом шаге просто увеличивается на пос- (для i) тоянный шаг 3) Поиск совпадающих элементов в параллельных ветвях ————————æ ʒ —+–æ ʒ ʒ * * ʒ ʒ L—— ʒ ——æ ——æ ʒ ʒb=d + aʒ ʒt=a+1ʒ ʒ ʒk=b * tʒ ʒb=d+aʒ ʒ L—— ʒk=b*tʒ ʒ L—— ʒ —æ ʒ ʒ * ʒ ʒ L-T— L————— Рассмотренные выше методы оптимизации циклов применимы толь- ко к тем участкам цикла, через которые проходит любой путь от начала к концу данного цикла. Если окажется, что в графе для цикла такие номера участков, что каждый путь проходит ровно через один из них и только через один. Если некоторая последовательность триад встречается на каждом из таких участках, ее можно вынести в верхний блок.

3.5 Оптимизация регистров для вычисления выражений

Рассмотрим пример уложенного в линию дерева как, например, представлено на рисунке 2.4. Допустим, что это выражение транслируется в последовательность команд языка ассемблера, причем все операции выполняются только с помощью команд типа регистр–регистр. Тогда дополнительно потребуются еще и команды загрузки данных в регистр. Если выражение транслируется без изменения порядков следования операндов, то легко заметить, что число занятых в какой-либо момент регистров равно количеству таких дуг дерева, которые расположены над соответствующей точкой в линейной укладке выражения. Следовательно, общее число регистров, необходимых для вычисления выражения, равно максимальному количеству дуг в вертикальном срезе дерево. Например, для вычисления выражения, представленного на рисунке 2.4, число используемых регистров равно трем.

Уменьшить число регистров можно с помощью измения порядка вершин в укладке дерева так, чтобы минимизировать число дуг в максимальном вертикальном срезе. Очевидно, что нам помогут функции, представляющие собой стоимость вычисления выражения. Пусть функция $Reg(a)$ представляет собой количество занятых регистров для трансляции вершины дерева a, а функция $l(a)$ — оптимальная укладка поддерева, корнем которой является вершина a. Построим следующий синтаксически— управляемый перевод Рассмотрим общий случай n–местной операции. Для простоты описания синтаксиса будем считать, что в правилах КС–грамматики опущен знак операции, а все операнды расположены последовательно. Пусть, например, такое правило имеет вид

$$A \rightarrow a_1 a_1 ... a_n,$$

и нам уже известна оптимальная укладка $l(a)$ каждого из поддеревьев a_i и соответствующее число регистров $Reg(a_i)$. В частности, для простого операнда нужен один регист и его укладка совпадает с изображением. Тогда минимальное число регистров для трансляции A получим, если отсортируем вершины a_i в порядке невозрастания $Reg(a_i)$ и уложим их в отсортированном порядке. Требуемое число регистров в этом случае равно

$$Reg(A) = max\{Reg(a_{i1}) + 0, Reg(a_{i2}) + 1, Reg(a_{i3}) + 2, ..., Reg(a_{in} + n - 1)\},$$

где a_{i1}), a_{i2}, ..., a_{in} — вершины правой части правила в отсортированном порядке. Оптимальная строка укладки дерева определяется формулой

$$l(A) = l(a_{i1})l(a_{i2})...l(a_{in}oper,$$

где *oper* — знак операции.

Рассмотрим пример оптимизации выражения

$$a * b + (c * d + (e * f + (g * h + (k * l + (m * n))))).$$

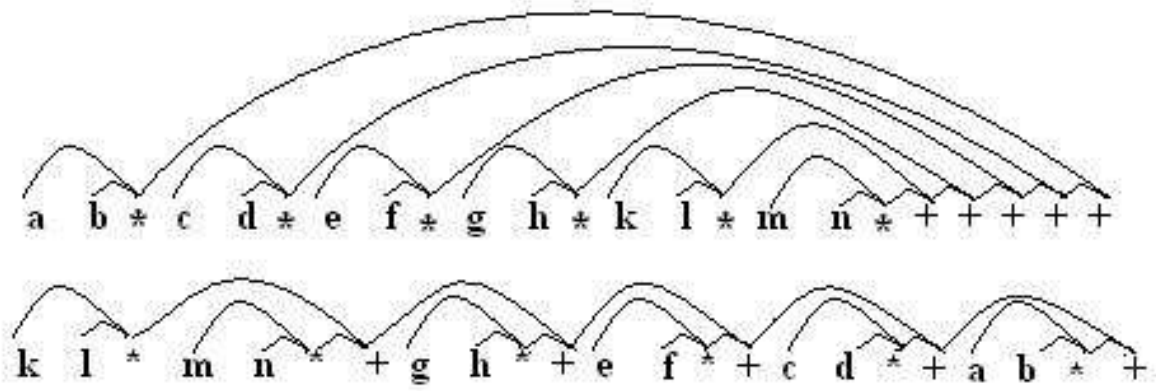


Рис. 3.3: Пример оптимизации: семь регистров в исходном выражении и три в оптимизированном

Покажем, как после оптимизации необходимое число регистров сократилось до трех вместо изначально требуемых семи (рисунок 3.3). Все вычисления сведем в таблицу:

a_i	$Reg(a_i)$	$l(a_i)$
a	1	a
b	1	b
$*$	$max\{1+0, 1+1\} = 2$	$ab*$
c	1	c
d	1	d
$*$	$max\{1+0, 1+1\} = 2$	$cd*$
e	1	e
f	1	f
$*$	$max\{1+0, 1+1\} = 2$	$ef*$
g	1	g
h	1	h
$*$	$max\{1+0, 1+1\} = 2$	$gh*$
k	1	k
l	1	l
$*$	$max\{1+0, 1+1\} = 2$	$kl*$
m	1	m
n	1	n
$*$	$max\{1+0, 1+1\} = 2$	$mn*$
$+$	$max\{2+0, 2+1\} = 3$	$kl*mn*+$
$+$	$max\{3+0, 2+1\} = 3$	$kl*mn*+gh*+$
$+$	$max\{3+0, 2+1\} = 3$	$kl*mn*+gh*+ef*+$
$+$	$max\{3+0, 2+1\} = 3$	$kl*mn*+gh*+ef*+cd*+$
$+$	$max\{3+0, 2+1\} = 3$	$kl*mn*+gh*+ef*+cd*+ab*+$

3.6 Контрольные вопросы к разделу

1. Какой внутренний код по Вашему мнению является наиболее подходящей формой для оптимизации кода? Почему?

2. Как используется граф управления программы?

3. Поясните понятие линейного участка программы.

4. Как найти циклы, подлежащие оптимизации?

5. Какие виды оптимизации кода на линейных участках Вы знаете?

6. Приведите пример оптимизированного кода линейного участка.

7. Как оптимизируются ветвления?

8. Как Вы думаете, чем отличается оптимизация операторов $while$ и $go - while$?

9. Пояните определение агрегата коммутативных и ассоциативных операций.

10. Как вычислить агрегаты коммутативных и ассоциативных операций, если имеется последовательность триад?

11. Почему нужно оптимизировать число регистров при трансляции выражений?

12. Приведите формулы вычисления оптимальной укладки дерева выражения с минимальным числом используемых регистров.

13. Как изменится алгоритм оптимизации числа регистров, если при трансляции выражения можно использовать команды типа регистр–память?

14. Как определяются границы фрагмента, на котором осуществляется поиск совпадающих триад?

15. Влияют ли вызовы функций внутри выражения на оптимизацию соответствующего линейного участка? Если влияют, то каким образом?

3.7 Упражнения к разделу

Цель данного задания – предложить алгоритм оптимизации промежуточного кода. Задание должно быть выполнено в соответствии со следующим планом работ.

1. Рассмотреть структуру внутреннего кода, который Вы построили при выполнении предшествующего задания, и определить конструкции, которые можно оптимизировать. Рассмотреть возможность оптимизации как по времени выполнения программы, так и по используемой памяти.

2. Выбрать тип оптимизации, в результате выполнения которой будет получен наибольший эффект.

3. Реализовать программу оптимизации в соответствии с выбранным методом.

4. Отладить программу. Особое внимание обратите на эквивалентость программ.

Глава 4

ГЕНЕРАЦИЯ КОДА

4.1 Принципы генерации ассемблерного кода

В результате работы блока анализа выполняются все действия, указанные в схеме синтаксически управляемого перевода. Это означает, что при использовани многопроходной схемы трансляции мы получаем две конструкции:

• семантическое дерево, в котором отражена вся информация о данных и типах программы,

• последовательность триад, которая соответствует выполняющемуся коду программы.

Эти данные поступают на вход генератора кода. В зависимости от типа триад генератор формирует различные конструкции кода на языке Ассемблера. По типу триады можно разделить на триады перехода, триады операций над данными, триады начала и конца функций. Вся информация об описаниях данных и типов в списке триад отстутствует, она находится только в семантическом дереве. Разные компиляторы могут генерировать существенно различающийся ассемблерный код, но принципы остаются неизменными:

• глобальные данные размещаются в сегменте данных, локальные — в стеке,

• команды зависят от типов данных и в соответствии с системой команд выполняются над данными одинакового типа (и разрядности в том числе).

4.1.1 Функции

Триады, отмечающие начала и концы функций, заключаются в функциональные скобки — операторы ассемблера $< name > PROC$ и $< name > ENDP$. Нусть, например, есть объявление функции:

```
void example(int param1, double param2){   ...   }
```

Тогда Visual Studio может сгенерировать код

```
?example@@YAXHN@Z PROC
  ...
?example@@YAXHN@Z ENDP
```

Операторы $PROC$ и $ENDP$ помечаются метками, которые генерирует компилятор из фактического имени функции и стандартных префикса и суффикса.

Код любой функции начинается с пролога. В прологе выделяется в стеке память для локальных данных и сохраняется содержимое некоторых регистров. Кроме этого, в некоторых реализациях пролог может содержать обнуление локальной памяти, проверку на переполнение стека при выделении памяти под локальные переменные и т.п. Размер выделяемой в стеке памяти вычисляется по информации из семантического дерева о локадьных данных функции. Кроме того, к этому размеру может быть добавлен фрагмент для хранения некоторых промежуточных данных. Иерархия выделения памяти в стеке при последовательном вызове функций основана на том, что в регистре *ebp* хранится текщее значение свободной позиции в стеке. Уменьшая значение регистра *esp* на величину выделяемого фрагмента, мы тем самым устанавливаем значение указателя стека на текущую верхушку стека. Если перед этим сохранить текущее значение регитсра *esp* в *ebp*, то в теле функции адресацию всех параметров и локальных данных можно выполнять с использованием регистра *ebp* с положительным или отрицательным смещением соответственно. Схематически это изображено на рисунке 4.1.

Рис. 4.1: Схема выделения памяти в стеке

Например, пролог может быть таким:

```
?example@@YAXHN@Z PROC
        push        ebp
        mov         ebp,esp
        sub         esp,192 ; выделяется память
        push        ebx
        push        esi
        push        edi      ; сохраняются регистры в стеке
        ...
?example@@YAXHN@Z ENDP
```

Перед командой возврата из функции генерируются команды эпилога. Главной задачей эпилога является освобождение памяти, а это означает, что содержимое регистров *ebp* и *esp* необходимо восстановить:

```
$LN1@example:
        pop edi
        pop esi
        pop ebx
        mov esp, ebp
        pop ebp
        ret 0
```

Метка перед командами эпилога генерируется для того, чтобы при трансляции любого оператора *return* в теле функции выход осуществлялся только через эпилог.

