МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОУ ВПО «ЧЕРЕПОВЕЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт информационных технологий

О. Г. Ганичева

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

ЧЕРЕПОВЕЦ
2011

УДК 681.3.068
ББК 32.973.018
Г19

Рассмотрено на заседании кафедры программного обеспечения ЭВМ, протокол № 9 от 08.05.09.

Одобрено УМС ГОУ ВПО ЧГУ, протокол № 3 от 19.11.09.

Г19

Ганичева О.Г. Теория языков программирования и методы трансляции: Учеб. пособие. – Череповец: ГОУ ВПО ЧГУ, 2011. – 186 с.

ISBN 978–5–85341–417–4

В настоящем учебном пособии изложен учебный материал по дисциплине «Теория языков программирования и методы трансляции» в соответствии с Государственным образовательным стандартом. Учебное пособие содержит теоретические основы формальных грамматик и языков, классификацию грамматик Хомского, основы лексического анализа, методы синтаксического анализа языков программирования, формальные методы описания и реализации синтаксически управляемого перевода.

Предназначено для студентов специальности 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем».

Рецензенты: *Е.В. Королева* – канд. техн. наук (ОАО «Фирма «СТОИК»); *А.Ф. Касторнов* – канд. пед. наук, профессор (ГОУ ВПО ЧГУ)

Научный редактор: *Е.В. Ершов* – д-р техн. наук, профессор (ГОУ ВПО ЧГУ)

ISBN 978–5–85341–417–4

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Теория построения трансляторов в большей степени посвящена методам выявления структуры входного текста и использования этой структуры для преобразования входа с сохранением его смысла. Эти методы базируются на идеях, высказанных Н. Хомским в середине XX века, с которых начиналась структурная лингвистика.

Согласно этим идеям, смысл текста существенно связан с его структурой. Двусмысленные тексты имеют несколько структур, недвусмысленные – одну структуру. Смысл двусмысленных и однозначных предложений языка определяется структурой предложений. В современных трансляторах искусственных языков эти идеи и понятия являются базовыми.

Примеры.

1. Порядок сменит хаос.
2. Казнить нельзя, помиловать.
   Казнить, нельзя помиловать.
3. Бытие определяет сознание.

Схема понимания человеком предложений языка формируется в подсознании: на первом шаге строится структура предложения, на втором – по структуре «вычисляется» смысл предложения.

Итак, смысл определяется предложением и его структурой.

Идеи Н. Хомского являются базовыми в современных трансляторах искусственных языков.

1. НАЧАЛЬНЫЕ СВЕДЕНИЯ О КОМПИЛЯЦИИ

1.1. Общие сведения о языке программирования и структуре транслятора

Любая ЭВМ имеет собственный язык программирования – машинный язык – и может исполнять программы, записанные только на этом языке. С помощью машинного языка можно описать любой алгоритм, что оказывается достаточно сложным для неподготовленного человека, так как здесь необходимо знать устройство и функционирование ЭВМ. Языки высокого уровня позволяют решать эту задачу; они обладают развитой структурой данных и средствами их обработки, не зависящими от языка конкретной машины.

Язык высокого уровня – это входной язык, описание на языке высокого уровня – исходная программа.

Языковой процессор – это программа на машинном языке, позволяющая вычислительной машине понимать и выполнять программы на входном языке.

Языковые процессоры делятся на интерпретаторы и трансляторы.

Интерпретатор – это программа, которая на входе имеет программу на входном языке, распознает и реализует конструкции этого языка, выдает на выходе результаты вычислений.

Транслятор – это программа, которая на входе имеет исходную программу, а на выходе – программу, функционально эквивалентную исходной, – объектную (на объектном языке).

В свою очередь, трансляторы делятся на ассемблеры и компиляторы.

Транслятор, который использует в качестве входного языка близкий к машинному, называют *ассемблером*.

Транслятор для языка высокого уровня называют *компилятором*.

Итак, *компилятор* – это программа, которая считывает текст на исходном языке и транслирует его на другой – целевой язык.

На первый взгляд, разнообразие компиляторов велико. Используются тысячи исходных языков: традиционные – Паскаль, Фортран, Си; специализированные, возникающие во всех областях компьютерных приложений. Целевые языки тоже разнообразны – от языков микропроцессоров до языков суперкомпьютеров.

Компиляторы классифицируют как однопроходные, многопроходные, исполняющие, отлаживающие, оптимизирующие – в зависимости от предназначения, принципов и технологий их создания.

Несмотря на кажущееся разнообразие и сложность задач, решаемых компиляторами, основные задачи, выполняемые компиляторами, по сути, одни и те же. Понимая это, можно создавать компиляторы для различных исходных языков и целевых машин с использованием одних и тех же базовых принципов.

Первый компилятор FORTRAN потребовал 18 человеко-лет работы.

1.2. Модель анализа – синтеза компиляции

Компиляция состоит из двух основных частей: анализа и синтеза.

Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления.

В процессе анализа определяются и записываются в иерархическую древовидную структуру операции, заданные исходной программой. Часто используется специальный вид дерева – синтаксическое дерево, в котором каждый узел представляет операцию, а его дочерние узлы – аргументы операции (операнды).

Синтез – это конструирование требуемой целевой программы из промежуточного представления.

Процесс трансляции разбивается на несколько этапов, за выполнение каждого из которых отвечает свой блок.

Основные фазы трансляции: лексический анализ, синтаксический анализ, семантический анализ, синтез объектной программы.

Цели лексического анализа:

1) перевод исходной программы на внутренний язык компилятора, в котором ключевые слова, идентификаторы, метки и константы приведены к одному формату и заменены условными кодами: числовыми или символьными, которые называются *дескрипторами*. Каждый дескриптор состоит из двух частей: класса-типа лексемы и указателя на адрес в памяти, где хранится информация о конкретной лексеме. Эта информация организуется в виде таблиц;

2) лексический контроль – выявление в программе недопустимых слов.

Цели синтаксического анализа:

1) перевод последовательности образов лексем в форму промежуточной программы;

2) синтаксический контроль – выявление синтаксических ошибок в программе.

Цель семантического анализа – проверка семантических ошибок в исходной программе. Например, проверка типов, единственность описания каждого идентификатора.

Синтез объектной программы – построение объектного кода программы на основании внутреннего представления и информации, содержащейся в таблице идентификаторов. Основные этапы синтеза: генерация промежуточного кода, оптимизация кода и генерация кода.

Порядок выполнения фаз компиляции может меняться в разных вариантах компиляторов. Состав фаз тоже может быть изменен, т.е. некоторые фазы могут быть объединены в одну фазу или, наоборот, разбиты на составляющие.

1.3. Понятие прохода. Однопроходные и многопроходные компиляторы

Проход – это процесс последовательного чтения компилятором данных из внешней памяти, их обработки и помещения результата работы во внешнюю память.

Компилятор может просмотреть текст исходной программы, сразу выполнить все фазы компиляции и получить результат – объектный код. В другом варианте он выполняет над исходным текстом только некоторые из фаз компиляции и получает не конечный результат, а набор некоторых промежуточных данных. Эти данные затем снова подвергаются обработке, и этот процесс может повторяться несколько раз.

Реальные компиляторы, как правило, выполняют трансляцию текста исходной программы за несколько проходов. Один проход чаще всего включает в себя выполнение одной или нескольких фаз компиляции. Результатом промежуточных проходов является внутреннее представление исходной программы, результатом последнего прохода – результирующая объектная программа. При выполнении каждого прохода компилятору доступна информация, полученная в результате предыдущих проходов. Как правило, компилятор стремится использовать информацию, полученную на предыдущем проходе, но может обращаться и к данным более ранних проходов.

Информация, получаемая компилятором при выполнении проходов, недоступна пользователю. Поэтому человек, работающий с компьютером, может не знать, сколько проходов выполняет компилятор. Он всегда видит только исходную программу и результирующую объектную программу. Однако количество проходов – важная техническая характеристика компилятора.

1.4. Фазы компилятора

Концептуально компилятор работает *пофазно*. В процессе каждой фазы происходит преобразование исходной программы из одного представления в другое. Типичное разбиение компилятора на фазы показано на рис. 1.

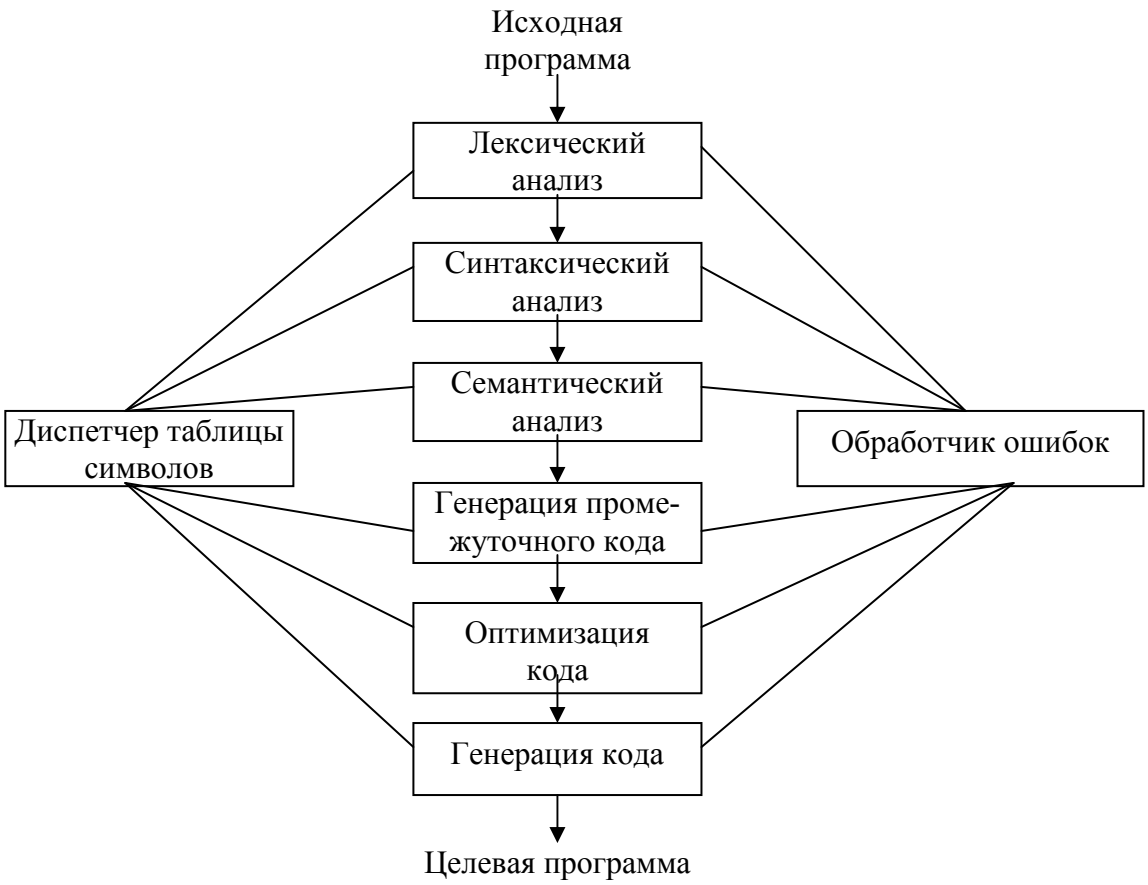


Рис. 1. Фазы компилятора

На практике некоторые фазы могут быть сгруппированы вместе и промежуточные представления программы внутри таких групп могут явно не строиться. Управление таблицей символов и обработка ошибок показаны во взаимодействии с шестью фазами: лексическим анализом, синтаксическим анализом, семантическим анализом, генерацией промежуточного кода, оптимизацией кода и генерацией кода. Неформально диспетчер таблицы символов и обработчик ошибок также могут считаться фазами компилятора.