4.1.2 Переходы

Генерация условных и безусловных триад перехода выполняется очень просто благодаоя тому, что в качестве операнда в триаде перехода указан номер триады *number*, на которую нужно сгенерировать переход, а в команде ассемблера в качестве операнда нужно указать метку соответствующего фрагмента программы. Проблема легко решается, если в генерируемой программе на ассемблере в качестве метки использовать конструкцию типа $prefix + number + suffix$, где $prefix$ и $suffix$ — стандартные префиксы и суффиксы метки. Чтобы упростить структуру генерируемой программы, метка ставится только перед началом каждого линейного участка. После завершения генераци всей программы неиспользуемые метки можно удалить. Например, при трансляции оператора

```
if ( data > param1 ) data= 1 ;
else data = 2;
```

генерируется код

```
_data$ = -8  ;  локальная переменная
_param1$ = 8 ; параметр функции
        ...
        ...
        mov eax, DWORD PTR _data$[ebp]
        cmp eax, DWORD PTR _param1$[ebp]
        jle SHORT $LN4@example
        mov DWORD PTR _data$[ebp], 1
        jmp SHORT $LN3@example
$LN4@example:
        mov DWORD PTR _data$[ebp], 2
$LN3@example:
```

Обратите внимание на значения смещения относительно регистра *ebp* для локальных данных и параметров функции!

4.1.3 Вызов функции

Рассмотрим трансляцию вызова функции. Параметры в функцию передаются через стек. В зависимости от так называемого соглашения о связях выбираются варианты реализации:

• выбирается порядок записи параметров в стек: параметры записываются в стек слева направо или справа налево;

• определяется позиция, в которой параметры удаляются из стека при завершении функции: параметры удаляются в функции перед командой *ret* или параметры удаляются вызывающей программой после возврата из вызванной функции.

Например, если параметры записываются в стек начиная с последнего, а удалением их из стека занимается вызывающая программа, то для вызова

```
#define  MAX_STR 1003*2
__int64   num[2][MAX_STR];
memset(&num[0][0],   '\0',   sizeof(num));
```

в ассемблерной программе появится код:

```
     ; запись параметров в стек:
push 32096 ; sizeof(num)
push 0
push OFFSET ?numRect@@3PAYOHNG@_JA ; numRect
call _memset
     ; убрать параметры из стека:
add esp, 12
```

4.1.4 Выражения

Очевидно, что информация из семантического дерева должна использоваться для генерации кода. И дело здесь не только в том, что коды команд зависят от типа операндов. В программе в зависимости от того, где и как объявлены данные, они соответственно размещаются в памяти. Примеры генерации кода для глобальных и описанных локально в функции целых данных приведены в таблице:

	int ia,ib,ic; ic = ia + ib;	char ca,cb,cc; cc = ca + cb;

Глобальные данные	mov eax, [ia]
	add eax, [ib]
	mov [ic], eax
		movsx eax, byte ptr [ca]
		movsx ecx, byte ptr [cb]
		add eax, ecx
		mov byte ptr [cc], al

Данные в теле функции	mov ecx, [ebp-0x04]
	add ecx, [ebp-0x08]
	mov [ebp-0x0c], ecx
		movsx eax, byte ptr [ebp-0x04]
		movsx ecx, byte ptr [ebp-0x08]
		add eax, ecx
		mov byte ptr [ebp-0x0c], al

Примеры генерации кода для глобальных и описанных локально в функции вещественных данных приведены в таблице:

	float fa,fb,fc; fc = fa + fb;	double da,db,dc; double da,db,dc;
Глобальные данные	fld dword ptr [fa] fadd dword ptr [fb] fstp qword ptr [dc]	fld qword ptr [da] fadd qword ptr [db] fstp qword ptr [dc]
Данные в теле функции	fld dword ptr [ebp-0x14] fadd dword ptr [ebp-0x1c] fstp dword ptr [ebp-0x24]	fld qword ptr [ebp-0x20] fadd qword ptr [ebp-0x28] fstp qword ptr [ebp-0x30]

4.1.5 Типы используемых регистров

Перейдем теперь к алгоритмам генерации кода для сложных выражений. Сложное выражение со скобками и операицями разных приоритетов потребует в общем случае использования четырех типов триад:

– + a b // оба операнда - константы или идентификаторы (непосредственные операнды),

– + (i) b // ссылка на триаду, и непосредственный операнд

– + a (i) // непосредственный операнд и ссылка на триаду,

– + (i) (j) // оба операнда — ссылки на триады.

Значения выражений над данными вещественных типов вычисляются в регистрах чисел с плавающей точкой ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7, а значения целых типов вычисляются в регистрах процессора Intel EAX, EBX, ECX, EDX, ESI, EDI. При трансляции выражений используется функция выделения очередного свободного регистра, генерируются команды, а затем для каждой триады следует запоминать имя регистра, в котором вычислено значение. Таким образом, для триады, в которой оба операнда являются непосредственными операндами, сначала нужно выделить один регистр (при использовании команд типа "регистр - память") или два регитсра (при генерации команд типа "регистр - регистр"). Для остальных типов триад выделять регистр нужно не всегда. Более того, если оба операнда в регитсрах, то после генерации команды один из этих регистров освобождается. Особенности алгоритмов выделения регистров рассмотрим далее.

4.2 Назначение регистров

Результаты выполнения операций должны находиться в регистре, это значит, что из множества свободных регистров в момент трансляции необходимо запросить какой-нибудь регистр. Для этого в программе используется логическая шкала свободных регистров. Программа выделения регистров работает в двух режимах:

• выделить какой-нибудь регистр нужного размера,

• выделить заданный регистр. Например, команды умножения и деления требуют определенных регистров.

В обоих случаях может оказаться, что нужного регистра нет. Это означает, что все регитсры заняты промежуточными результатами, которые в дальнейшем обязательно будут нужны. Тогда используется один из двух вариантов временного хранения вычисленных данных:

• в стеке с использованием обычных операций со стеком *push* и *pop*,

• в дополнительной области памяти, которая выделяется в стеке в начале функции.

Хранение данных в стеке — это самый очевидный и простой способ, но он может корректно работать только при условии, что транслируемое выражение не было оптимизировано, а, следовательно, используются результаты триад в том же порядке, в каком эти триады появились в процессе генерации промежуточного кода. При хранении промежуточных результатов в специально выделенной для этого области памяти таких ограничений нет. Более того, если одно и то же подвыражение в процессе оптимизации используется неоднократно, никаких проблем это не вызывает. В триаде появляется дополнительное (четвертое) поле, указывающее, где хранится результат. Оно представляет собой:

- FlagRegister — признак хранения результата в регистре,

- NameResult — текст, который представляет собой имя регитсра и области памяти, в которой хранится значение.

При запросе отсутствующего свободного регистра программа просматривает уже сгенерированные триады, начиная с последней, находит занятий подходящий регистр, генерирует команду сохранения содержимого найденного регистра, модияицирует поле результата в триаде.

4.3 Понятие объектного кода и его структура

Object code - объектный код - код, который создается транслятором из исходного кода. Объектный модуль — это неделимое единство кода и другой информации, создаваемое в результате работы транслятора из одного исходного файла. Для того, чтобы программа могла выполняться, нужно запустить редактор связей, который преобразует объектный код в выполняемый код.

Вообще говоря, объектные файлы бывают трех видов:

• Перемещаемый объектный файл. Содержит бинарный код и данные в форме, пригодной для связывания с другими перемещаемыми объектными файлами. В результате соединения таких файлов получается исполняемый файл.

• Исполняемый объектный файл. Содержит бинарный код и данные в форме, пригодной для загрузки программы в память и ее исполнения.

• Разделяемый объектный файл. Особый тип перемещаемого объекта. Его можно загрузить в память и связать динамически, во время загрузки или во время выполнения программы.

Компиляторы и ассемблеры генерируют перемещаемые и разделяемые объектные файлы. Компоновщик связывает эти объектные файлы, в результате чего получается исполняемый объектный файл.

Редактор связей (или компоновщик) — это системная обрабатывающая программа, редактирующая и объединяющая объектные ранее оттраслированные модули в единый загрузочный готовый к выполнению код. Первая задача компоновщака заключается в назначении адресов загрузки различным частям программы. В процессе назначения адресов происходит выравнивание на требуемые границы, и таким образом во время выполнения везде имеются корректные адреса. Вторая задача компоновщака заключается в сборке прпрограммного кода из отдельных модулей проекта и статических бибилиотек: компоновщик добавляет к компонуемой программе коды библиотечных функций (они обеспечивают выполнение математических и стандартных функций, вывод информации на экран монитора и т.п.), а также код, обеспечивающий размещение программы в памяти, еч корректное начало и завершение. Таким образом, связывание - это процесс объединения различных элементов кода и данных для получения одного исполняемого файла, который затем может быть загружен в память. Операционная система помещает загрузочный модуль в оперативную память и запускает его на выполнение.

Для того, чтобы компоновщик мог выполнить указанные задачи, объектный код должен содержать как минимум следующую информацию:

• имя модуля,

• программный код, соответствующий оттранслированным операторам,

• используемые в программе внешние данные,

• определенные в программе имена, на которые могут сылаться другие модули (внешние ссылки),

• точка входа в главную программу.

Объектные файлы различаются для разных операционных систем. Для разных операционных систем существуют различные стандарты, но все они используют один из двух способов хранения данных:

• табличный способ (в современных версиях UNIX используется формат UNIX ELF – executable and linking format, формат исполнения и связывания),

• в виде множества записей (Windows).

Табличный способ реализуется в виде заголовка, за которым следуют таблицы кодов, внешних имен, ссылок и т.д. В заголовке указаны длины соответствующих таблиц. В соответствии с форматом UNIX ELF формат типичного перемещаемого объектного файласодержит следующие поля:

• Заголовок ELF;

• .text — машинный код скомпилированной программы;

• .rodata — данные только для чтения, такие как строки формата printf;

• .data — инициализированные глобальные переменные;

• .bss — неинициализированные глобальные переменные (BSS означает Black Storage Start — начало блока хранения); эта секция не занимает места в объектном файле;

• .symtab — таблица символов с информацией о функциях и глобальных переменных, определенных и упоминаемых в программе (фактически, эта таблица содержит только видимые наружу имена и в принципе не должна содержать имен локальных переменных);

• .rel.text — список относительных адресов в секции .text, которые нужно изменить при связывании этого объектного файла с другими объектными файлами (фактически, это адреса внешних идентификаторов);

• .rel.data — информация о размещении глобальных переменных, упомянутых, но не определенных в текущем модуле;

• .debug — таблица отладочных символов, содержащая записи для локальных и глобальных переменных; секция присутствует лишь в том случае, если компилятор вызывался с соответствующей опцией;

• .line — необходимая для отладчика таблица соответствия номеров строк исходного кода на C и машинных инструкций в секции .text;

• .strtab — таблица строк для таблиц символов в секциях .symtab и .debug.

Способ представления объектного кода в виде записей основан на представлении файла в виде последовательности записей разных типов. Разные записи могут иметь разную длину. Каждая запись имеет фиксированную структуру, соответствующую типу. Первый байт записи определяет тип этой записи, далее следует длина записи, затем информационное поле, размер которого зависит от его содержимого. В конце записи, как правило, указывается контрольная сумма.

Рассмотрим некоторые типы записей.

Заголовочная запись модуля содержит имя модуля и всегда идет первой в модуле. Кроме этого модуль может представлять собой главную программу (main) с указанием стартового адреса. При сборке различных модулей можно указать только один модуль, имеющий атрибут main. Если таковых будет несколько, то главным будет считаться первый. Таким образом, модули могут или не могут быть главными и могут иметь или не могут иметь стартовый адрес.

Запись определения сегмента. Модуль представляет собой совокупность объектного кода, описываемую последовательностью записей, создаваемых транслятором. Объектный код представляет собой непрерывные участки памяти, содержимое которых определяется во время трансляции. Этими участками являются логические

сегменты. Частным случаем сегмента является сегмент кода. Модуль определяет атрибуты каждого логического сегмента. Логический сегмент (ЛСЕГ) — это непрерывный участок памяти, чье содержимое определяется во время трансляции. Запись определения сегмента (SEGDEF) содержит всю информацию по ЛСЕГ (имя, длина, выравнивание и т.п.). Сборщик запрашивает эту информацию, когда комбинирует различные ЛСЕГ и устанавливает сегментную адресацию. SEGDEF всегда следует за заголовочной записью.

Определение имен. Определение имен осуществляется с помощью трех типов записей - PUBDEF (записи определения имен *public*), COMDEF (инициализированные переменные) и EXTDEF (список имен *external* и описание для каждого имени типа объекта, представляемого этим внешним именем).