1.5. Управление таблицей символов

Одной из важных функций компилятора является запись используемых в исходной программе идентификаторов и сбор информации о различных атрибутах каждого идентификатора. Эти атрибуты представляют собой сведения об отведенной идентифи-

катору памяти, его типе, области видимости. При использовании имен процедур атрибуты говорят о количестве и типе их аргументов, методе передачи каждого аргумента (например, по ссылке) и типе возвращаемого значения, если таковое имеется.

Таблица символов представляет собой структуру данных, содержащую записи о каждом идентификаторе с полями для его атрибутов. Данная структура позволяет быстро найти информацию о любом идентификаторе и внести необходимые изменения.

Если в исходной программе лексическим анализатором обнаружен идентификатор, он записывается в таблицу символов. Однако атрибуты идентификатора обычно не могут быть определены в процессе лексического анализа. Например, в объявлении переменных на языке Pascal

Var name1, name2, name3: real;

когда лексический анализатор находит идентификаторы *name1, name2, name3*, их тип real еще неизвестен.

В процессе остальных фаз информация об идентификаторах вносится в таблицу символов и используется различными способами. Например, при семантическом анализе и генерации промежуточного кода необходимо знать типы идентификаторов, чтобы гарантировать их корректное использование в исходной программе и сгенерировать правильные операции по работе с ними. Обычно генератор кода вносит в таблицу символов и использует детальную информацию о памяти, назначенной идентификаторам.

1.6. Обнаружение ошибок и сообщение о них

В каждой фазе компиляции могут встретиться ошибки. Однако после их обнаружения необходимы определенные действия, чтобы продолжить компиляцию и выявить другие ошибки в исходной программе. Компилятор, который останавливается при обнаружении первой же ошибки, не настолько полезен в работе. В процессе синтаксического и семантического анализа обычно обрабатывается большая часть ошибок, обнаруживаемых компилятором.

При лексическом анализе выявляются ошибки, при которых символы из входного потока не формируют ни одну из лексем языка.

При синтаксическом анализе выявляются ошибки, при которых поток нарушает структурные правила (синтаксис) языка.

В процессе семантического анализа компилятор пытается обнаружить конструкции, корректные с точки зрения синтаксиса, но не имеющие смысла с точки зрения выполняемых операций (например, попытка сложения двух идентификаторов, один из которых – имя массива, а второй – имя процедуры).

1.7. Фазы анализа

В процессе трансляции изменяется внутреннее представление исходной программы. Например, рассмотрим инструкцию

$$\text{pos} := \text{i} + \text{r} * 60 \qquad (1)$$

На рис. 2 показано внутреннее представление инструкции после каждой фазы.

При лексическом анализе символы исходной программы считываются и группируются в поток лексем, в котором каждая лексема представляет собой логически связанную последовательность символов – идентификатор, ключевое слово (if, while и т.п.), символ пунктуации или многосимвольный оператор типа :=. Последовательность символов, формирующая слово, образует *лексему*.

Некоторые лексемы дополняются «лексическим значением». Например, когда найден идентификатор **r**, лексический анализатор не только генерирует лексему **id**, но и вносит ее в таблицу символов, если ее там нет. Лексическое значение, связанное с этим появлением лексемы **id**, указывает на запись **r** в таблице символов.

Инструкция (1) после лексического анализа выглядит так:

$$id_1 = id_2 + id_3 \cdot 60 \qquad (2)$$

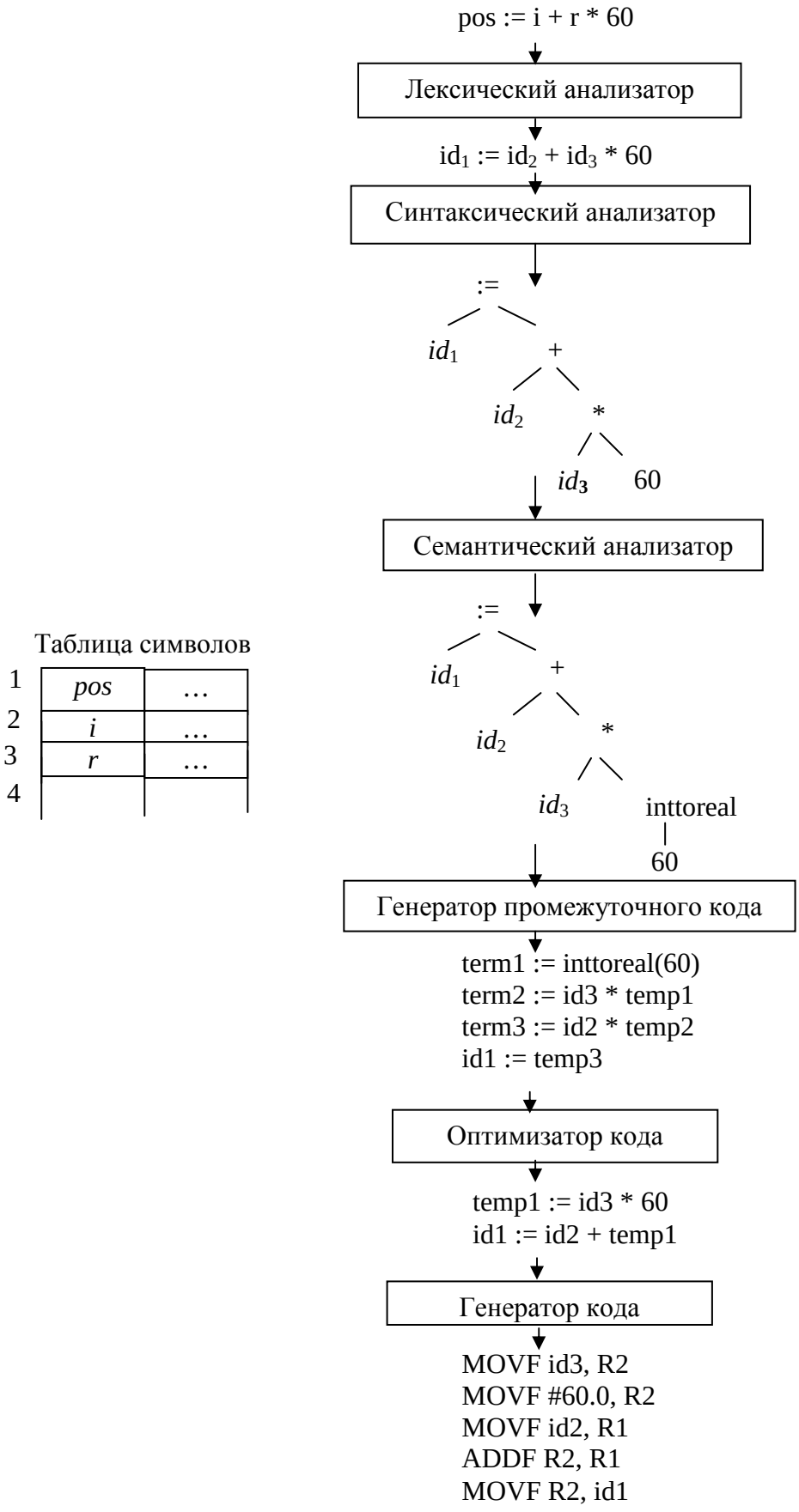
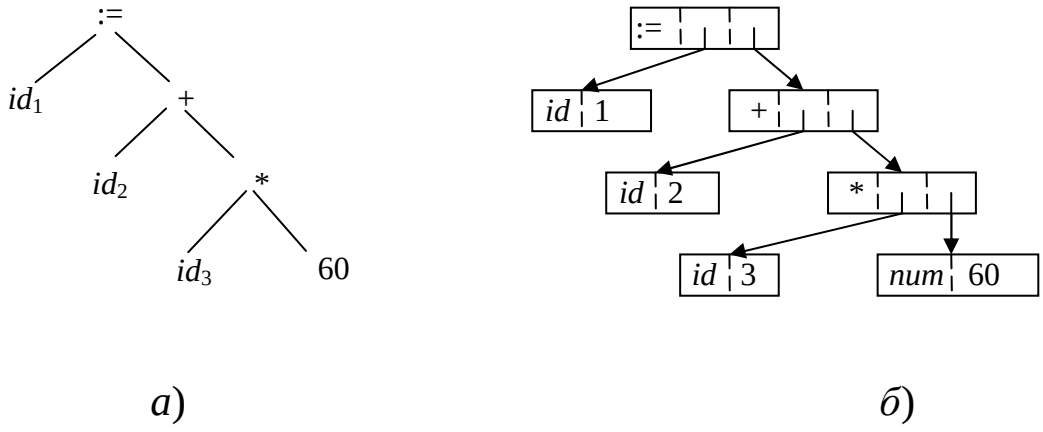


Рис. 2. Трансляция инструкции

При синтаксическом анализе на поток лексем налагается иерархическая структура, которую можно изобразить с помощью синтаксического дерева (рис. 3, *а*). Типичная структура данных для такого дерева показана на рис. 3, *б*. Ее внутренний узел представляет собой запись с полем для оператора и двумя полями с указателями на дочерние записи. Лист представляет собой запись с двумя или более полями. Дополнительная информация о языковых конструкциях может содержаться в дополнительных полях записей для узлов.



а) *б*)

Рис. 3. Структура данных (*б*) для дерева (*а*)

Вопросы

1. Постройте структуры и проанализируйте следующие примеры двусмысленных предложений русского языка.

Я мечтаю жить в Париже, как мой отец. (Мечтаю, как отец, или жить в Париже, как отец?);

Вывешены списки студентов, которые находились в деканате. (Списки или студенты находились в деканате?);

Мать велела сыну налить воды себе. (Кому налить воды, сыну или матери?).

2. Что такое синтаксис и семантика языка?

3. В чем отличие компилятора от транслятора?

4. От чего зависит количество проходов, выполняемых компилятором?

5. Почему компилятор языка Си работает дольше, чем компилятор языка Паскаль?