Записи нумерации строк LINNUM позволяет транслятору соотносить номер строки исходного кода с соответствующей строкой транслированного кода.

Конечная запись модуля MODEND предназначена для указания сборщику конца модуля, а также для сообщения ему, содержит ли данный модуль стартовый адрес всей собираемой программы

Записи комментариев позволяет включать в объектный текст необходимые комментарии. Запись содержит флаги NP и NL, определяющие использование комментариев. Если $NP = 1$, то комментарии не могут быть удалены из объектного файла. Если $NL = 1$, то комментарии не должны появляться в листинге (распечатке) объектного файла.

4.4 Контрольные вопросы к разделу

1. Поясните назначение объектного кода.
2. Какие данные хранятся в объектном коде?
3. В каких форматах может храниться объектный код?
4. Зачем в объектном коде имеются секции имен?
5. Поясните работу редактора связей.
6. Предложите алгоритм реализации редактора связей.
7. Как выделяется память для локальных данных функции?
8. Как генерируется ассемблерный код для вызова функции?
9. Поясните назначение пролога и эпилога функции.
10. Проанализируйте ассемблерный код, генерируемый компилятором для сложных выражений в операторе if. Как генерируются переходы?
11. Проанализируйте ассемблерный код, генерируемый компилятором для оператора *switch*.
12. Проанализируйте ассемблерный код, генерируемый компилятором для оператора *while*.
13. Проанализируйте ассемблерный код, генерируемый компилятором для простого оператора for, в котором параметр цикла изменяется с постоянным шагом.
14. Проанализируйте ассемблерный код, генерируемый компилятором для адресации глобальных данных.
15. Проанализируйте ассемблерный код, генерируемый компилятором для локальных переменных.
16. Как осуществляется адресация параметров и локальных данных функции относительно регитсра *ebp*?
17. Как можно обнаружить переполнение стека?

18. Как генерируются команды работы с многомерными массивами?

19. Как генерируется ассемблереый код для арифметических операций над данными разных типов?

20. Какие проблемы могут возникнуть при трансляции сложнх выражений?

4.5 Упражнения к разделу

Цель данного задания – написать программу генерации программы на языке ассемблера по промежуточному коду. При выполнении задания следует обратить внимание на следующие моменты:

1. Глобальные данные должны быть описаны в сегменте данных. Тело каждой функции должно содержать команды выделения памяти для локальных данных и освобождения памяти перед выполнением команды ret.

2. Необходимо выбрать способ именования данных в командах ассемблера: явно по имени в соответствии с псевдооператорами описания данных или использовать косвенную адресацию.

3. При трансляции выражений используются регистры в соответствии с типами операндов, участвующих в операции. Операции с плавающей точкой выполняются сопроцессором.

4. Если транслируется сложное выражение, для вычисления которого не хватает имеющихся регистров, Вы должны предусмотреть команды сохранения промежуточных значений в памяти, а затем команды извлечения этих значений для дальнейшего использования.

5. Если транслируются переходы, то необходимо выбрать способ генерации меток.

Глава 5

АВТОМАТИЗАЦИЯ ПРОЕКТИРОВАНИЯ ТРАНСЛЯТОРОВ

Табличные принципы разработки алгоритмов лексических и синтаксических анализаторов позволяют рассмотреть как вопросы автоматизации построения таких таблиц, так и программирования самих анализаторов. В настоящее время существует множество систем автоматизации проектирования языковых процессоров.

Рассмотрим простейшие примеры таких систем. Операционная система UNIX предоставляет инструментальное средство для лексического анализа — *Lex* и синтаксического анализа *Yacc* входного потока (Yet Another Compiler Compiler — еще один компилятор компиляторов). Лексический анализатор, полученный с помощью *Lex*, работает с помощью управляющих таблиц, синтаксический анализатор реализует нисходящий разбор.

Сначала рассмотрим некоторые особенности языка описания лексики. Во–первых, кроме множества значимых лексем в анализируемом исходном тексте почти всегда имеются участки, которые нужно исключить из дальнейшей обработки. Примером игнорируемых цепочек могут служить комментарии в программах и символы–разделители лексем: пробелы, знаки табуляции, перевод строки и тому подобное. Для каждого игнорируемого типа цепочек в языке нужно написать специальный оператор, который помечает именованное регулярное выражение как игнорируемое.

Во-вторых, в процессе лексического анализа часто возникают исключительные ситуации, на которые сканер обязан отреагировать (неизвестный символ, ошибка в лексеме, критическая длина лексемы, конец исходного модуля).

В-третьих, язык описания лексики должен поддерживать возможность задания ключевых слов. Простния лексики аейший вариант такого задания — перечисление списка ключевых слов.

Очевидно, что языковый процессор — это не только лексический и синтаксический анализатор, хотя эти две программы составляют ядро языкового процессора. Наряду с лексикой и синтаксисом необходима реализация семантического уровня языка программирования. Таким образом, система автоматизации построения трансляторов должна получать на вход описание не только лексических и синтаксческих конструкций, но и правила синтаксически управляемого первода.

5.1 Лексический анализатор *Lex*

Рассмотрим сначала в качестве примера инструментальное средство для лексического анализа текста *Lex*, которе предоставляет ОС UNIX. Лексический анализ в *Lex* — это процесс извлечения слов из текста и их последующий анализ. Слово — это строка, которая соответствует регулярному выражению. *Lex* — это генератор программы лексического анализатора на языке Си. Процесс лексического анализа управляется с помощью таблицы, а функция *yylex*() осуществляет собственно анализ. Рассмотрим характерные свойства этой программы.

Файл спецификаций *Lex*

В качестве исходных данных для генерации *Lex* использует файл спецификации. Спецификации разделяются на три блока, и граница между ними состоит из строки с двумя символами процента (%%) в начале. Первый раздел — определения. Второй — раздел правил *Lex*, и заключительный третий — раздел подпрограмм пользователя. Простейший пример спецификаций программы, выделяющей слова:

```
word [A-Za-z][-A-Za-z']*
eol \n
%%
{word}    { printf("%s\n", yytext); }
{eol}     {}
.         {}
%%
```

Определения *Lex*

Определением является имя, сопровождаемое образцом для замены. Все имена должны начинаться в первом столбце, поскольку любые строки с пробелом в нем обрабатываются как строки исходного кода Си и без изменений записываются в выходной файл. Это свойство передачи исходных кодов позволяет объявлять переменные, препроцессорные директивы Си и так далее. Кроме того, можно сгруппировать весь исходный код Си в ограничители

```
%{... %}.
```

Тогда весь код, расположенный между символами, будет без изменений записан в выходной файл. Эти определения не обязательны для работы *Lex*, но весьма полезны. Если один и тот же образец замены используется в различных правилах, то его не нужно повторять. Аналогично, если существует необходимость изменить определение, нужно откорректировать только один фрагмент, а не несколько. Например, можно определить как

```
DIGIT 0-9
```

Затем можно поставить в соответствие этому типу числа:

```
{DIGIT}+            /* целое число */
{DIGIT}+.{DIGIT}*   /* вещественное число */
```

Допускается указывать объявления *Lex* в разделе определений. Они включают спецификации размеров таблиц, состояний и определений для текстового указателя.

Правила *Lex*

Правила *Lex* являются его сутью. Правила — это расширенные регулярные выражения и действия. Каждое регулярное выражение имеет соответствующее действие, и этому действию соответствует код Си. Действие может быть довольно сложным и заключаться в фигурные скобки. С помощью вертикальной полосы можно разбить большое регулярное выражение на несколько строк.

При определении соответствия некоторому действию происходит несколько операций. Сначала переменной yytext присваивается адрес строки, которая соответствует регулярному выражению и завершается *null* – указателем. Затем переменной yyleng присваивается длина строки. В заключение выполняются все необходимые действия.

В *Lex* определены четыре специальных действия:

1) действие "|" (использовать действие для следующего правила этого регулярного выражения);

2) "ECHO;" — вывести значение yytext в поток стандартного вывода;

3) "REJECT;" — продолжить до следующего выражения, соответствующего текущему вводу;

4) "BEGIN;" — переключить соответствующую определенному состоянию переменную (см. п.2.2.1.4).

Lex пытается найти соответствие самой длинной строке, которая соответствует определенному регулярному выражению. Если найдено соответствие двум регулярным выражениям одной длины, то используется первое определенное выражение. REJECT - заставить искать соответствие для второго и последующих регулярных выражений.

Подпрограммы *Lex*

В разделе подпрограмм пользователя можно определять любые подпрограммы, включая подпрограмму *main*. Этот код без изменений записывается в файл *lex.yy.c* и компилируется обычным способом. Таким образом можно полностью написать программу внутри файла спецификаций *Lex*.

Функции и переменные *Lex*

Lex создает несколько функций, которые являются доступными программисту. Кроме переменных yytext и yyleng существует переменная yyin. Это дескриптор файла, который используется для операций ввода.

Первой рассмотрим функцию *yylex*. Она не имеет аргументов и возвращает целое число. По достижении конца файла возвращается ноль, иначе возвращается считанная лексема (для использования в синтаксическом анализаторе, например в *Yacc*). Обращение к этой функции осуществляется для вызова лексического анализатора.

Функция yymore также не имеет аргументов и возвращает целое число. При вызове этой функции *Lex* конкатенирует следующее регулярное выражение к *yytext*.

Функция yyless принимает в качестве аргумента целое число и возвращает целое число. При ее вызове *Lex* сохраняет оканчивающуюся null–указателем строку из определенного количества символов. Эти символы игнорируются при анализе.

Функция ввода не имеет аргументов и возвращает целое число. Эта подпрограмма возвращает следующий символ из потока ввода. Символ удаляется из входного потока. Функция unput принимает в качестве аргумента целое число и возвращает определенный символ в начало входного потока. Функция yywrap не имеет аргументов и возвращает значение 1, чтобы сообщить *Lex* об окончании ввода (обычно это конец файла).

Объявления таблиц *Lex*

Поскольку *Lex* управляет процессом лексического анализа с помощью таблиц, на размер грамматики наложены некоторые ограничения. Размеры таблиц можно изменить в разделе определений спецификаций *Lex*:

1) %a (минимум 2000) - количество переходов *Lex*;
2) %e (минимум 1000) - количество вершин дерева синтаксического разбора;
3) %k (минимум 1000) - количество упакованных классов символов;
4) %n (минимум 500) - количество разрешенных состояний;
5) %o (минимум 3000) - размер вводимого массива;
6) %p (минимум 2500) - количество разрешенных позиций в *Lex*;

Можно также объявить, каким образом будет сохраняться значение *yytext*. Если в определениях находится %*array*, то *yytext* — символьный массив. Если присутствует %pointer, yytext является указателем на символ. В обоих случаях, yytext всегда оканчивается null–указателем.

Состояния *Lex*

Иногда существует необходимость отслеживать состояние ввода и, в зависимости от состояния, выполнять различные типы замен или действий. Хорошим примером является комментарий в Си — если считанные лексемы принадлежат комментарию, то они игнорируются. Поскольку комментарии могут распространяться на несколько строк, необходимо изменять состояние.

Состояние определяется с помощью ключевых символов %s. При определении зависимых от состояния правил необходимо указывать в угловых скобках определенное состояние перед регулярным выражением. При объявлении состояния, по умолчанию оно рассматривается неактивным. Например:

```
%s COMMENT
%%
<COMMENT>.      {}
<COMMENT>"/*"   {}
<COMMENT>"*/"   { BEGIN INITIAL; }
"/*"            { BEGIN COMMENT; }
```

Если нужно начать обработку в режиме COMMENT, следует написать в разделе определений следующие строки:

```
%{
     BEGIN COMMENT;
%}
```

5.2 Система *ANTLR*

ANTLR (Another Tool For Language Recognition - http://www.antlr.org) — генератор синтаксических анализаторов, позволяющий по описанию $LL(k)$–грамматики автоматически создавать программу–парсер (синтаксический и и лексический анализатор). ANTLR позволяет конструировать компиляторы, интерпретаторы и трансляторы с различных формальных языков.

ANTLR, как и многие другие подобные средства, состоит из библиотеки классов, облегчающих основные операции при разборе (буферизация, поиск и т. д.), и утилиты, генерирующей код парсера на основе файла, описывающего грамматику разбираемого языка. Сам ANTLR написан на Java, но позволяет генерировать код на Java

и C++. Кроме того, ANTLR отличается от других подобных программ наличием визуальной среды разработки ANTLRWorks, позволяющей создавать и отлаживать грамматики: это многооконный редактор, поддерживающий подсветку синтаксиса, визуальное отображение грамматик в виде синтаксических диаграмм, отладчик, инструменты для рефакторинга.

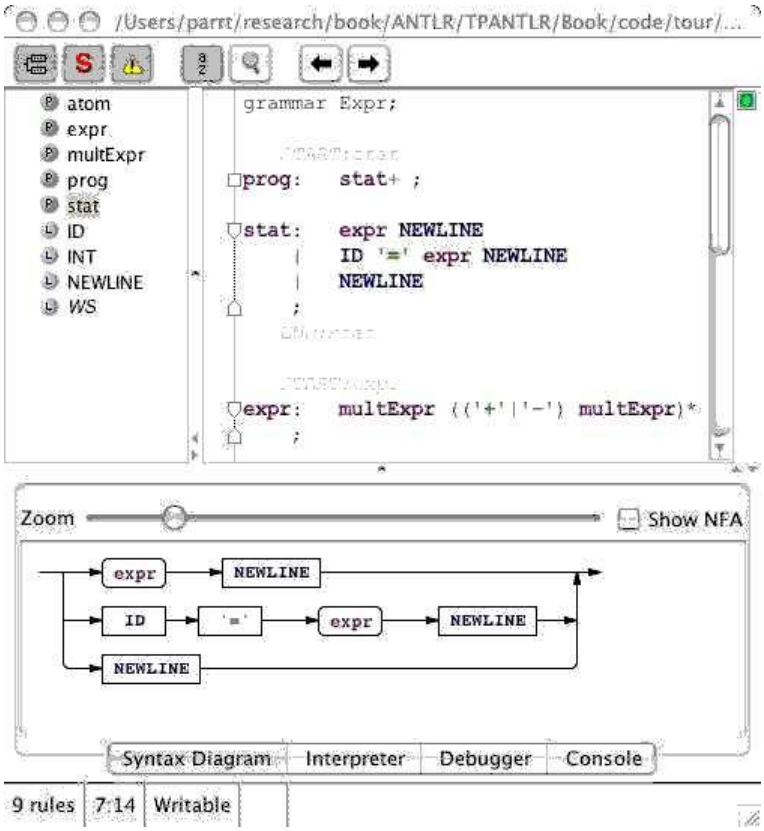


Рис. 5.1: ANTLRWorks

Результатом работы ANTLR являются два класса: лексер (лексический анализатор) и парсер (синтаксический анализатор). Лексер разбивает поток символов на поток лексем в соответствии с правилами, а парсер обрабатывает поток лексем в соответствии с другими правилами. Правила пишутся в грамматике на специальном языке, основанном на регулярных выражениях.

Заголовочная секция

Заголовочная секция является первой секцией в грамматическом файле и содержит исходный код, который должен быть скопирован перед сгенерированной программой синтаксического анализатора. Это очень полезная секция при генерации на C++, так как в программе на C++ необходимо, чтобы типы, объекты и функции были объявлены перед их использованием. При программировании на Java эта секция может содержать описание импортируемых классов. Заголовочная секция имеет вид:

```
header {
< исходный код на языке, на котором генерируется программа >
}
```

Секция лексического анализа

Код	Описание
(...)	подправило
(...)*	итерация подправила
(...)+	усеченная итерация подправила
(...)?	подправило, которое может отсутствовать
...	семантические действия
[...]	параметры правила
...?	семантический предикат
(...)=>	синтаксический предикат
|	оператор альтернативы
..	оператор диапазона
~	отрицание
.	любой символ
=	присвоение
:	метка, начало правила
;	конец правила
class	класс грамматики
extends	определяет базовый класс для грамматики
returns	описание возвращаемого значения для правила
options	секция опций
tokens	секция токенов (лексем)
header	заголовок

Рис. 5.2: Краткий перечень элементов языка описания грамматики

Чтобы выполнять лексический анализ, нужно определить класс *lexer*, описание которого имеет следующую структуру:

```
{ optional class code preamble }
class YourLexerClass extends Lexer;
options
tokens
{ optional action for instance vars/methods }
lexer rules...
```

Лексические правила, содержащиеся в классе *lexer*, становятся методами члена в сгенерированном классе. Каждая грамматика (файл *.g) может содержать только один класс *lexer*. Синтаксический анализатор и класс *lexer* могут появиться в описании грамматики в любом порядке.

Например, если класс *lexer* содержит только знаки равенства, идентификаторы и целые числа, а символы пробела, табуляции и перевода строки – игнорируемые символы, то описание класса типа *lexer* имеет вид:

```
class MyLexer extends Lexer;
// опции лексера.
options{
// диапазон возможных символов (0-127 ASCII).
```

```
    charVocabulary = ''..'177';
}

// идентификатор:
NAME: (('a'..'z')|('A'..'Z'))
      (('a'..'z')|('A'..'Z')|('0'..'9'))*
;

// целое число:
NUM:    ('0'..'9')+;

//  разделители (пробелы и символы перевода строки):
SPACES:  (    ' '|'\n'|"\t"    )+
      // Не разбирать этот тип токенов в парсере
      {$setType(Token.SKIP);}
;

// знак равенства:
EQU : "=";
```

Секция синтаксического анализатора

Все правила синтаксического анализатора должны быть связаны с классом синтаксического анализатора *parser*. Грамматика (файл *.g) может содержать только один класс синтаксического анализатора вместе с единственным классам *lexer*. Спецификация класса синтаксического анализатора имеет вид:

```
{ optional class code preamble }
class YourParserClass extends Parser;
options
tokens
{ optional action for instance vars/methods }
parser rules...
```

Например, если текст состоит из последовательности идентификаторов, причем за каждым идентификатором после знака равенства следует одно число, то описание синтаксиса имеет вид:

```
class MyParser extends Parser;

// текст содержит хотя бы одну конструкцию и заканчивается словом "end"
mainRule:
    (element)+
    "end"
;

element:
     NAME EQU NUM
;
```

Синтаксические предикаты

ANTLR реализует $ll(k)$ – анализатор. Очевидно, что при $k > 1$ увеличивается сложность синтаксического анализатора, поэтому увеличивать k не имеет смысла. Однако, в некоторых случаях вместо преобразования КС–грамматики к виду $LL(1)$ можно определить условия, при которых следут применять то или иное правило из числа тех, у которых совпадают значения функции $first_1(...)$. Для этого можно использовать синтаксические предикаты. Для каждого синтаксического предиката ANTLR генерирует специальный метод, возвращающий true или false в зависимости от того, имеет очередной разбираемый фрагмент требуемую структуру или нет. Синтаксический предикат позволяет проанализировать префикс произвольной длины и записывается в виде:

$$(predictionblock) => production$$

Например, цепочкам $((a))$; и (a); будут соответствовать выводы

$$Rule \to elem; \to (elem); \to ((ID)); \to ((a)); Rule \to cast; \to (ID); \to (a);$$

в грамматике с синтаксическими предикатами:

```
Rule:
       (cast ';') => cast ';'
   |   (elem ';') => elem ';'
   |                 elem '.'
;

cast: '(' ID ')' ;

elem:    '(' elem ')'
      |  ID
;

ID : 'a'..'z' + ;
```

Приведенные примеры показывают, что синтаксические предикаты позволяют выполнить анализ префикса произвольной длины, поэтому для реализации эффективных вычислений использовать синтаксические предикаты следует только при невозможности разрешения конфликтов в управляющей таблице $LL(1)$–анализатора другим путем.

Семантические предикаты

Семантические предикаты так же, как и синтаксические предикаты, представляют собой логические выражения, определяющие порядок применения альтернативных правил в процессе синтаксического анализа. Однако, если синтаксические предикаты позволяют рассмотреть синтаксическую структуру цепочки, то семантические предикаты предназначены для выбора альтернативы по семантическим признакам. Пусть, например, оператором может служить присваивание и вызов функции:

$$S \to a = V; \mid a(P);$$

Тогда выбор одного из этих двух праил может осуществляться по семантическому типу идентификатора a: если это имя функции, то применяется второе правило. Структура правила с семантическим предикатом имеет вид:

$$\{expression\}?\ production$$

Выражения в семантическом предикате не должны иметь побочных эффектов и должны возвращать логическое значение (boolean в Java или bool в C++). Например,

```
oper
     : {isMethod(input.LT(1))}?    ID '(' expr (',' expr)* ')' )
     | ID '=' expr ';'
;
```

Здесь функция пользователя $isMethod(lex)$ проверяет, является ли ее параметр идентификатором функции.

Для ознакомления с ANTLR полезно использовать следующие ссылки:

http://www.antlr.org/ - официальный сайт системы ANTLR

http://club.shelek.ru/viewart.php?id=39 - Написание парсеров с помощью ANTLR

http://habrahabr.ru/blogs/programming/110710/ - Грамматика арифметики или пишем калькулятор на ANTLR

5.3 Контрольные вопросы к разделу

1. Какие минимальные требования должен обеспечивать язык описания лексики для системы автоматизации программирования?
2. Поясните назначение Lex.
3. Запишите правила определения вещественного числа в синтаксисе Lex.
4. Для какой цели используются состояния в Lex?
5. Поясните назначение системы ANTLR.
6. Перечислите возможности системы ANTLR.
7. Какую конструкцию имеет секция лексического анализа ANTLR?
8. Как указать игнорируемые символы в ANTLR?
9. Какие классы необходимо описать в ANTLR при генерации лексического и синтаксического анализаторов?
10. Зачем используются синтаксические предикаты в ANTLR?
11. Поясните назначение семантических предикатов в ANTLR.

5.4 Упражнения к разделу

5.4.1 Задание

Цель данного задания – использовать программу ALTLR для генерации транслятора. При изучении предшестующих тем Вы уже написали синтаксически управляемый перевод и реализовали его. Теперь Вы должны написать этот перевод в терминах языка системы ANTLR. Работа должна быть организована следующим образом.

1. Перепишите КС–грамматику Вашего языка программирования. Проанализируйте правила этой грамматики с целью обнаружения несовместимости с требованиями, предъявляемыми системой ANTLR.

2. Опишите лексику языка.

3. Опишите синтаксис языка.

4. Добавьте в описание правила синтаксически управляемого перевода.

4. С помощью ANTL сгенерируйте программу транслятора.

5. Проверьте совпадение результатов трансляции, построенных Вашей программой из предшествующей лабораторной работы и построенных сгенерированной ANTL программой.

6. Проверьте работу сгенерированной программы на правильных и ошибочных текстах программ.