2. ЛЕКСИЧЕСКИЙ АНАЛИЗ

2.1. Назначение лексического анализатора

Лексический анализ представляет собой первую фазу компиляции. Его основная задача состоит в чтении новых символов и выдачи последовательности лексем, используемых синтаксическим анализатором в своей работе. Лексической единицей языка является лексема.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка.

Лексемами языков естественного языка общения между людьми являются слова. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п. *Состав возможных лексем* каждого конкретного языка программирования определяется *синтаксисом* этого языка.

Каждый класс лексем описывается правилом, называемым *шаблоном*. Шаблон соответствует каждой строке в наборе-классе лексем. Лексема же представляет собой последовательность символов исходной программы, которая соответствует шаблону. Например, в инструкции Pascal

```
const e = 2,78;
```

подстрока e представляет собой лексему «идентификатор».

Шаблон – правило, описывающее набор лексем, которые могут представлять определенную лексему в исходной программе. Так, шаблон **const** (см. табл. 1) представляет собой просто строку const, являющуюся ключевым словом. Шаблон **relation** – набор всех шести операторов сравнения в Pascal. Для точного описания сложных шаблонов типа **id** (для идентификатора) и **num** (для числа) нужно использовать регулярные выражения.

Лексический анализатор (или *сканер*) – это часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка

Таблица 1

Примеры лексем

Шаблон-лексема	Пример лексем	Неформальное описание шаблона
const	const	const
relation	<, <=, >, >=, <>, =	< или <= или >или >= или <> или =
num	2.78	Любая числовая константа
id	e	Буква, за которой следуют буквы и цифры

На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

С теоретической точки зрения лексический анализатор не является обязательной частью компилятора. Все его функции могут выполняться на этапе синтаксического разбора. Однако лексический анализ включают в состав практически всех компиляторов по следующим причинам:

– применение лексического анализатора упрощает работу с текстом исходной программы на этапе синтаксического разбора и сокращает объем обрабатываемой информации;

– для выделения в тексте и разбора лексем применяется простая и эффективная техника анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;

– при конструкции компилятора, когда лексический анализ реализован отдельно от синтаксического, для перехода от одной версии языка программирования к другой достаточно только перестроить относительно простой лексический анализатор.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. Какие функций должен выполнять лексический анализатор и какие типы лексем он должен выделять во входной программе, решают разработчики компилятора.

Поскольку лексический анализатор является частью компилятора, который считывает исходный текст, он также может выполнять некоторые второстепенные задачи. К ним, например, относятся удаление из текста исходной программы комментариев и не несущих смысловой нагрузки пробелов (а также символов табуляции и новой строки). Еще одна задача состоит в согласовании сообщений об ошибках компиляции и текста исходной программы. Например, лексический анализатор может подсчитать количество считанных строк и указать строку, вызвавшую ошибку. В некоторых компиляторах лексический анализатор создает копию текста исходной программы с указанием ошибок. Если исходный язык поддерживает макросы и директивы процессора, то они также могут реализовываться лексическим анализатором.

Основные функции лексического анализатора:

1) исключение из текста исходной программы комментариев;

2) исключение из текста исходной программы незначащих пробелов, символов-табуляций и перевода строки;

3) выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей.

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. Но для многих языков программирования на этапе лексического анализа может быть недостаточно информации для однозначного определения типа и границ очередной лексемы.

Поэтому в большинстве компиляторов лексический и синтаксический анализаторы – это взаимосвязанные части. Возможны два метода организации взаимосвязи лексического анализа и синтаксического разбора:

– последовательный;

– параллельный.

При последовательном варианте лексический анализатор просматривает весь текст исходной программы от начала до конца один раз и преобразует его в структурированный набор данных. Этот набор данных называют также *таблицей лексем*. В таблице

лексем ключевые слова языка, идентификаторы и константы, как правило, заменятся на специально оговоренные коды, им соответствующие (конкретная кодировка определяется разработчиком при реализации компилятора). Таблица лексем строится полностью вся сразу, и больше к ней компилятор не возвращается. Всю дальнейшую обработку выполняют следующие фазы компиляции.

При параллельном варианте лексический анализ исходного текста выполняется поэтапно так, что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора при возникновении ошибки может происходить «откат назад», чтобы попытаться выполнить анализ текста на другой основе. И только после того, как синтаксический анализатор успешно выполнит разбор очередной конструкции языка, лексический анализатор помещает найденные лексемы в таблицу лексем и таблицу идентификаторов и продолжает разбор дальше в том же порядке [1].

Работа синтаксического и лексического анализаторов при их параллельном взаимодействии изображена на рис. 4.

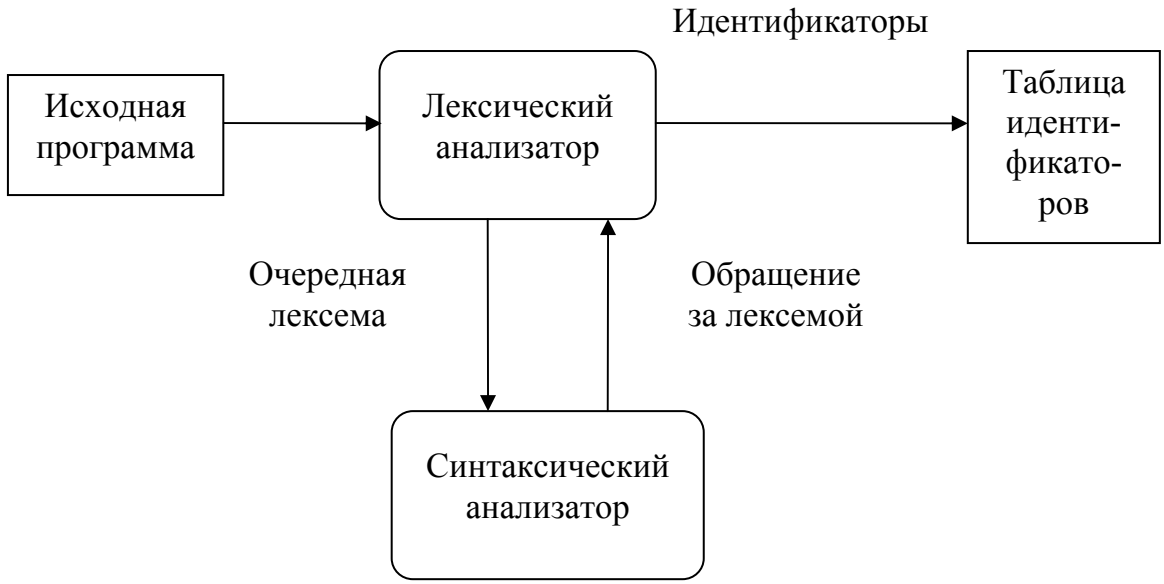


Рис. 4. Параллельное взаимодействие лексического и синтаксического анализаторов

Пример 1. Таблица лексем фрагмента кода на языке Pascal и соответствующая ему таблица лексем (см.табл. 2):

```
…
begin
    for j:=1 to K do
            f := f * 0.5
…
```

Таблица 2

Лексемы программы

Лексема	Тип лексемы	Значение
begin	Ключевое слово	*Y*1
for	Ключевое слово	*Y*2
j	Идентификатор	*id*:1
:=	Знак присваивания	:=
1	Целочисленная константа	1
to	Ключевое слово	*Y*3
K	Идентификатор	*K*: 2
do	Ключевое слово	*Y*4
f	Идентификатор	*id* : 3
:=	Знак присваивания	:=
f	Идентификатор	*id*: 3
*	Знак арифметической операции	*
0.5	Вещественная константа	0.5

Графа «Значение» в табл. 2 подразумевает некое кодовое значение, которое будет помещено в итоговую таблицу лексем в результате работы лексического анализатора. Эти значения являются условными. Конкретные коды определяются при реализации компилятора. Для идентификаторов устанавливается связь таблицы лек-

сем с таблицей идентификаторов (в примере это отражено индексом, следующим после идентификатора за знаком «:», а в реальном компиляторе это определяется его реализацией).

Последовательный вариант организации взаимодействия лексического анализа и синтаксического разбора является более эффективным, так как он не требует организации сложных механизмов обмена данными и не нуждается в повторном прочтении уже разобранных лексем. Этот метод является и более простым. Однако не для всех языков программирования можно организовать такое взаимодействие. Это зависит в основном от синтаксиса языка, заданного его грамматикой. Большинство современных широко распространенных языков программирования, таких как C и Pascal, тем не менее позволяет построить лексический анализ по более простому, последовательному методу, который дает ряд определенных преимуществ.

Причины, по которым фаза анализа компиляции разделяется на лексический и синтаксический анализы:

1. Упрощение разработки. Отделение лексического анализатора от синтаксического часто позволяет упростить одну из фаз анализа. Например, включение в синтаксический анализатор комментариев и пробелов существенно сложнее, чем удаление их лексическим анализатором.

2. Увеличивается эффективность компилятора. Отдельный лексический анализатор позволяет создать специализированный и потенциально более эффективный процессор для решения поставленной задачи.

3. Увеличивается переносимость компилятора. Особенности входного алфавита и другие специфичные характеристики используемых устройств могут ограничивать возможности лексического анализатора.

2.2. Атрибуты лексем

Если шаблону соответствует несколько лексем, лексический анализатор должен обеспечить дополнительную информацию о

лексемах для последующих фаз компиляции. Например, шаблон **num** (см. табл. 1) может соответствовать как строке 2.78, так и строке 2.19, и при генерации кода крайне важно знать, какая именно строка соответствует шаблону.

Лексический анализатор хранит информацию о лексемах и связанных с ними атрибутах. На практике лексемы обычно имеют единственный атрибут – указатель на запись в таблице символов, в которой хранится информация о соответствующей лексеме. Для диагностических целей могут понадобиться как лексемы идентификаторов, так и номера строк, в которых они впервые появились в программе. Вся эта (и другая) информация может храниться в записях в таблице символов.

Пример 2. Лексемы и связанные с ними значения атрибутов для инструкции

$$E = M * 2$$

записываются как последовательность пар:

< **id**, указатель на запись в таблице символов для *E* >
< **assign_op**, >
< **id**, указатель на запись в таблице символов для *M* >
< **mult_op**, >
< **num**, целое значение 2 >

В некоторых парах нет необходимости в значении атрибута – первого компонента вполне достаточно, чтобы идентифицировать лексему. В этом примере лексема **num** задана атрибутом с целым значением. Компилятор может хранить строку символов, составляющих число, в таблице символов, сделав, таким образом, атрибут лексемы **num** указателем на запись в таблице символов.