5.4.2 Пример выполнения задания

ANTL поддерживает секцию *tokens*, предназначенную для описания лексических единиц так, как мы делали это при разработке программы сканера. Поэтому первая секция — это секция *tokens*, затем укажем подключаемый тод функции *main*, далее запишем КС–грамматику транслируемого языка.

```
grammar my;

options
{
        backtrack = true;
        k = 3;
        language = C;
}

tokens{
      TMain  = 'main';
      TInt   = 'int';
      TFloat = 'float';
      ...
}

@members
{
     #include "myLexer.h"
     int main(int argc, char * argv[])
     {
      ////////////////////
     z = 0;
     p_triad = 0;
     p_operac = 0;
     p_operand = 0;
      p_obl_vid = 0;
      p_if = 0;
      Root = new Tree();
```

```
      Root->Cur = Root;
      triadaF = fopen("Triada.txt", "w");
      Root->errorF = fopen("Error.txt", "w");
     ////////////////////

        pANTLR3_INPUT_STREAM           input;
        pmyLexer                lex;
        pANTLR3_COMMON_TOKEN_STREAM    tokens;
        pmyParser              parser;

        input  = antlr3AsciiFileStreamNew          ((pANTLR3_UINT8)"input.txt");
        lex    = myLexerNew                (input);
        tokens = antlr3CommonTokenStreamSourceNew  (ANTLR3_SIZE_HINT, TOKENSOURCE(l
        parser = myParserNew               (tokens);

        parser  ->prg(parser);

PrintMagTriad();
        parser ->free(parser);
        tokens ->free(tokens);
        lex    ->free(lex);
        input  ->close(input);

        system("pause");
        return 0;
     }
}

  //Сначала создадим  описание грамматики языка:


public s:  (TClass TMain '{' n '}')? ;
n : (d n | q n | f u)? ;
u : (d u | q u)? ;
q : TClass ID '{' n '}'| ID ID ';' ;
d : k | e ;
k : t v ';' ;
e : TFinal t c ';' ;
c : ID '=' CONST z ;
z : (',' ID '=' CONST z)? ;
v : ID v1 ;
v1 : '=' a1 j | j ;
j : (',' ID j1 j)? ;
j1 : ( '=' a1 )? ;
f : TVoid TMain '(' ')' x ;
x : '{' m '}' ;
m : (o m | d m)? ;
o : x | w | TReturn ';' |';' | ID o2 ';' | ID ID ';' ;
o2 : y a1 | '.' ID k1 y a1 ;
w : TFor '(' ID o2 ';' a1 ';' ID o2 ')' o ;
```

```
a1 : '+' a2 | '-' a2 | a2;
a2 : a3 u1 ;
u1 : (('+' | '-' ) a3 u1)? ;
a3 : a4 u2 ;
u2 : (('*' | '/' | '%' ) a4 u2)? ;
a4 : '(' a1 ')' | ID k1 | CONST ;
k1 : ( '.' ID k1)? ;

t : TInt | TFloat ;

y : '+='{ExtObj.df.SaveOperation("+=",Defs.Tpluseq);} |
'-='{ ExtObj.df.SaveOperation("-=",Defs.Tminuseq);} |
'*='{ ExtObj.df.SaveOperation("*=",Defs.Tmuleq);} |
'/='{ ExtObj.df.SaveOperation("/=",Defs.Tdiveq);} |
'%='{ ExtObj.df.SaveOperation("\%=",Defs.Tmodeq);} |
'='{ ExtObj.df.SaveOperation("=",Defs.Teq);} ;

ID : LETTER SECONDPART;
CONST : VAL_DEC ;

COMMENT : '/*' .* '*/' {$channel=Hidden;};
LINE_COMMENT
  : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=Hidden;};

fragment SECONDPART : (SYMBOLS)* ;
fragment SYMBOLS : LETTER | DIGIT ;
fragment LETTER : 'a'..'z'|'A'..'Z' | '_' ;
fragment DIGIT : '0'..'9' ;
fragment VAL_DEC : (DIGIT)+ ;
```

После того, как мы убедились, что синтаксис работает правильно, подключим семантические подпрограммы. Фактически, это вставка вызовов семантических функций в соответствии с правилами синтаксически управляемого перевода. Вызовы семантических подпрограмм указываются в фигурных скобках. Не будем приводить здесь весь текст, достаточно привести пример одного правила для составного оператора и для выражения второго уровня приоритетов:

```
x :      '{' {DeltaGenLevel();}
               m '}'{DeltaRestoreRet();} ;

a2 :       a3 u1 ;
u1 :       ((
               {DeltaSaveOperation($text);}'+'
               |
               {DeltaSaveOperation($text);} '-'
               ) a3 {DeltaFormTriad();} u1)? ;
```

Глава 6

ОБЩИЕ МЕТОДЫ НЕЙТРАЛИЗАЦИИ ОШИБОК

6.1 Нейтрализация и исправление ошибок

Программа, поступающая на вход транслятора, часто не принадлежит распознаваемому языку. Такая программа называется неправильной. В соответствии со структурой языка программирования (а, следовательно, и со структурой компилятора) ошибки можно разделить на лексические, синтаксические и семантические. Вопрос о том, сколько и какие сообщения об ошибках для данной неправильной цепочки выдавать, зависит от разработчика и часто является спорным. На самом деле различные компиляторы для одного и того же языка программирования часто выдают разные сообщения об ошибках для одной и той же неправильной цепочки. Когда компилятор обнаруживает первую ошибку, то он может либо прекратить работу, либо попытаться продолжить анализ. В первом случае никаких особых проблем не возникает, и мы во всех предшествующих главах занимались проектированием именно таких компиляторов. Многие компиляторы после обнаружения первой ошибки пытаются продолжить работу. Рассмотрим возникающие при этом проблемы.

Существуют два способа, которыми компилятор пытается справиться с ошибкой. *Под нейтрализацией ошибки* понимается алгоритм продолжения анализа исходного модуля после обнаружения ошибки. *При исправлении ошибки* должна получиться действительно правильная программа. Очевидно, что исправить можно только очень незначительное число ошибок, как правило, связанных с пропусками лексем, однозначно определяемых по контексту (например, отсутствие открывающейся скобки после if в программе на языке Си или соответствующей закрывающейся скобки). Можно исправить некоторые орфографические ошибки. Такое исправление легко осуществляется, когда в процессе синтаксического анализа известно, что очередной символ должен быть словом из некоторого набора ключевых слов. Если вместо него оказался идентификатор, надо проверить, не служебное ли это слово, искаженное орфографической ошибкой. Такая ситуация возникает при анализе языков программирования, где почти любой оператор начинается с ключевого слова.

Нередко из-за ошибки в написании некоторый идентификатор встречается однократно и он либо не описан, либо описан, но не используется (а в языках с умолчанием такому идентификатору либо не присваивается значение, либо ему только присвоено значение, а далее он нигде не используется). Такие идентификаторы становятся кандидатами на исправление ошибки, причем режим исправления управляется пользователем и выполняется либо для каждого идентификатора по запросу, либо

устанавливается опция исправления для всех идентификаторов.

При исправлении орфографических ошибок учитывается тот факт, что 80% всех ошибок попадают в следующие четыре класса:

1) пропущена одна буква;

2) неверно написана одна буква;

3) вставлена одна буква;

4) два соседних символа переставлены.

К сожалению, большинство ошибок исправить нельзя, их можно только нейтрализовать с большим или меньшим успехом. Если в компилятор встроен блок нейтрализации ошибок, то он должен выявлять по возможности максимальное их число с минимальным числом наведенных ошибок. Наведенные ошибки появляются в результате неверно проведенной нейтрализации ошибки.

Нейтрализация для различных методов анализа определяется тем способом, который положен в основу построения анализатора, а, точнее, той памятью, которая отображает текущее состояние процесса анализа. Для метода рекурсивного спуска такой памятью является стек возвратов программы анализатора, следовательно, нейтрализация для метода синтаксических диаграмм должна быть основана на умении так выйти из последовательности рекурсивных вызовов, чтобы все дальнейшие вызовы и возвраты осуществлялись по возможности корректно. Для магазинных методов, работающих по нисходящей и восходящей стратегии, алгоритмы нейтрализации должны быть основаны на внесении изменений в верхушку магазина и, следовательно, алгоритмы нейтрализации определяются стратегией разбора и не зависят от конкретного алгоритма разбора по выбранной стратегии.

При возникновении любой ошибки возможны два способа нейтрализации этой ошибки:

a) исключить ошибочную лексему;

b) вставить цепочку лексем.

Конечно, вставка и исключение не выполняются физически на уровне исходного модуля. Все изменения при нейтрализации выполняются либо в магазине — вставки элементов или их пропуск, либо в процессе дополнительного сканирования исходного модуля — пропуск текста.

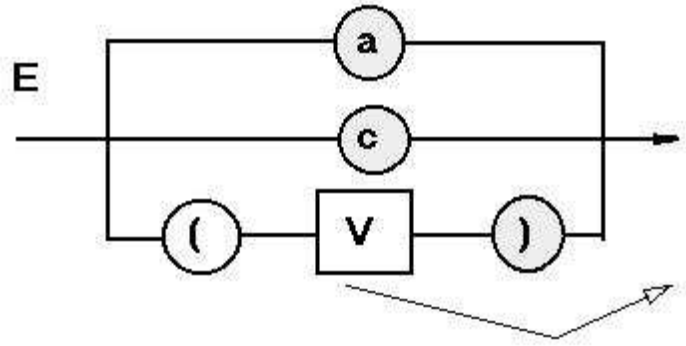
После нейтрализации ошибки алгоритм синтаксического анализа продолжает работу в надежде, что восстановленная конфигурация является "правильной" в том смысле, что она соответствует конфигурации, которая имела бы место, если бы программист не сделал первой ошибки. Поэтому, если после нейтрализации ошибки будут получены еще какие–то сообщения, то они соответствуют уже другим ошибкам, о которых программисту тоже хотелось бы знать. Риск состоит в том, что конфигурация будет восстановлена неверно и компилятор может в дальнейшем сообщать о фиктивных (наведенных) ошибках, которых на самом деле в программе нет. Любая попытка нейтрализации ошибок представляет собой компромисс между желанием обнаружить как можно больше ошибок, и желанием избежать сообщений о несуществующих ошибках.

6.2 Нейтрализация ошибок при рекурсивном спуске

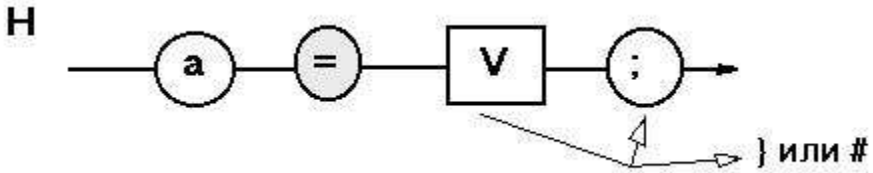
Если транслятор работает до первой ошибки, то при программировании его методом синтаксических диаграмм завершение работы при обнаружении ошибки реализуется через функцию $PrintError()$, в конце которой ставится оператор $exit$. Если

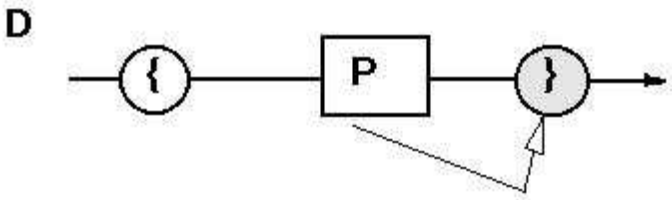
транслятор выявляет все ошибки, то предусматривают специальные методы нейтрализации ошибки. Они основаны на двух методах:

а) вставке символа, например, можно вставить "a", "c" или ")" в диаграмме



б) игнорировании символа или последовательности символов до тех пор, пока не встретится конструкция, допускающая анализ; например, при ошибке в каком-либо операторе языка Си можно проигнорировать текст до знаков ";" или "}".





Сразу следует отметить, что вставка символов — операция весьма опасная с точки зрения возможного появления так называемых *наведенных ошибок*. Наведенные ошибки — это такие ошибки, которые отсутствуют в исходном модуле, но транслятор выдает сообщение об ошибке из–за исправлений, которые он выполнил в предшествующем тексте программы. Вообще говоря, не существует ни одного алгоритма нейтрализации ошибок, свободного от наведенных ошибок. Точнее, если транслятор достаточно полно анализирует текст, не пропуская чересчур большие фрагменты этого текста, для любого алгоритма нейтрализации можно предложить пример программы, при трансляции которой появятся наведенные ошибки. Поэтому предлагаемые Вами алгоритмы нейтрализации должны выдавать минимум наведенных ошибок при максиммальном обработанном тексте. С этой точки зрения вставка символов "a" или "c" вряд ли разумна, т.к. выражение будет анализироваться дальше и появятся наведенные ошибки. В отличие от вставки символов "a" или "c" вставка закрывающейся скобки весьма продуктивна: во–первых, для каждой пропущенной скобки будет выдано сообщение об ошибке, во–вторых, несоответствие скобок — весьма часто встречаемая на практике ошибка — поэтому велика вероятность, что далее следует правильный текст.