2.3. Общие принципы построения лексических анализаторов

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов в большинстве случаев является регулярным, т. е. может быть описан

с помощью регулярных грамматик. Распознавателями для регулярных языков являются конечные автоматы. Существуют правила, с помощью которых для любой регулярной грамматики может быть построен недетерминированный конечный автомат, распознающий цепочки языка, заданного этой грамматикой. Конечный автомат для каждой входной цепочки языка дает ответ на вопрос о том, принадлежит или нет цепочка языку, заданному автоматом.

В общем случае задача сканера несколько шире, чем просто проверка цепочки символов лексемы на соответствие ее входному языку. Кроме того, сканер должен выполнить следующие действия:

– четко определить границы лексемы, которые в исходном тексте явно не заданы;

– выполнить действия для сохранения информации об обнаруженной лексеме (или выдать сообщение об ошибке, если лексема неверна).

2.4. Определение границ лексем

Выделение границ лексем представляет определенную проблему. Ведь во входном тексте программы лексемы не ограничены никакими специальными символами. Определение границ лексем – это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание. В общем случае эта задача может быть сложной, и тогда требуется параллельная работа лексического анализатора, синтаксического разбора и, возможно, – семантического анализа. Для большинства входных языков границы лексем распознаются по заданным терминальным символам. Эти символы – пробелы, знаки операций, символы комментариев, а также разделители (запятые, точки с запятой и т.п.) Набор таких терминальных символов может варьироваться в зависимости от синтаксиса входного языка.

Как правило, лексические анализаторы действуют по следующему принципу:

1) очередной символ из входного потока данных добавляется в лексему всегда, когда он может быть туда добавлен;

2) как только символ не может быть добавлен в лексему, то считается, что он является границей лексемы и началом следующей лексемы.

При этом от пользователя явно требуется указать с помощью пробелов (или других незначащих символов) границы лексем. Такой подход возможен для большинства входных языков.

2.5. Выполнение действий, связанных с лексемами

Основная задача лексического анализатора – обнаружение лексемы во входном потоке и ее распознавание, т. е. определение класса, к которому она относится.

В основе распознавателя лексем лежит конечный автомат (КА), который должен иметь не только входной, но и выходной язык. Он не только должен уметь распознать правильную лексему на входе, но и породить связанную с ней последовательность символов на выходе. В такой конфигурации КА преобразуется в конечный преобразователь.

Таким образом, кроме порождения цепочки символов выходного языка, лексический анализатор должен уметь выполнять такие действия, как запись найденной лексемы в таблицу лексем, поиск ее в таблице символов и запись новой лексемы в таблицу символов. Набор действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же при обнаружении конца распознаваемой лексемы.

2.6. Практическая реализация лексических анализаторов

В принципе компилятор может иметь в своем составе не один, а несколько лексических анализаторов, каждый из которых предназначен для выборки и проверки определенного типа лексем.

Таким образом, обобщенный алгоритм работы простейшего лексического анализатора в компиляторе можно описать следующим образом:

– из входного потока выбирается очередной символ, в зависимости от которого запускается тот или иной лексический анализа-

тор (символ может быть также проигнорирован либо признан ошибочным);

– запущенный лексический анализатор просматривает входной поток символов программы на исходном языке, выделяя символы, входящие в требуемую лексему, до обнаружения очередного символа, который может ограничивать лексему, либо до обнаружения ошибочного символа;

– при успешном распознавании информация, о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, алгоритм возвращается к первому этапу и продолжает рассматривать входной поток символов с того места, на котором остановился лексический анализатор;

– при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации лексического анализатора – либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

В целом техника построения лексических анализаторов основывается на моделировании работы детерминированных и недетерминированных КА с дополнением функций распознавателя вызовами функций обработки ошибок, а также заполнения таблиц символов и таблиц идентификаторов. Такая техника не требует сложной математической обработки и принципиально важных преобразований входных грамматик. Для разработчиков лексических анализаторов важно только решить, где кончаются функции лексического анализа и начинаются функции синтаксического разбора. После этого процесс построения лексического анализатора легко поддается автоматизации.

2.7. Лексические ошибки

На уровне лексического анализатора определяются только некоторые ошибки, поскольку лексический анализатор рассматривает исходный текст программы в ограниченном контексте. Если в программе на языке С строка *if* впервые встретится в контексте

$$if\ (a = = f(x)) \ldots ,$$

то лексический анализатор не сможет определить, что именно представляет собой *if* – неверно записанное слово *if* или необъявленный идентификатор функции. Поскольку *if* является корректным идентификатором, лексический анализатор должен вернуть лексему для идентификатора и предоставить обработку ошибки другой части компилятора.

Лексический анализатор может не продолжать работу, поскольку ни один из шаблонов не соответствует оставшейся части входного потока. Простейшим выходом в этой ситуации будет восстановление в «режиме паники». То есть просто пропускаются входные символы до тех пор, пока лексический анализатор не встретит распознаваемую лексему. Иногда это запутывает синтаксический анализатор, но для интерактивной среды данная технология может оказаться вполне подходящей.

Существуют и другие возможные действия по восстановлению после ошибки:

1. Удаление лишних символов.
2. Вставка пропущенных символов.
3. Замена неверного символа верным.
4. Перестановка двух соседних символов.

При восстановлении корректного входного потока могут выполняться различные преобразования. Простейшая стратегия состоит в проверке, не может ли начало оставшейся части входного потока быть заменено корректной лексемой путем единственного преобразования. Эта стратегия полагает, что большинство лексических ошибок вызвано единственным неверным преобразованием (по сути, опечатка), и это предположение подтверждается практикой [1].

2.8. Способы построения лексических анализаторов

Имеется три основных способа реализации лексического анализатора.

1. Использование генератора лексических анализаторов, такого как LEX, для создания лексического анализатора по спецификациям, основанным на регулярных выражениях. В этом случае генератор предоставляет функции для чтения и буферизации ввода.

2. Написание лексического анализатора на подходящем языке программирования с использованием возможностей ввода-вывода этого языка для чтения входной информации.

3. Написание лексического анализатора на языке ассемблера и явное управление процессом чтения входной информации.

Эти методы перечислены в порядке усложнения их реализации. К сожалению, более трудные для реализации подходы часто дают более быстрые лексические анализаторы [3].

LEX – программа для генерации лексических анализаторов. Входной язык содержит описания лексем в терминах регулярных выражений. Результатом работы LEX является программа на некотором языке программирования, которая читает входной файл (обычно это стандартный ввод) и выделяет из него последовательности символов (лексемы), соответствующие заданным регулярным выражениям.

Принцип работы LEX достаточно прост: на вход ей подается текстовый файл, содержащий описания нужных лексем в терминах регулярных выражений, а на выходе получается файл с текстом исходной программы-сканера, на заданном языке программирования (обычно – на C). Текст исходной программы-сканера может быть дополнен вызовами любых функций из любых библиотек, поддерживаемых данным языком и системой программирования. Таким образом, LEX позволяет значительно упростить разработку лексических анализаторов, практически сводя эту работу к описанию требуемых лексем в терминах регулярных выражений.

Вопросы

1. Что представляет собой входной алфавит следующих языков программирования: Pascal, C, Lisp?

2. Какие существуют соглашения по использованию пробелов в каждом из языков, указанных в пункте 1?

3. Какую роль выполняет лексический анализ в процессе компиляции?

4. Как могут быть связаны между собой лексический и синтаксический анализ?

5. Почему большинство современных компиляторов используют фазу лексического анализа, притом что она является необязательной?

3. ОПРЕДЕЛЕНИЕ ЛЕКСЕМ

В определении лексем важную роль играют регулярные выражения. Каждый шаблон соответствует множеству строк, так что регулярные выражения можно рассматривать как имена таких множеств.

3.1. Строки и языки

Алфавит, или *класс символов*, обозначает любое конечное множество символов. Типичным примером символов могут служить буквы или алфавитно-цифровые символы. Множество {0, 1} представляет собой *бинарный алфавит*. ASCII является примером компьютерного алфавита.

Строка над некоторым алфавитом – это конечная последовательность символов, взятых из алфавита. В теории языков термины «*предложение*» и «*слово*» часто используются как символы термина «строка». Длина строки *s*, обычно обозначаемая как $|s|$, равна количеству символов в строке. Например, длина строки banana равна шести.

Пустая строка, обозначаемая как λ (или ε), представляет собой специальную строку нулевой длины.

Язык обозначает произвольное множество строк над некоторым фиксированным алфавитом (например, {λ}-множество, содержащее только пустую строку).

3.2. Операции над языками

Имеется ряд важных операций, выполняемых над языками. С точки зрения лексического анализа особое значение имеют объединение, конкатенация и замыкание (см. табл. 3). Можно также обобщить оператор возведения в степень для языка, определив L^0 как {ε}, а L^i – как $L^{i-1}L$. Таким образом, L^i представляет собой L, конкатенированный с самим собой i раз [3].

Таблица 3

Определения операций над языками

Операция	Определение
Объединение L и M – $L \cup M$	$L \cup M$ $\{s \mid s \in L$ или $s \in M\}$
Конкатенация L и M – LM	LM $\{s \mid s \in L$ и $s \in M\}$
Замыкание Клини или итерация L – L^*	L^* – нуль или более конкатенаций L
Позитивное замыкание L – L^+	L^+ – одна или более конкатенаций L

Пример 3. Пусть L – множество (A, B, …, Z, a, b, …, z), а M – множество (0, 1, 2, …, 9). Можно рассматривать множества L и M двояко. С одной стороны, L представляет собой алфавит, состоящий из набора прописных и строчных букв, а M – алфавит, состоящий из цифр. С другой стороны, поскольку символ можно рассматривать как строку единичной длины, множества L и M представляют собой конечные языки. Вот несколько примеров новых языков, созданных из L и M с применением операторов.

1. $L \cup M$ представляет собой множество букв и цифр.
2. LM – множество строк, состоящих из букв, за которыми следует цифра.
3. L^4 – множество всех четырехбуквенных строк.
4. L^* – множество всех строк из букв, включая пустую строку ε.
5. $L(L \cup M)^*$ – множество всех строк из букв и цифр, начинающихся с буквы.
6. M^+ – множество всех строк из одной или нескольких цифр.

3.3. Регулярные выражения

Множество, определенное в п. 5 примера 3, описывает идентификаторы языка Pascal и представляет собой букву, за которой следует нуль или несколько букв или цифр. Представленный способ записи называется *регулярным выражением*, который позволяет точно определять подобные множества. С помощью этого способа записи можно определить идентификаторы Pascal как

letter (letter | digit) *

Вертикальная черта означает «или», скобки используются для группирования подвыражений, а непосредственное соседство **letter** с оставшейся частью выражения означает конкатенацию.

Регулярное выражение строится из более простых регулярных выражений с использованием набора правил. Каждое регулярное выражение r обозначает или задает язык $L(r)$.

Правила, которые определяют *регулярные выражения над алфавитом A*.

1. λ (или ε) представляет собой регулярное выражение, обозначающее { λ }, т.е. множество, содержащее пустую строку.

2. Если a является символом из A, то a – регулярное выражение, обозначающее $\{a\}$, т.е. множество, содержащее строку a. Хотя мы используем одну и ту же запись, технически регулярное выражение a отличается от строки a. О чем идет речь (о регулярном выражении, строке или символе) – становится понятно из контекста.

3. Если r и s – регулярные выражения, обозначающие языки $L(r)$ и $L(s)$, тогда:

– $(r)\,|\,(s)$ представляет собой регулярное выражение, обозначающее $L(r) \cup L(s)$;

– $(r)(s)$ – регулярное выражение, обозначающее $L(r)$ и $L(s)$;

– $(r)^*$ – регулярное выражение, обозначающее $(L(r))^*$;

– (r) – регулярное выражение, обозначающее $L(r)$.

Язык, задаваемый регулярным выражением, называется *регулярным множеством*.

Лишние скобки в регулярном выражении могут быть устранены, если принять следующие соглашения.

1. Унарный оператор * имеет высший приоритет и левоассоциативен.

2. Конкатенация имеет второй по значимости приоритет и левоассоциативна.

3. | (объединение) имеет низший приоритет и левоассоциативно.

При этих соглашениях запись $(a)\,|\,((b)^*(c))$ эквивалентна $a\,|\,b^*c$. Оба выражения обозначают множества строк, которые представляют собой либо единичный a, либо нуль, либо несколько b, за которыми следует единственный c [3].

Пример 4. Пусть $A = (a,b)$.

1. Регулярное выражение $a \mid b$ обозначает множество $\{a, b\}$.

2. Регулярное выражение $(a \mid b)\ (a \mid b)$ обозначает $\{aa, ab, ba, bb\}$, множество всех строк из a и b длины 2.

3. Другое регулярное выражение для того же множества – $aa \mid ab \mid ba \mid bb$.

4. Регулярное выражение a^* – множество всех строк из нуля или более a, т.е. $\{\varepsilon, a, aa, aaa, \ldots\}$.

5. Регулярное выражение $(a \mid b)^*$ обозначает множество всех строк, содержащих нуль или несколько экземпляров a и b, т.е. множество всех строк из a и b. Другое регулярное выражение для этого множества – $(a^*b^*)^*$.

6. Регулярное выражение $a \mid a^*\, b$ – множество, содержащее строку a и все строки, состоящие из нуля или нескольких a, за которыми следует b.

Если два регулярных выражения r и s задают один и тот же язык, то r и s называются *эквивалентными*, т.е. $r = s$. Например, $(a \mid b) = (b \mid a)$.

Имеется ряд алгебраических законов, используемых для преобразования регулярных выражений в эквивалентные. В табл. 4 приведены некоторые из этих законов для регулярных выражений r, s и t.

Таблица 4

Алгебраические свойства регулярных выражений

Аксиома	Описание
$r \mid s = s \mid r$	Оператор | коммутативен
$r \mid (s \mid t) = (r \mid s) \mid t$	Оператор | ассоциативен
$(r\ s)\ t = r\ (s\ t)$	Конкатенация ассоциативна
$r\ (s \mid t) = r\ s \mid r\ t$ $(s \mid t)\ r = s\ r \mid t\ r$	Конкатенация дистрибутивна над |
$\lambda\ r = r$ $r\ \lambda = r$	λ является единичным элементом по отношению к конкатенации
$r^* = (r \mid \lambda)^*$	Связь между λ и *
$r^{**} = r^*$	Оператор * идемпотентен

3.4. Регулярные определения

Для удобства записи регулярным выражениям можно давать имена и определять регулярные выражения с использованием этих

имен так, как если бы это были символы. Если A является алфавитом базовых символов, то *регулярное определение* представляет собой последовательность вида:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$D_n \rightarrow r_n$$

где каждая d_i – индивидуальное имя, а каждое r_i – регулярное выражение над символами из $A \cup \{d_1, d_2, \ldots, d_{i-1}\}$, т.е. базовыми символами и уже определенными именами. Ограничивая каждое r_i символами из A и ранее определенными именами, можно построить регулярное выражение над A для любого r_i, заменяя (возможно, неоднократно) имена регулярных выражений обозначенными ими именами. Если r_i использует d_j для некоторого $j \geq i$, то r_i может быть определено рекурсивно и подстановка никогда не завершится.

Для того чтобы отличить имена от символов, имена в регулярных выражениях выделяются полужирным шрифтом.

Пример 5. Множество идентификаторов Pascal представляет собой множество строк из букв и цифр, начинающихся с буквы. Регулярное определение этого множества:

letter → $A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z$
digit → $0 \mid 1 \mid \ldots \mid 9$
Id → **letter (letter** $\mid$ **digit)***

Замечание. Не все языки могут быть описаны регулярными выражениями. Регулярные выражения не могут быть использованы для описания сбалансированных или вложенных конструкций. Например, с одной стороны, множество всех строк из сбалансированных скобок не может быть описано регулярным выражением. С другой стороны, это множество может быть описано посредством контекстно-свободной грамматики.

Регулярные выражения могут использоваться для описания только фиксированного количества повторений данной конструкции.

3.5. Распознавание лексем и регулярные выражения

Рассмотрим пример языка, порождаемого грамматикой.

Пример 6. Рассмотрим следующий фрагмент грамматики:

Stmt	→	**if** *Expr* **then** *Stmt* | **if** *Expr* **then** *Stmt* **else** *Stmt* | λ
Expr	→	*Term* **relop** *Term* | *Term*
Term	→	**id**

где терминалы **if**, **then, else, relop, id** и **num** порождают множество строк, задаваемых следующими регулярными определениями:

if →	*if*
then →	*then*
else →	*else*
relop →	< | <= | = | <> | > | >=
id →	**letter (letter | digit) ***

где **letter** и **digit** определены так же, как и ранее.

Для этого фрагмента языка лексический анализатор будет распознавать ключевые слова *if*, *then*, *else*, а также лексемы, обозначенные **relop, id.** Считаем, что ключевые слова зарезервированы и не могут использоваться в качестве идентификаторов.

Цель – построение лексического анализатора, который сможет выделять из входного потока очередную лексему и, используя табл. 5, выдавать пары, состоящие из значения лексемы и атрибута-значения. Атрибут-значение для операторов отношения определяется символическими константами *LT, LE, EQ, NE, GT, GE*.

Таблица 5

Шаблоны регулярных выражений для лексем

Регулярное выражение	Лексема	Атрибут-значение
ws	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Id	**id**	Указатель на запись в таблице
num	**num**	Указатель на запись в таблице
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

3.6. Диаграммы переходов

В качестве промежуточного шага при создании лексического анализатора можно рассматривать стилизованные блок-схемы, называемые *диаграммами переходов.* Диаграмма переходов изображает действия, выполняемые лексическим анализатором при вызове его синтаксическим анализатором для получения очередной лексемы. Позиции в диаграмме переходов изображаются кружками и называются *состояниями*. Состояния соединены стрелками, называемыми *дугами*. Выходящие из состояния *s* дуги имеют метки, указывающие входные символы, которые могут появиться во входном потоке по достижении состояния *s.* Метка **other** означает появление любого символа, не указанного другими исходящими дугами.

Диаграммы переходов в данном случае детерминированные, т.е. ни один символ не может быть меткой двух исходящих из одного состояния дуг. Одно из состояний имеет метку *start* – это начальное состояние диаграммы переходов в момент начала распознавания лексемы. Некоторые состояния имеют действия, выполняемые при

их достижении. При попадании в некоторое состояние считывается следующий входной символ. Если имеется исходящая дуга с меткой, соответствующей этому символу, перемещаемся по ней в следующее состояние. Если такой дуги нет, то входящий символ некорректен и произошла ошибка.

На рис. 5 показана диаграмма переходов для шаблонов >= и >. Диаграмма работает следующим образом. Работа начинается в состоянии 0, в котором считывается следующий символ из входного потока. Дуга, помеченная символом «>», ведет в состояние 6, если этот символ – «>». В противном случае мы не можем распознать лексику > или >=.

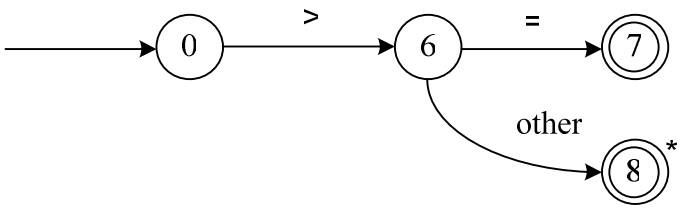


Рис. 5. Диаграмма перехода для лексики >=

По достижении состояния 6 считываем следующий входной символ. Дуга, помеченная символом «=», приводит из состояния 6 в состояние 7, если считанный символ – «=». В противном случае по дуге **other** попадаем в состояние 8. Двойной кружок, который изображает состояние 7, показывает, что это – заключительное состояние, в котором найдена лексема >=.

Символ «>» с другим дополнительным символом приводит в другое допускаемое состояние 8. Поскольку этот другой символ не является частью лексемы, он должен быть возвращен во входной поток, что и указано звездочкой около этого состояния.

Вообще говоря, может существовать ряд диаграмм переходов, каждая из которых определяет группу лексем. Если при перемещении по диаграмме переходов происходит сбой, осуществляется возврат к тому месту, где он был в стартовом состоянии данной диаграммы, и переходим к следующей диаграмме.

Пример 7. Диаграмма переходов для лексемы **relop** показана на рис. 6. Диаграмма на рис. 7 представляет собой часть этой более сложной диаграммы переходов.

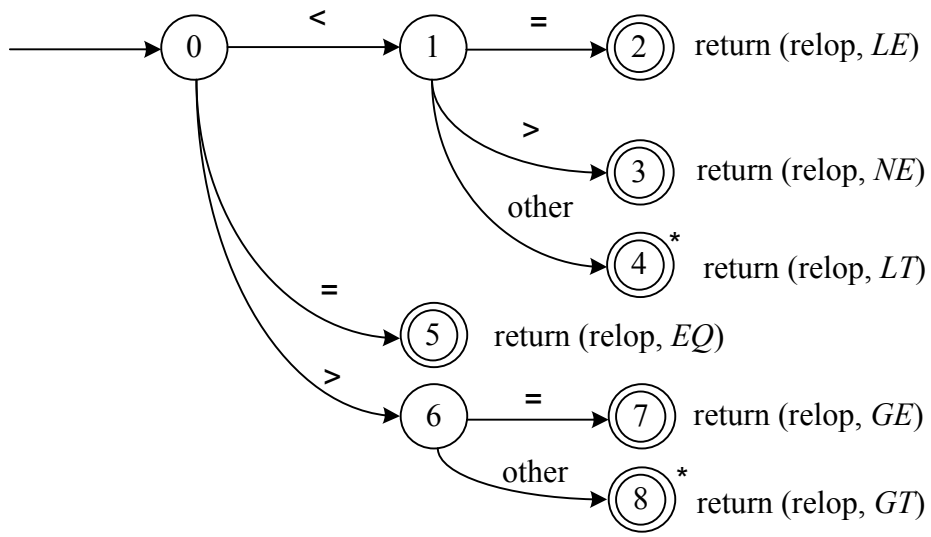


Рис. 6. Диаграмма переходов для операторов отношения

Пример 8. Поскольку ключевые слова – последовательности букв, они представляют собой исключение из правила, гласящего, что последовательность букв и цифр, начинающаяся с буквы, является идентификатором. Но вместо кодирования исключений в диаграмме переходов можно рассматривать ключевые слова как специальные идентификаторы. Тогда по достижении заключительного состояния (рис. 7) выполняется некоторый код, который определяет, чем является лексема, приведшая в это состояние, - ключевым словом или идентификатором.