Для того, чтобы сигнализировать о наличии ошибки или об исправлении ошибки, вводятся два флага ошибки:

```
int FlagError;       // имеется еще не нейтрализованная ошибка
int FlagErrorProg;   //была обнаружена ошибка в исходном модуле
```

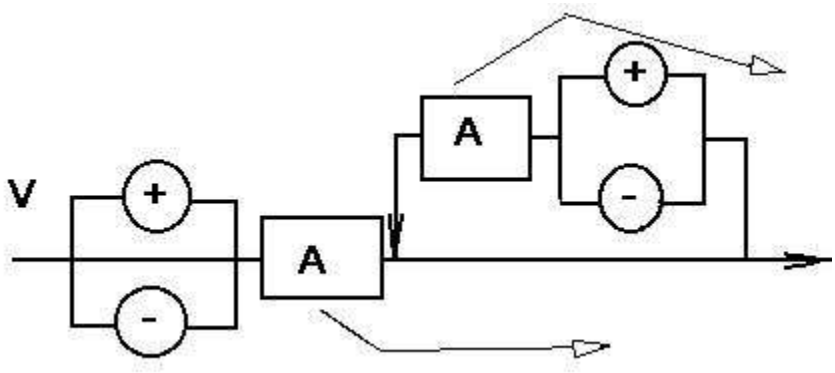
$FlagErrorProg$ — признак того, что когда–то была обнаружена хотя бы одна ошибка. Этот флаг всегда устанавливается в $PrintError()$ и никогда не сбрасывается. Он используется для отказа от генерации объектного кода. $FlagError$ используется для того, чтобы показать, что ошибка обнаружена и еще не нейтрализована. Этот флаг устанавливается в $PrintError()$ и сбрасывается при нейтрализации ошибки в программах, соответствующих синтаксическим диаграммам.

Чтобы выполнить практическую нейтрализацию ошибки, надо предварительно разделить все синтаксические диаграммы на два типа:

1) синтаксические диаграммы, в которых нейтрализуется ошибка методом вставки или пропуска;

2) синтаксические диаграммы, в которых ошибку нейтрализовать нельзя.

В синтаксических диаграммах второго типа при обнаружении ошибки осуществляется выход из текущей процедуры без изменения флага. Например, если в одном из выражений, связанных аддитивным знаком операции, обнаружена ошибка, то нужно прервать дальнейший анализ данного выражения:



В синтаксической диаграмме первого типа выполняется нейтрализация пропуском или вставкой символа и сбрасывается флаг $FlagError$. Обычно вставляются символы, наличие которых обязательно в тексте исходного модуля: закрывающие скобки, специальные ключевые слова в середине оператора, ограничители операторов и т.п. Пропуск текста исходного модуля выполняется по возможности минимальной длины, на логически законченном фрагменте — чаще всего до символа конца оператора (”end”, ”}” и т.п).

Фактически, пропуск можно организовать до таких символов, которые можно считать *”надежными символами”*. В программах можно выделить три типа надежных символов:

1) синхронизирующие символы,

2) начинающие символы,

3) завершающие символы.

Синхронизирующие символы — это такие символы, относительно которого разработчик по разумным причинам уверен, что он знает, как возобновить процесс обработки. Множество синхронизирующих символов для данного языка может включать

некоторые знаки пунктуации, такие, как запятая, некоторые зарезервированные слова, например, *else* или *then*. В ходе нейтрализации, когда найден один синхронизирующий символ, из магазина выталкиваются символы до тех пор, пока не будет найден магазинный символ, к которому будет применен данный синхронизирующий символ для правильного продолжения обработки. В качестве синхронизирующего символа в простых неструктурированных языках программирования часто используется знак окончания строки (например, в языках Ассемблер, Basic, Фортран). Метод синхронизирующих символов в сложных языках программирования применяется достаточно редко.

Начинающие символы — это такие символы, которые помогают разработчику установить начало некоторой конструкции. В большинстве языков программирования операторы, за исключением оператора присваивания, начинаются специальными ключевыми словами, которые и используются в качестве начинающих символов. К начинающим символам относится левая операторная скобка, например, *begin* в языке Паскаль, открывающая фигурная скобка в языке Си.

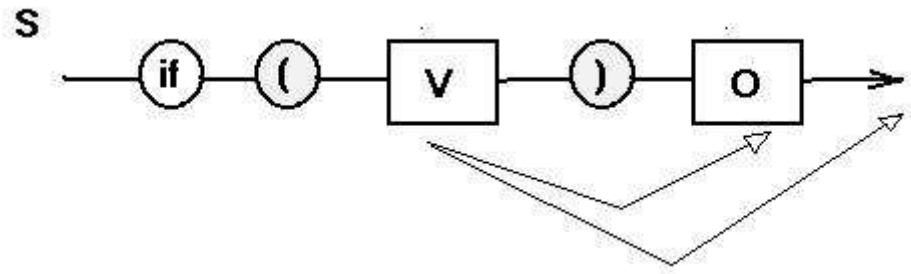
Завершающие символы означают завершение некоторой конструкции. Эти символы помогают успешно завершить обработку некоторой конструкции и перейти к обработке той конструкции, которая может стоять после только что обработанной. Как правило, такими символами являются правые операторные скобки: *end* в языке Паскаль, закрывающая фигурная скобка в языке Си. Завершающим символом является знак завершения оператора, например, знак точки с запятой во многих языках программирования.

Нет смысла исправлять несколько ошибок в одном мелком фрагменте программы, излишне перегружая тем самым функции синтаксических диаграмм. Например, при отсутствии операнда в выражении проще не вставлять операнд, как это указано в диаграмме, приведенной выше. Можно выдать сообщение об ошибке, выйти из функции с установленным флагом $FlagError$ и выполнить нейтрализацию в блоках верхнего уровня рекурсии.

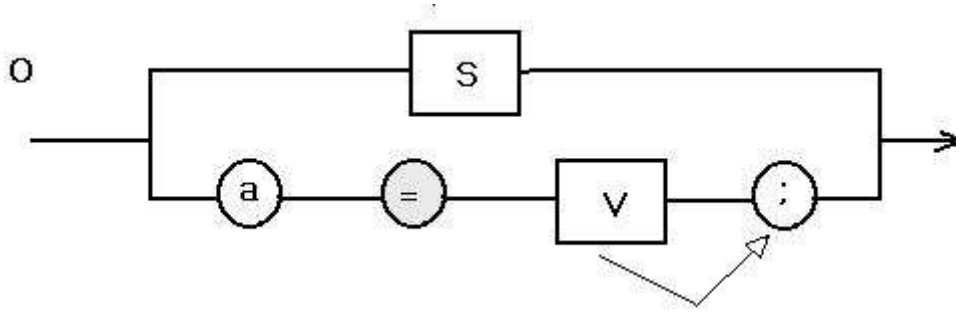
Пример 10.1. Рассмотрим нейтрализацию ошибок для грамматики

$$\begin{array}{ll} G: & S \to if(V)O \\ & O \to S|a = V; \\ & V \to V + A|V - A| + A| - A|A \\ & A \to A * T|A/T|T \\ & T \to a|c|(V) \end{array}$$

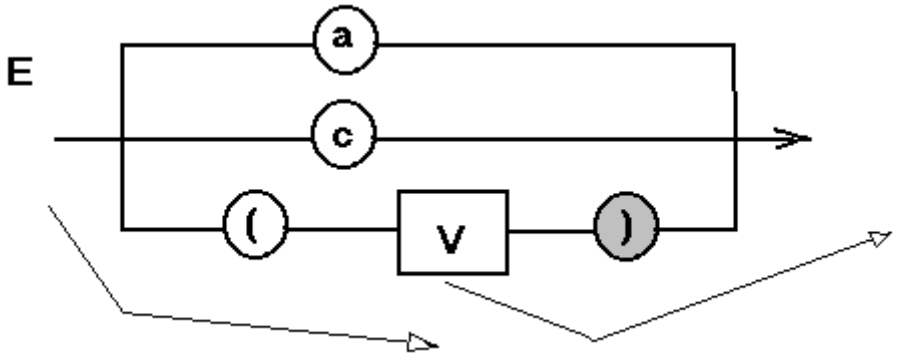
В синтаксической диаграммаме S выполняется вставка пропущенных скобок, а при обнаружении ошибки в V — пропуск до $first(O)$ или $follow(S)$:



В O выполняется вставка пропущенного знака "=", выход при неверном $first(O)$, пропуск до знака ";" при ошибке в V.



В диаграммах V и A нейтрализация не осуществляется и при обнаружении ошибки в вызываемой функции выполняется выход из процедуры, а при трансляции E выполняется вставка пропущенного знака ")", выход при неверном $first(E)$ и при ошибке в V.



6.3 Нейтрализация при нисходящем разборе

В отличие от метода синтаксических диаграмм алгоритм нейтрализации для всех магазинных методов не зависит от применяемого правила и реализуется в виде единственной подпрограммы, которая может быть вызвана из функции $PrintError()$.

Любой нисходящий анализатор, выполняющий подстановки по правилам грамматики, использует магазин для записи терминалов и нетерминалов, последовательность которых соответствует ожидаемой структуре исходного модуля. Несоответствие элементов магазина и сканируемых лексем приводит к регистрации ошибки. Символы в магазине являются висячими вершинами дерева разбора, которым к моменту регистрации ошибки не поставлены в соответствие лексемы входной цепочки. Как и раннее, при нейтрализации будем использовать понятие надежных символов. Тогда нейтрализацию возникшей ошибки можно выполнить на основе поиска таких надежных символов в анализируемом тексте программы, которым можно поставить в соответствие некоторый символ в магазине.

Допустим, в магазине имеется последовательность вершин $L = \{A_1, A_2, ..., A_k\}$, где A_1 — верхний символ магазина. Пусть остаток входной цепочки имеет вид

$$b_0 b_1 b_2 ... b_m,$$

где символ b_0 — отсканированная лексема, $b_1 b_2 ... b_m$ — не сканированный фрагмент. Ошибка появилась по одной из двух причин:

а) символ b_0 не принадлежит множеству $first_1(A_1)$;

б) $\varepsilon \in first_1(A_1)$ и символ b_0 не принадлежит множеству $follow_1(A_1)$.

Найдем ближайший к верхушке магазина символ A_i, для которого в остатке входной цепочки найдется такой символ b_j, который принадлежит множеству $first_1(A_i)$. Заменим цепочку $b_0 b_1 ... b_{j-1}$ на цепочку, выводимую из $A_1 A_2 ... A_{i-1}$. В результате такой замены ошибка перед символом b_0 будет нейтрализована.

Это не единственный вариант поиска соответствия надежного символа и символа в магащине. Часто возникает ситуация, когда надежный символ заканчивает некоторое правило грамматики, т.е. найденный надежный символ b_j принадлежит множеству $last_1(A_i)$ (такая ситуация, например, часто появляется для символа ";"). Но это означает, что мы нашли возможное завершение некоторой конструкции и анализ может быть продолжен с позиции после символа b_j для конструкции, которая лежит в магазине под символом A_i.

Реализация в программе нейтрализации указанных замен проста и заключается в стирании верхушки магазина до символа A_i (в первом случае) или вместе с ним (во втором случае). Что касается пропуска текста, то в процессе поиска надежного символа такое сканирование уже было выполнено.

Пример 10.2. Рассмотрим нейтрализацию ошибки в простейшем операторе присваивания:

$$\begin{aligned} G: \quad & O \to a = S; \\ & S \to S + A | S - A | A \\ & A \to A * B | A / B | B \\ & B \to c | a | (S) \end{aligned}$$

Пусть в исходном модуле допущена ошибка следующего вида: $a = a+);$. Тогда в дереве грамматического разбора висячими являются вершины множества $L = \{A, ;\}$, причем $first_1(A) = \{c, a, (\}$ и $first_1(;) = \{;\}$. Следовательно, установив соответствие знака ";" в L знаку ";" во входной цепочке, получим замену фрагмента ")" на самый короткий фрагмент, выводимый из A, например, на цепочку a. Иллюстрация приведена на рис. 6.1.

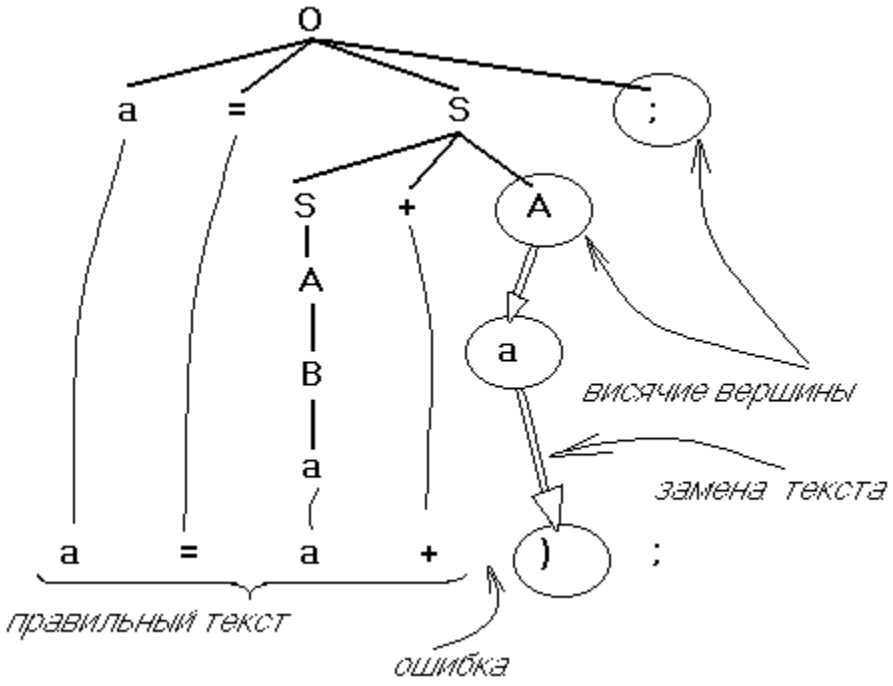


Рис. 6.1: Пример нейтрализации ошибки при нисходящем анализе

6.4 Нейтрализация при восходящем разборе

При восходящем разборе на каждом шаге выполняется либо запись отсканированного символа в магазин, либо редукция верхушки магазина по правым частям правил грамматики. Ошибка регистрируется либо при несовместимости соседних символов, либо при отсутствии правила, правая часть которого совпадает с выделенной в магазине основой. При восходящем анализе нейтрализация ошибки происходит сложнее, чем при нисходящем анализе, т.к. синтаксический анализатор имеет меньше информации о структуре полного дерева разбора. В этом случае можно просто попытаться привести фрагмент дерева разбора к такому виду, который согласуется с некоторой частью какого-нибудь правила грамматики.

Пусть между символами A и B обнаружена ошибка. Можно предложить следующий алгоритм нейтрализации ошибки:

1) если в грамматике имеется правило $D \to xAyBz$, то между символами A и B вставим терминальную цепочку, выводимую из y;

2) если в грамматике имеется правило $D \to xATz$ и $T \stackrel{*}{\Rightarrow} yB$ (или, что то же самое, $B \in last_1(T)$), то между символами A и B вставим терминальную цепочку, выводимую из y;

3) если в грамматике имеется правило $D \to xTBz$ и $T \stackrel{*}{\Rightarrow} Ay$ (или $A \in first_1(T)$), то между символами A и B вставим терминальную цепочку, выводимую из y;

4) если ни один из предыдущих пунктов не применим, удалим B.

Пример 10.3. Рассмотрим нейтрализацию ошибки в простейшем операторе присваивания примера 10.2. Частичное дерево разбора построено только над терминалом a. Ошибка допущена между символами $''+''$ и $'')''$. Правило вида $D \to x+y)z$ в грамматике отсутствует, однако, имеется правило $S \to S + A$ и символ ")" принадлежит множеству $last1(A)$. Тогда вставим между "+" и ")" цепочку, выводимую из $''(S''$. Самой короткой цепочкой такого вида будет, например, цепочка $''(c''$. Иллюстрация приведена на рис. 6.2.

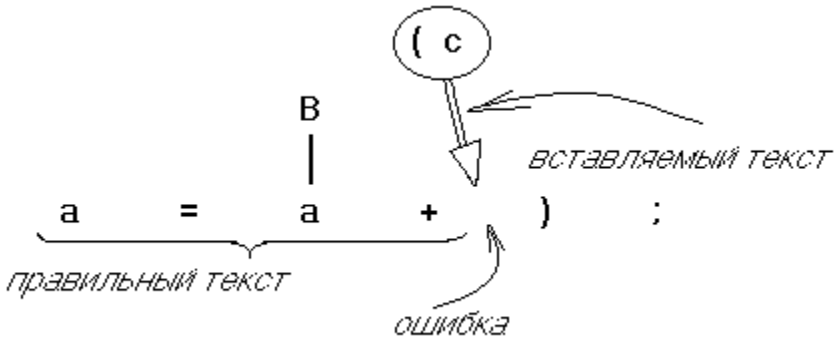


Рис. 6.2: Нейтрализация ошибки при восходящем разборе

В заключение необходимо отметить, что указанные в данной главе способы нейтрализации ошибок никоим образом не являются единственно возможными способами, которые можно использовать. Можно придумать другие алгоритмы нейтрализации, чтобы учесть особенности данного языка, Нейтрализация ошибок — это скорее искусство, чем наука, и указанные в этой главе способы можно рассматривать как примеры, реализующие некоторые идеи и методы нейтрализации ошибок.

6.5 Контрольные вопросы к разделу

1. Какие существуют методы исправления ошибок?

2. Чем исправление ошибки отличается от ее нейтрализации?

3. Как нейтрализуются ошибки в синтаксическом анализаторе, построенном методом синтаксических диаграмм?

4. Какие ошибки нет смысла нейтрализовать?

5. Какие символы можно вставлять? Как программируется нейтрализация вставкой?

6. Почему в выражениях синтаксические ошибки нейтрализуются редко?

7. Как избавиться от большого числа наведенных ошибок?

8. Можно ли предложить алгоритмы нейтрализации ошибок, полностью свободные от наведенных ошибок?

9. Какие символы выбираются в качестве базовых, до которых осуществляется пропуск при нейтрализации ошибок?

10. Какой алгоритм нейтрализации ошибок в $LL(1)$–анализаторе Вы можете предложить?

11. При какой стратегии разбора нейтрализация ошибок проще? Почему?

12. Какие методы нейтрализации ошибок при восходящем анализе Вы можете предложить?

13. Когда появляется необходимость нейтрализации семантических ошибок?

14. Как нейтрализуются семантические ошибки?

15. Зачем вводится понятие "ошибочный семантический тип"? Какой вид имеет таблица приведений для этого типа?

16. Приведите пример нейтрализации вставкой.

17. Приведите пример нейтрализации пропуском.

18. Приведите пример синтаксической диаграммы, в которой нейтрализации возникшей ошибки нерациональна.

19. Зачем в программе нейтрализации ошибок используются флаги ошибок? Сколько таких флагов нужно для метода синтаксических диаграмм? Сколько их необходимо для программы $LL(1)$–анализатора?

20. Нужна ли нейтрализация ошибок в программе интерпретатора?

6.6 Тесты для самоконтроля к разделу

1. Различаются или нет понятия нейтрализации и исправления ошибок компилятором?

Варианты ответов:

а) Понятия нейтрализации идентичны, это синонимы, обозначающие одни и те же действия.

б) Нейтрализация ошибки — это пропуск текста до позиции, с которой можно продолжать безошибочный анализ, а исправление — вставка верного символа в ошибочной позиции.

в) Нейтрализация ошибки применяется только при нисходящем анализе и заключается в удалении из магазина символов до тех пор, пока в верхушке магазина не окажется символ, соответствующий очередной отсканированной лексеме. Исправление ошибки применяется при любой стратегии разбора и заключается в замене неверного отсканированного символа на правильный.

г) Нейтрализация ошибки — это действия, направленные на продолжение анализа исходного модуля после обнаружения ошибки. При исправлении ошибки эти действия таковы, что должна получиться действительно правильная программа.

д) Нейтрализация ошибок выполняется на синтаксическом уровне, а исправление ошибок — на лексическом уровне.

Правильный ответ: г.

2. Что Вы можете сказать о наведенных ошибках?

1) Хороший компилятор наведенных ошибок не выдает. Появление наведенных ошибок — следствие плохого алгоритма нейтрализации ошибок.

2) Любой алгоритм нейтрализации ошибок представляет собой компромисс между желанием обнаружить как можно больше ошибок, и желанием избежать сообщений о несуществующих ошибках.

3) Наведенные ошибки — это несуществующие в транслируемой программе ошибки, которые выдал компилятор из–за того, что была выполнена нейтрализация предшествующей ошибки.

4) Наведенные ошибки могут быть всех типов: лексические, синтаксические, семантические.

Какие из указанных утверждений ложны?

Варианты ответов:

а) ложно 1;

б) ложно 2;

в) ложно 3;

г) ложно 4;

д) ложны 1 и 3;

е) ложны 1 и 4;

ж) ложны 2 и 3;

з) ложны 2 и 4;

и) ложны 1, 3 и 4;

к) ложны 2, 3 и 4;

д) все утверждения ложны;

м) все утверждения истинны.

Правильный ответ: е.

3. Что Вы можете сказать об алгоритмах нейтрализации ошибок? Какие из следующих утверждений истинны?

1) Все методы нейтрализации ошибки основаны на двух методах: вставке символа и пропуске символа или последовательности символов до тех пор, пока не встретится конструкция, допускающая анализ.

2) При нисходящем анализе нейтрализация ошибки происходит сложнее, чем при восходящем анализе, т.к. синтаксический анализатор имеет меньше информации о структуре дерева разбора.

3) При восходящем анализе нейтрализация основана на поиске соответствия висячих вершин и отсканированных лексем.

Варианты ответов:

а) истинно только 1;

б) истинно только 2;

б) истинно только 3;

в) истинны только 1 и 2;

г) истинны только 1 и 3;

г) истинны только 2 и 3;

д) все утверждения истинны;

е) все утверждения ложны.

Правильный ответ: а.

4. Какой метод нейтрализации ошибок применяется при восходящем анализе? Укажите применяемые методы из следующего перечня.

Варианты ответов:

1) сопоставление нетерминалов в верхушке магазина и текущего текста;

2) сопоставление терминалов в верхушке магазина и текущего текста;

3) удаление символа, после которого обнаружена ошибка;

4) удаление символа, перед которым обнаружена ошибка;

5) сопоставление текущего текста и правых частей правил грамматики.

а) совместное применение 1 и 2;

б) совместное применение 3 и 5;

в) совместное применение 4 и 5;

г) совместное применение 1, 2 и 3;

д) совместное применение 1, 2 и 5;

е) совместное применение 1, 2 и 4;

ж) применяются все перечисленные способы в совокупности.

Правильный ответ: в.

5. Какой принцип нейтрализации ошибок применяется при нисходящем анализе? Укажите применяемые методы из следующего перечня.

Варианты ответов:

а) поиск соответствия множеств $last(A_i)$ и $first(A_i)$ висячих вершин A_i и отсканированных лексем;

б) поиск соответствия множеств $follow(A_i)$ и $first(A_i)$ висячих вершин A_i и отсканированных лексем;

в) удаление символов магазина до тех пор, пока для нетерминала A в верхушке магазина не установится соответствие $first(A)$ с очередным отсканированным символом;

в) сканирование символов транслируемого текста до тех пор, пока для нетерминала A в верхушке магазина не установится соответствие $first(A)$ с очередным отсканированным символом;

Правильный ответ: а.

6.7 Упражнения к разделу

6.7.1 Задание 1.

Цель данного задания – программирование блока нейтрализация ошибок в программе синтаксического анализа, построенного методом рекурсивного спуска. Работу над заданием следует организовать методом последовательного выполнения нижеследующих операций.