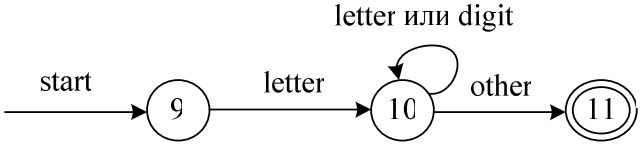


Рис. 7. Диаграмма переходов для идентификаторов и ключевых слов

Простейшая технология отделения ключевых слов от идентификаторов состоит в соответствующей инициализации таблицы символов, в которой хранится информация об идентификаторах. Для лексем *if*, *then* и *else* надо внести в таблицу символов строки *if*, *then*

и *else* до начала работы с входным потоком. При распознавании такой строки возвращается лексема ключевого слова. Программа проверяет таблицу символов, и, если лексема найдена в ней и помечена как ключевое слово, возвращается указатель на запись в таблице символов, в противном случае возвращается лексема **id**.

3.7. Конечные автоматы

Распознавателем языка называется программа, которая получает на входе строку *x* и отвечает «да», если *x* – предложение языка, или в противном случае – «нет». Регулярное выражение компилируется в распознаватель путем построения обобщенной диаграммы переходов, называемой *конечным автоматом*. Такой автомат может быть детерминированным или недетерминированным (недетерминированный автомат может иметь более одного перехода из некоторого состояния при одном и том же входном символе).

Как детерминированные, так и недетерминированные конечные автоматы способны к распознаванию точных регулярных множеств. Таким образом, они могут распознавать все, что могут обозначать регулярные выражения. Однако детерминированные конечные автоматы, которые приводят к более быстрому распознаванию, обычно больше по размеру, чем эквивалентные недетерминированные. Существуют методы преобразования регулярных выражений в оба типа конечных автоматов.

3.7.1. Недетерминированные конечные автоматы

Недетерминированный конечный автомат (*НКА*) $A = (Q, V, \delta, q_0, F)$ представляет собой математическую модель, состоящую:

– из множества *состояний* Q;

– из множества входных символов V (*символов входного алфавита*);

– из функции переходов δ, которая отображает пары символ – состояние на множество состояний;

– из состояния q_0, известного как *стартовое* (*начальное*);

– из множества состояний F, известных как *допускающие* (*конечные*).

НКА может использоваться в виде помеченного ориентированного графа, так называемого *графа переходов*, узлы которого представляют собой состояния, а помеченные дуги составляют функцию переходов. Такой граф похож на диаграмму переходов, однако один и тот же символ может помечать два и более переходов из одного состояния, а некоторые переходы могут быть помечены специальным символом ε, как обычным входным символом (ε-переходы).

Граф переходов для НКА, распознающего язык $(a|b)^*abb$, показан на рис. 8. Множество состояний НКА – $\{0, 1, 2, 3\}$, а входной алфавит – $\{a, b\}$. Состояние 0 стартовое, а заключительное 3 представлено двойным кружком.

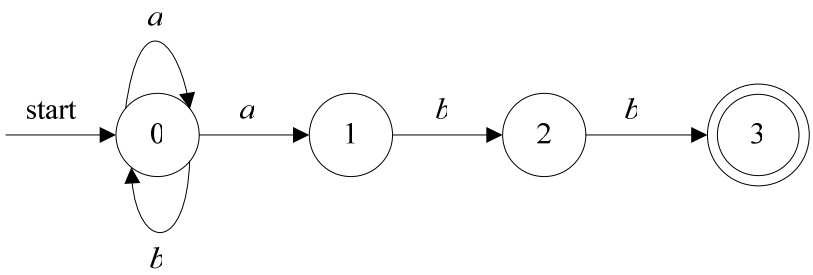


Рис. 8. Недетерминированный конечный автомат

Функция переходов НКА может быть реализована различными способами. Простейший из них – *таблица переходов* (табл. 6), в которой строки представляют состояния, а столбцы — входные символы. Запись в строке *i* для символа *a* является множеством состояний, которые могут быть достигнуты переходом из состояния *i* при входном символе *a*. Переходы для НКА показаны в табл. 6.

Таблица 6

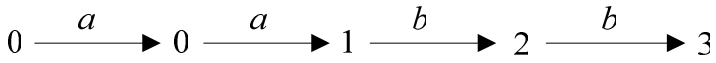
Таблица переходов для конечного автомата (см. рис. 8)

Состояние	Входной символ	
	a	b
0	$\{0, 1\}$	$\{0\}$
1	–	$\{2\}$
2	–	$\{3\}$

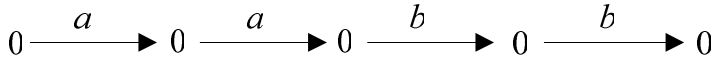
Представление автомата таблицей переходов обеспечивает быстрый доступ к переходам из данного состояния по данному символу. Вместе с тем таблица занимает слишком много места, когда входной алфавит велик и большинство переходов ведут в пустое множество состояний.

НКА *допускает* или *принимает* входную строку *x* (а эта строка является *допустимой*), когда в графе переходов существует некоторый путь от начального состояния к какому-либо из заключительных, такой, что метки дуг этого пути соответствуют строке *x*. На рис. 9 НКА допускает входные строки *abb*, *aabb*, *babb*, *aaabb*, ... Например, *aabb* допускается по пути из 0 вдоль дуги *a* в состояние 0, затем в состояния 1, 2 и 3 вдоль дуг, помеченных соответственно *a*, *b* и *b*.

Путь может быть представлен в виде последовательности переходов состояний, так называемых *перемещений*. Следующая диаграмма показывает перемещения, выполненные для входной строки *aabb*:



Вообще говоря, в заключительное состояние может приводить более чем одна последовательность перемещений. Входная строка *aabb* может быть получена и другими последовательностями перемещений, которые не приводят в заключительное состояние. Например, для той же входной строки *aabb* может быть выполнена следующая последовательность перемещений, оставляющая нас в состоянии 0:



Язык, определяемый НКА, представляет собой множество допускаемых им выходных строк. На рис. 9 НКА допускает строки $(a|b)*abb$.

3.7.2. Детерминированный конечный автомат

Детерминированный конечный автомат (*ДКА*) является специальным случаем недетерминированного конечного автомата, в котором:

– отсутствуют состояния, имеющие λ-переходы;

– для каждого состояния *s* и входного символа *a* существует не более одной дуги, выходящей из *s* и помеченной как *a*.

Для любого входного символа детерминированный конечный автомат имеет не более одного перехода из каждого состояния. Если для представления функции переходов ДКА используется таблица, то каждая запись в ней представляет собой единственное состояние. Следовательно, очень просто проверить, допускает ли данный ДКА некоторую строку, поскольку имеется не более одного пути от стартового состояния, помеченного этой строкой. Следующий алгоритм имитирует поведение ДКА при обработке входной строки.

На рис. 9 показан граф переходов детерминированного конечного автомата, допускающего тот же язык (*a*|*b*)**abb*, что и НКА (см. рис. 8). При работе с этим ДКА и входной строкой *ababb* в алгоритме будет выполнена последовательность состояний 0, 1, 2, 1, 2, 3 и получен ответ «да».

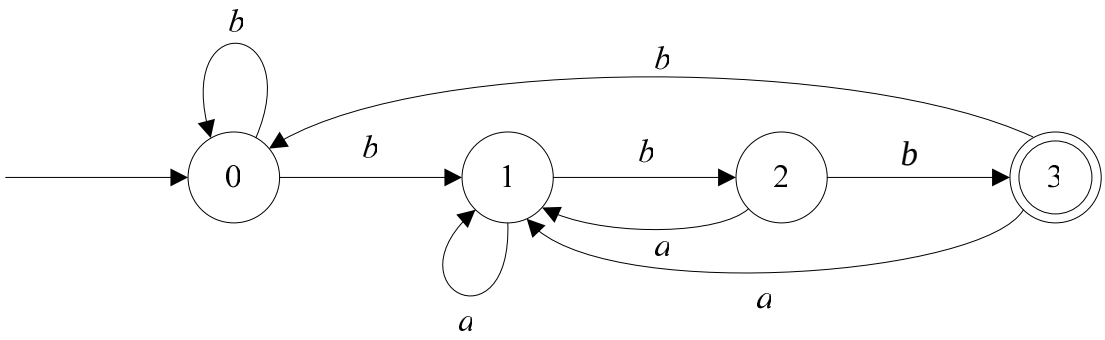


Рис. 9. ДКА, допускающий строку (*a*|*b*) * *abb*

3.7.3. Преобразования НКА

При построении лексического анализатора, над НКА необходимо выполнить ряд преобразований.

Например, на рис. 9 НКА имеет два перехода из состояния 0 для входного символа a – в состояние 0 или 1. Ситуации, в которых функция переходов многозначна, делают моделирование НКА с помощью компьютерной программы весьма сложной задачей. Определение допустимости утверждает только, что должен существовать некоторый путь, помеченный входной строкой и ведущий от начального состояния к заключительному. Однако когда имеется много путей для одной и той же входной строки, возможно, придется рассматривать их все, чтобы найти путь к заключительному состоянию или выяснить, что такого пути не существует.

Алгоритм *преобразования НКА в ДКА*, распознающий тот же язык, что и НКА, является алгоритмом *построения подмножеств* и может использоваться при моделировании НКА компьютерной программой.

В табл. 6 каждая запись представляет собой множество состояний; в табл. 7 – единственное состояние. Общая идея преобразования НКА в ДКА состоит в том, что каждое состояние ДКА соответствует множеству состояний НКА. ДКА использует свои состояния для отслеживания всех возможных состояний, в которых НКА может находиться после чтения очередного входного символа. Таким образом, после чтения входного потока $a_1, a_2, ..., a_n$ ДКА находится в состоянии, которое представляет собой подмножество состояний НКА, достижимых из стартового состояния НКА по пути, помеченному как $a_1, a_2, ..., a_n$. Количество состояний ДКА может оказаться экспоненциально зависящим от количества состояний НКА.