1. Выделить синтаксические диаграммы, в которых невозможна никакая нейтрализация. Для каждого блока такой диаграммы указать стрелкой выход из соответствующей процедуры при обнаружении ошибки.

2. Выделить синтаксические диаграммы, в которых возможна нейтрализация ошибок методом вставки терминалов. Определить вставляемые лексические единицы и заштриховать их в соответствующих диаграммах.

3. Выделить терминальные символы, пропуском до которых может осуществляться нейтрализация обнаруженных ошибок. Определить синтаксические диаграммы, в которых осуществляется соответствующий пропуск. Отметить стрелками пропуски символов.

4. Если после выполнения пунктов 1 — 3 у Вас остались неиспользуемые синтаксические диаграммы, повторить действия 1 — 3. Исключение могут составлять только такие диаграммы, которые содержат только нетерминальные блоки, полная нейтрализация ошибок в которых проведена методами вставки или пропуска символов (пункты 2 или 3).

5. Написать подпрограмму пропуска до одного из заданных символов. В зависимости от предложенных Вами решений Вы можете либо построить универсальную программу, либо набор таких программ пропуска до требуемых символов.

6. Для того, чтобы реализовать функцию вставки символов, Вам надо предусмотреть в своей программе запоминание и восстановление текущего указателя исходного модуля.

7. В соответствии с разметкой синтаксических диаграмм, которую вы выполнили, вставить в отмеченные Вами точки программы фрагменты, соответствующие пунктам 1 — 2.

8. Отладить программу на ошибочных исходных модулях с большим числом ошибок. При отладке следить за тем, чтобы не пропускались конструкции, размеры которых сравнимы с длиной исходного модуля.

9. Если Ваша программа нейтрализации выдает слишком большое число наведенных ошибок или пропускает практически весь исходный модуль, вернуться к разметке синтаксических диаграмм и скорректировать разметку так, чтобы устранить замеченные недостатки.

6.7.2 Пример выполнения задания

Рассмотрим простейшую нейтрализацию ошибок в программе синтаксического анализатора, реализованного методом рекурсивного спуска.

Пусть грамматика имеет вид:

$$
\begin{array}{lll}
G: & S \rightarrow & intmain()T \\
 & T \rightarrow & T\ W|\varepsilon \\
 & W \rightarrow & D|F \\
 & D \rightarrow & \textbf{var}\ Z; \\
 & Z \rightarrow & Z,\ I|I \\
 & I \rightarrow & a \mid a{=}V \\
 & F \rightarrow & \textbf{function}\ (Z)\ Q \\
 & Q \rightarrow & \{K\} \\
 & K \rightarrow & KD|KO|\varepsilon \\
 & O \rightarrow & P\textbf{;}\ |Q|H|U|M|\ ; \\
 & U \rightarrow & \textbf{for}\ (P;V;P)\ O \\
 & M \rightarrow & \textbf{if}\ (V)\ O|\textbf{if}\ (V)\ O\ \textbf{else}\ O \\
 & P \rightarrow & N{=}V \\
 & N \rightarrow & N.a|a \\
 & V \rightarrow & V{>}A|V{>=}A|V{<}A|V{<=}A|V{==}A|V!{=}A|{+}A|{-}A|A \\
 & A \rightarrow & A{+}B|A{-}B|B \\
 & B \rightarrow & B{*}E|B/E|E \\
 & E \rightarrow & N|C|H|(V) \\
 & H \rightarrow & a(X)|a() \\
 & X \rightarrow & X,\ V|V \\
 & C \rightarrow & c_1|c_2|c_3|c_4
\end{array}
$$

В качестве иллюстрации метода вставки рассмотрим вставку пропущенной закрывающейся скобки в выражениях.

```
void E() {  // элементарное выражение
TypeLex l; int t,uk1;
uk1=GetUK(); t=Scaner(l);
if ( (t==TConsChar) || (t==TConsInt)
   ||(t==TConsFloat) ||(t==TConsExp) ) return;
if (t==TLS)
    {
    V();
    uk1=GetUK(); t=Scaner(l);
    if (t!=TPS) PrintError("ожидался символ )",l);
    PutUK(uk1);   // НЕЙТРАЛИЗАЦИЯ: вставили скобку
    return;
    }
if (t != TIdent) {
    PrintError("ожидался идентификатор вместо ",l);
    return; // НЕТ НЕЙТРАЛИЗАЦИИ
            // выходим из функции с установленным флагом ошибки
}
// для определения N или  H нужно иметь first2
t=Scaner(l);  PutUK(uk1); // восстанавливается uk начальное
if (t==TLS) H();
     else   N();
    // Независимо от того, были или нет в H() или N() ошибки,
}
```

Основное внимание всегда следует обращать на поведение метода нейтрализа-

ции на программе в целом. Поэтому важнейшее значение имеют методы обработки структурных, в частности, составных операторов. В нашем примере это диаграммы Q — составной оператор и K — последовательность операторов и описаний. Обнаруженная в простом операторе ошибка приведет к тому, что при возврате в K признак ошибки — переменная $FlagError$ — равна 1. Выделим в качестве символов–ограничителей встреченной ошибки знаки

— точка с запятой,

— конец текста,

— обе фигурные скобки.

Будем пропускать текст до тех пор , пока не встретим один из этих знаков. Теперь необходимо помнить, что не всегда в ошибочной программе обязателен знак закрывающейся фигурной скобки, поэтому цикл в K работает не по условию

```
while ( t != TFPS ) ,
```

а по условию

```
while ( ( t != TFPS ) && ( t != TEnd ) ).
```

Кроме того, следует учесть, что после пропуска до какого–либо из указанных знаков–ограничителей нужно либо продолжить обработку этого знака (в нашем примере это знаки фигурных скобок), либо проигнорировать знак наряду с предшествующими (в нашем примере это знак точки с запятой). Тогда программа со встроенной нейтрализаций примет вид:

```
void K()
// Операторы и описания
{
TypeLex l; int t,uk1;
uk1=GetUK(); t=Scaner(l);  PutUK(uk1);
while (( t!=TFPS) && (t!=TEnd) )
     {
     if (t==TVar)  D();
else       O();
     if (FlagError)
 {
 do{
   // будем пропускать до конца оператора, в котором была ошибка
   uk1=GetUK(); t=Scaner(l);
   }while(( t!=TFPS) && (t!=TEnd) && (t!= TTZpt ) && (t!=TFLS ));
 if (t != TTZpt )  // пропуск оператора вместе с точкой с запятой
PutUK(uk1);
 FlagError=0;
 }
     uk1=GetUK(); t=Scaner(l);  PutUK(uk1);
     }
}
```

Если во всех оставшихся функциях организовать выход по оператору $return$ как при обнаружении ошибки, так и при условии $FlagError == 1$, то в большинстве случаев нейтрализация будет работать без наведенных ошибок. Исключение составят ошибки в других структурных операторах, например, в операторах if или for.

6.7.3 Задание 2.

Цель данного задания – встроить в программу LL(1)–анализатора блок нейтрализации ошибок. Для этого предлагается выполнить следующие операции.

1. Выделить в языке такие терминальные символы, пропуск текста до которых (включая эти символы или нет) позволит перейти к анализу последующего фрагмента программы.

2. Выделить список символов, функция $first_1()$ для которых содержит указанные в пункте 1 терминальные символы. Такими символами могут быть как нетерминалы, находящиеся в верхушке магазина, так и терминальные символы.

3. Выделить список символов, функция $last_1()$ для которых содержит указанные в пункте 1 терминальные символы. Такими символами могут быть как нетерминалы, находящиеся в верхушке магазина, так и терминальные символы.

4. Если обнаружена ошибка, Ваша программа перешла на выполнение функции $PrintError()$. Следовательно, именно в эту функцию проще всего встроить операторы нейтрализации ошибки. После выдачи сообщения об ошибке следует пропустить текст исходного модуля до тех пор, пока не будет прочитан либо маркер конца, либо один из выделенных символов. После того, как первый подходящий символ найден, надо подготовить соответствующим образом верхушку магазина. Допустим, это терминал a. Возможны два случая.

а) Терминал a выделен по правилам пункта 2, т.е. существуют нетерминалы A_1, A_2, ... , A_k, вывод из которых может начинаться символом a. Тогда нужно стирать верхушку магазина до тех пор, пока верхним символом не окажется один из A_1, A_2, ..., A_k. Если такой символ есть, программа синтаксического анализа может продолжить работу.

б) Терминал a выделен по правилам пункта 3, т.е. существуют нетерминалы B_1, B_2, ..., B_m, вывод из которых может заканчиваться символом a. Тогда в отличие от варианта (а) нужно найденный в магазине символ стереть, а затем отсканировать новую лексему и продолжить работу.

При отладке программы обратите особое внимание на конструкцию ошибочных текстов: нужно проверить работоспособность алгоритма на исходном модуле с единственной ошибкой в программе, а затем на примерах с большим количеством ошибок.

6.7.4 Пример выполнения задания

Рассмотрим грамматику примера к главе 5. В качестве символов–ограничителей выберем знаки, которые позволяют закончить анализ текста и с достаточно большой вероятностью считать, что дальше идет понятная анализатору конструкция. Это знаки открывающейся и закрывающейся фигурных скобок, точка с запятой. Соответствующие терминалы и нетерминалы, функции $first_1()$ и $last_1()$ для которых содержат эти символы, представлены таблице (см. рис. 6.3).

Реализация программы показывает, что такой пропуск текста обладает следующими недостатками:

1) иногда пропускается слишком большой фрагмент исходного модуля; особенно это заметно, если ошибка обнаружена не внутри тела функции, а в списке глобальных описаний, когда в магазине находится нетерминал T или W;

знак	для функции $first_1$	для функции $follow_1$
{	{, Q, O	
}	}	O, Q
;	;, O, K	O

Рис. 6.3: Таблица нейтрализации для $LL(1)$ – анализатора

2) из-за пропусков слишком больших фрагментов текста появляются наведенные ошибки (как мы уже отмечали ранее, избежать наведенных ошибок невозможно, но надо стараться минимизировать из число).

Поэтому в список символов – ограничителей внесем еще некоторые ключевые слова, начинающие конструкции, синтаксис которых независим от других. В качестве таких слов можно выбрать var, $function$, for, if, $else$. Ключевые слова из тегов мы не будем выбирать по соображениям семантики – если их синтаксис неверен, то рассматривать программу бессмысленно. Кроме того, они встречаются в тексте программы только однократно в начале и конце, а, следовательно, пользователь легко исправит такие ошибки в программе.

В результате проделанных операций у нас получилась базовая программа нейтрализации, которая вызывается из PrintError(). Поведение программы можно существенно улучшить, накладывая ограничения на длину удаляемой верхушки магазина: например, лучше отсканировать следующий символ–ограничитель в исходном модуле, чем удалять из магазина слишком сложную конструкцию.

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Ахо А., Лам М., Сети Р., Ульман Д.. Компиляторы: принципы, технологии и инструментарийю — М: "Вильямс", 2008, 768с.

2. Вирт н. Построение компиляторов. - ДМК Пресс, 2010. - 192 с.

3. Свердлов С.З. Языки программирования и методы трансляции. — "Питер", 2007, 637 с.

4. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. — М. "Вильямс", 2008, 378с.

Дополнительная литература

1. Ахо А., Ульман Дж.. Теория синтаксического анализа, перевода, компиляции. В 2 т. Т. 1,2. — М.: Мир, 1980.

2. Бек Л. Введение в системное программирование. - М.: Мир, 1988, 448 с.

3. Вирт Н. Алгоритмы и структуры данных. — СПб: "Невский диалект" , 2001, 351 с.

4. Гордеев А.В., Молчанова А.Ю. Системное программное обеспечение. — СПб, "Питер" , 2002, 736 с.

5. Керниган Б., Пайк Р. Практика программирования. — СПб: "Невский диалект", 2001, 380 с.

6. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 1999.

7. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979, 654 с.

8. Молчанов А. Системное программное обеспечение. — "Питер", 2010, 400с.

Оглавление