На рис. 10 показан еще один НКА, допускающий язык $(a|b)^*abb$. Этот автомат получен с использованием алгоритма *построения НКА по регулярному выражению*.

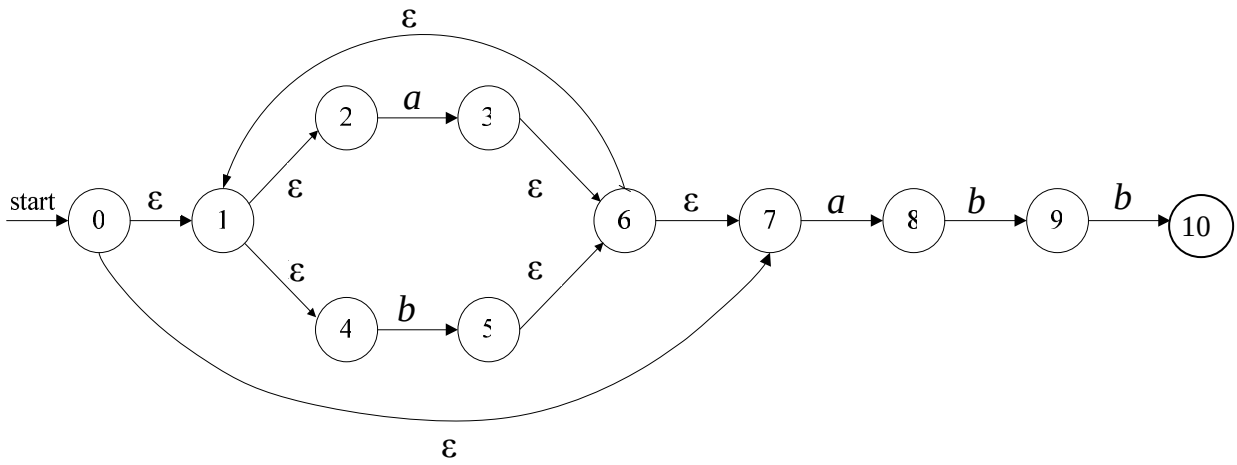


Рис. 10. НКА для $(a|b)^*abb$

Преобразование НКА в ДКА путем построения подмножеств дает пять различных множеств состояний:

$A = \{0, 1, 2, 4, 7\}$; $D = \{1, 2, 4, 5, 6, 7, 9\}$;
$B = \{1, 2, 3, 4, 6, 7, 8\}$; $E = \{1, 2, 4, 5, 6, 7, 10\}$.
$C = \{1, 2, 4, 5, 6, 7\}$;

Состояние A является начальным, а E – единственным заключительным состоянием. Переходы ДКА показаны в табл. 7.

Таблица 7

Переходы ДКА

Состояние	Входной символ	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*E*
E	*B*	*C*

Граф переходов, полученный в результате преобразований ДКА, показан на рис. 11. Следует заметить, что ДКА, представленный на рис. 9, также допускает язык $(a \mid b)^* abb$ и имеет на одно состояние меньше.

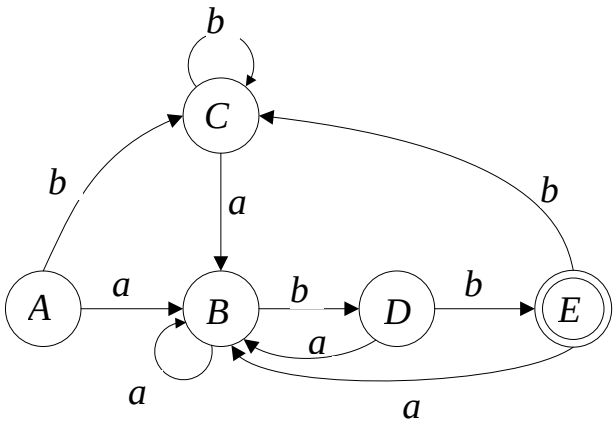


Рис. 11. Диаграмма переходов ДКА

Далее необходимо минимизировать количество состояний ДКА.

Алгоритм *минимизации количества состояний ДКА* работает путем поиска всех групп состояний, различимых посредством некоторой входной строки. Каждая группа состояний, которые не отличаются друг от друга, сливается в одно состояние. В алгоритме используется метод работы с разбиениями множеств состояний. Каждая группа состояний в разбиении содержит состояния, которые еще не были отличными друг от друга, и в процессе работы выбираются пары состояний из разных групп.

Изначально разбиение состоит из двух групп: заключительные состояния и незаключительные состояния. Основной шаг состоит в том, чтобы взять некоторую группу символов, скажем $A = \{S_1, S_2, ..., S_k\}$ и входной символ a и посмотреть, какие переходы имеют состояния $\{S_1, S_2, ..., S_k\}$ для этого входного символа. Если эти переходы представляют собой переходы в состояния, попадающие в две или более различные группы текущего разбиения, группу A разбиваем на подгруппы так, чтобы для каждого из подмножеств переходы ограничивались одной группой текущего разбиения.

Процесс разбиения повторяется до тех пор, пока не останется групп, которые следует разбить. Переходы детерминированного конечного автомата с минимальным числом состояний представлены в табл. 8.

Таблица 8

Переходы оптимизированного автомата

Состояние	Входной символ	
	a	*b*
A	*B*	*A*
B	*B*	*D*
D	*B*	*E*
E	*B*	*A*

3.7.4. Построение конечного автомата по регулярной грамматике

Конечный автомат является хорошей математической моделью для представления алгоритмов распознавания лексем в лексическом анализаторе. При этом источником, по которому строится конечный автомат, является регулярная грамматика (а не регулярное выражение). →

Если задана регулярная грамматика $G = <N, T, P, S>$, правила вывода которой имеют вид: $A \rightarrow aB$ или $A \rightarrow a$, где $A, B \in N$, а $\in T$. Тогда конечный автомат $A = < V, Q, \delta, q_0, F >$, задающий тот же самый язык, что порождает регулярная грамматика G, строится следующим образом:

1) $V = T$;

2) $Q = N \cup \{Z\}$, Z – заключительное состояние КА, не принадлежит N и T;

3) $q_0 = \{S\}$;

4) $F = \{Z\}$;

5) отображение δ строится по правилам:

– каждому правилу подстановки в грамматике G вида $A \rightarrow aB$ ставится в соответствие команда $(A, a) \rightarrow B$;

– каждому правилу подстановки в грамматике G вида $A \rightarrow a$ ставится в соответствие команда $(A, a) \rightarrow Z$.

Допустим *обратный переход*: конечному автомату $A = < V, Q, \delta, q_0, F >$ можно поставить в соответствие регулярную грамматику $G = < N, T, P, S >$, у которой:

1) $T = V$;

2) $N = Q$;

3) $S = \{ q_0 \}$;

4) множество правил подстановки P строится таким образом: каждой команде автомата $(q_i, a) \rightarrow q_k$ ставится в соответствие правило подстановки $q_i \rightarrow a\, q_k$, если $q_k \in Q$, либо $q_i \rightarrow a$, если $q_k \in Z$, где Z – заключительное состояние.

Вопросы

1. Что называется регулярным выражением?
2. Какая существует взаимосвязь между регулярным выражением и распознаваемой лексемой?
3. Что такое регулярное множество?
4. Как можно доказать эквивалентность двух регулярных выражений?
5. Как можно доказать эквивалентность двух языков?

4. ФОРМАЛЬНЫЕ ЯЗЫКИ И ГРАММАТИКИ

4.1. Цепочки символов. Операции над цепочками символов

Цепочка символов – это произвольная последовательность символов, записанных один за другим. Понятие *символа* (или буквы) является базовым в теории формальных языков.

Цепочки символов обозначаются греческими буквами: α, β, γ.

Цепочка – это последовательность, в которую могут входить любые допустимые символы. Цепочка – это необязательно некоторая осмысленная последовательность символов. Последовательность «*аввв…аагрьь, …, лл*» – тоже пример цепочки символов.

Для цепочки символов важен состав, количество и порядок символов в ней. Один и тот же символ может произвольное число раз входить в цепочку. Поэтому цепочки «а» и «аа», а также «аб» и «ба» – это различные цепочки символов. Цепочки символов α и β равны (совпадают), α = β, если они имеют один и тот же состав символов, одно и то же их количество и одинаковый порядок следования символов в цепочке.

Количество символов в цепочке называется *длиной цепочки*. Длина цепочки символа α обозначается как |α|. Очевидно, что если α = β, то и |α| = |β|.

Основной операцией над цепочками символов является операция конкатенации (объединения или сложения) цепочек.

Конкатенация (сложение, объединение) двух цепочек символов – дописывание второй цепочки в конец первой. Конкатенация цепочек α и β обозначается как αβ. Выполнить конкатенацию цепочек просто: например, если α = *аб*, а β = *вг*, то αβ = *абвг*.

Так как в цепочке важен порядок символов, то операция конкатенации не обладает свойством коммутативности, т.е. в общем случае α и β такие, что αβ ≠ βα Также очевидно, что конкатенация обладает свойством ассоциативности, т.е. (αβ)γ = α(βγ).

Еще одна операция – *итерация* (повторение) цепочки *n* раз, где $n \in N$, $n > 0$ – это конкатенация цепочки самой с собой *n* раз. Итерация цепочки α *n* раз обозначается α^n. Для операции повторения справедливы следующие равенства : $\alpha^1 = \alpha$, $\alpha^2 = \alpha\alpha$, $\alpha^3 = \alpha\alpha\alpha$, … и т.д.

Среди всех цепочек символов выделяется одна особенная – пустая цепочка. *Пустая цепочка символов* – это цепочка, не содержащая ни одного символа. Пустую цепочку обозначают греческой буквой λ (в литературе ее иногда обозначают латинской буквой *е* или греческой ε).

Для пустой цепочки справедливы следующие равенства:

1. |λ| = 0;
2. λα: = αλ = α;
3. $\lambda_1 = \lambda$;
4. При $n \geq 0$: $\lambda^n = \lambda$;
5. $\alpha^0 = \lambda$.

4.2. Понятие языка. Формальное определение языка

В общем случае язык – это заданный набор символов и правил, устанавливающих способы комбинации этих символов между собой для записи осмысленных текстов. Основой любого естествен-

ного или искусственного языка является алфавит, определяющий набор допустимых символов языка.

Алфавит – это счетное множество допустимых символов языка. Это множество символов обозначают *V*. Согласно формальному определению, алфавит не обязательно должен быть конечным (перечислимым) множеством, но реально все существующие языки строятся на основе конечных алфавитов.

Цепочка символов α является цепочкой над алфавитом *V*: α (*V*), если в нее входят только символы, принадлежащие множеству символов *V*. Для любого алфавита *V* пустая цепочка λ может как являться, так и не являться цепочкой λ (*V*). Это условие оговаривается дополнительно.

Если *V* – некоторый алфавит, то:

V^+ – множество всех цепочек над алфавитом *V* без λ;

*V** – множество всех цепочек над алфавитом *V*, включая λ.

Справедливо равенство: $V^* = V^+ \cup \{\lambda\}$.

Языком L над алфавитом *V*: *L*(*V*) называется некоторое счетное подмножество цепочек конечной длины из множества всех цепочек над алфавитом *V*. Из этого определения следуют два вывода: во-первых, множество цепочек языка не обязано быть конечным; во-вторых, хотя каждая цепочка символов, входящая в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Все существующие языки попадают под это определение. Большинство реальных естественных и искусственных языков содержат бесконечное множество цепочек. Также в большинстве языков длина цепочки ничем не ограничена. Цепочку символов, принадлежащую заданному языку, называют *предложением языка*, а множество цепочек символов некоторого языка – *L*(*V*)-множеством предложений этого языка.

Два языка *L*(*V*) и *L′*(*V*) совпадают (эквивалентны): *L′*(*V*) = *L*(*V*), если $L'(V) \in L(V)$ и $L(V) \in L'(V)$.

Множества допустимых цепочек символов для эквивалентных языков должны быть равны.

Два языка $L(V)$ и $L'(V)$ почти эквивалентны, если $L'(V) \cup \{\lambda\} = L(V) \cup \{\lambda\}$. Множества допустимых цепочек символов почти эквивалентных языков могут различаться только на пустую цепочку символов.

4.3. Способы задания языков

Итак, каждый язык – это множество цепочек символов над некоторым алфавитом. Но кроме алфавита язык предусматривает и задание правил построения допустимых цепочек, так как не все цепочки над заданным алфавитом принадлежат языку. Символы могут объединяться в слова или лексемы – элементарные конструкции языка, на их основе строятся предложения – более сложные конструкции. И те и другие в общем виде являются цепочками символов и предусматривают некоторые правила построения. Таким образом, необходимо указать эти правила или, строго говоря, задать язык.

Язык можно задать тремя способами:

1. Перечислением всех допустимых цепочек языка.
2. Указанием способа порождения цепочек языка (заданием грамматики языка).
3. Определением метода распознавания цепочек языка.

Первый из методов является чисто формальным и на практике не применяется, так как большинство языков содержит бесконечное число допустимых цепочек и перечислить их просто невозможно.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда любая цепочка, построенная с помощью этих правил из символов и алфавита языка, будет принадлежать заданному языку.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) – автомата, который на входе получает цепочку символов, а на выходе выдает ответ: принадлежит или нет эта цепочка заданному языку [1].

4.4. Синтаксис и семантика языка

Говоря о любом языке, можно выделить синтаксис и семантику. Кроме того, трансляторы имеют дело также с лексическими конструкциями (лексемами), которые задаются лексикой языка.

Синтаксис языка – это набор правил, определяющий допустимые конструкции языка. Синтаксис определяет «форму языка» - задает набор цепочек символов, которые принадлежат языку. Чаще всего синтаксис языка можно задать в виде строгого набора правил, но полностью это утверждение справедливо только для чисто формальных языков.

Семантика языка – это раздел языка, определяющий значения предложений языка. Семантика определяет «содержание языка» - задает значения для всех допустимых цепочек языка. Семантика для большинства языков определяется неформальными методами.

Лексика – это совокупность слов (словарный запас) языка. Слово, или лексическая единица языка, – это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Иначе говоря, лексическая единица может содержать только элементарные символы и не может содержать других лексических единиц.

Лексическими единицами русского языка являются слова русского языка, а знаки препинания и пробелы представляют собой разделители, не образующие лексем. Лексическими единицами алгебры являются числа, знаки математических операций, обозначения функций и неизвестных величин. В языках программирования лексическими единицами являются ключевые слова, идентификаторы, константы, метки, знаки операций; в них также существуют и разделители (запятые, скобки, точки с запятой и т. д.)

4.5. Особенности языков программирования

Языки программирования занимают некоторое промежуточное положение между формальными и естественными языками. С формальными языками их объединяют строгие синтаксические прави-

ла, на основе которых строятся предложения языка. От языков естественного общения в языки программирования перешли лексические единицы, представляющие собой основные ключевые слова.

Для задания языка программирования необходимо решить три вопроса:

– определить множество допустимых символов языка;

– определить множество правильных программ языка;

– задать смысл для каждой правильной программы.

Только первые два вопроса полностью или частично удается решить с помощью теории формальных языков.

Первый вопрос решается легко. Определяя алфавит языка, мы автоматически определяем множество допустимых символов. Для языков программирования алфавит – это чаще всего тот набор символов, который можно ввести с клавиатуры. Основу его составляет младшая половина таблицы международной кодировки символов (таблицы ASCII), к которой добавляются символы алфавитов.

Второй вопрос решается в теории формальных языков только частично. Для всех языков программирования существуют правила, определяющие синтаксис языка, но их недостаточно для того, чтобы строго определить все возможные синтаксические конструкции. Дополнительные ограничения накладываются семантикой языка. Эти ограничения оговариваются для каждого отдельного языка программирования. К таким ограничениям можно отнести необходимость предварительного описания переменных и функций, необходимость соответствия типов переменных и констант в выражениях, формальных и фактических параметров в вызовах функций и др.

Отсюда следует, что практически все языки программирования, строго говоря, не являются формальными языками. И именно поэтому во всех трансляторах, корме синтаксического и анализа предложений языка, дополнительно предусмотрен семантический анализ.

Третий вопрос не относится к теории формальных языков, поскольку такие языки лишены какого-либо смысла. Для ответа на

него нужно использовать другие подходы. В качестве таких подходов можно указать следующие:

– изложить смысл программы, написанной на языке программирования, на другом языке, более понятном тому, кому адресована программа;

– использовать для проверки смысла некоторую «идеальную машину», которая предназначена для выполнения программ, написанных на данном языке.

Использование первого подхода – комментарии в хорошей программе – это и есть изложение ее смысла. Построение блок-схемы, а также любое другое описание алгоритма программы – это тоже способ изложить смысл программы на другом языке (например, языке графических символов – блок-схем алгоритмов, смысл которого, в свою очередь, изложен в соответствующем ГОСТе). Документация к программе – тоже способ изложения ее смысла. Но все эти способы ориентированы на человека, которому они более понятны. Однако пока не существует универсального способа проверить, насколько описание соответствует программе.

Машина же понимает только один язык – язык машинных команд. Но изложить программу на языке машинных команд – задача слишком трудоемкая для человека, как раз для ее решения и создаются трансляторы.

Второй подход используется при отладке программы. Оценку результатов выполнения программы при отладке выполняет человек. Любые попытки доверить это дело машине лишены смысла вне контекста решаемой задачи.

Например, предложение в программе на языке Pascal вида

$$i\text{: = 0; while } i = 0 \text{ do } i\text{: = 0;}$$

может быть легко оценено любой машиной как бессмысленное. Но если необходимо обеспечить взаимодействие с другой параллельно выполняемой программой или, например, просто проверить надежность и долговечность процессора или какой-то ячейки памяти, то это предложение уже не лишено смысла.

Осмысление исходной программы закладывает в компилятор его создатель (или коллектив создателей), т.е. человек, который руководствуется неформальными методами (чаще всего описанием входного языка). В теории формальных языков вопрос о смысле программ не решается.

Поэтому большинство трансляторов обнаруживает только незначительный процент от общего числа смысловых (семантических) ошибок, а следовательно, подавляющее число такого рода ошибок всегда, к большому сожалению, остается на совести автора программы.

4.6. Понятие о грамматике языка

Грамматика – это описание способа построения предложений некоторого языка, или *грамматика* – это математическая система, определяющая язык.

Фактически определение грамматики языка указывает правила порождения цепочек символов, принадлежащих этому языку. Таким образом, *грамматика* – это генератор цепочек языка. Она относится ко второму способу определения языков – порождению цепочек символов.

Грамматику языка можно описать различными способами; например, грамматика русского языка описывается довольно сложным набором правил, которые изучают в начальной школе. Но для многих языков (и для синтаксической части языков программирования в том числе) допустимо использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Правило (или *продукция*) – это упорядоченная пара цепочек символов (αβ). В правилах очень важен порядок цепочек, поэтому их чаще записывают в виде $\alpha \rightarrow \beta$ (или $\alpha ::= \beta$).Такая запись читается как «α порождает β» или «β по определению есть α».

Грамматика языка программирования содержит правила двух типов:

1) первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию;

2) вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме.

Поэтому любое описание языка программирования обычно состоит из двух частей: вначале формально излагаются правила построения синтаксических конструкций, а потом на естественном языке дается описание семантических правил. Естественный язык понятен человеку, пользователю, который будет писать программы на языке программирования; для компилятора же семантические ограничения необходимо излагать в виде алгоритмов проверки правильности программы (речь идет о семантических ограничениях на исходный текст). Такой проверкой в компиляторе занимается семантический анализатор – специально для этого разработанная часть компилятора.

Язык, заданный грамматикой G, обозначается как $L(G)$.

Две грамматики G и G' являются *эквивалентными*, если они определяют один и тот же язык: $L(G) = L(G')$. Две грамматики G и G' называются *почти эквивалентными*, если заданные ими языки различаются не более чем на пустую цепочку символов: $L(G) \cup \{\lambda\} = L(G') \cup \{\lambda\}$.

4.7. Формальное определение грамматики. Форма Бэкуса – Наура

Для полного формального определения грамматики, кроме правил порождения цепочек языка, необходимо задать также алфавит языка.

Формально грамматика G определяется как четверка компонентов $G = < T, N, P, S >$, где T – множество терминальных символов; N – множество нетерминальных символов, $N \cap T = \varnothing$; P – множество правил (продукций) грамматики вида $\alpha \rightarrow \beta$, где $\alpha \in V$; $\beta \in V^*$; S – стартовый (начальный) символ грамматики (или аксиома грамматики), $S \in N$.

Множество $V = N \cup T$ – *полный алфавит* грамматики G.

Множество терминальных символов T содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как

правило, символы из множества T встречаются только в цепочках правых частей правил, если же они встречаются в цепочке левой части правила, то обязаны быть и в цепочке его правой части.

Множество нетерминальных символов N содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой части правил грамматики, но он обязан хотя бы один раз быть в левой части хотя бы одного правила. Правила грамматики строятся так, чтобы в левой части каждого правила был хотя бы один нетерминальный символ.

Эти два множества не пересекаются: каждый символ может быть либо терминальным, либо нетерминальным. Ни один символ в алфавите грамматики не может быть нетерминальным и терминальным одновременно.

Стартовый символ грамматики – это всегда нетерминальный символ.

P – конечное множество правил грамматики, т.е. цепочек вида $\varphi \rightarrow \psi$ (*правила подстановки* или *продукции*, при этом φ, ψ – цепочки в словаре $V = N \cup T$ и $\varphi \in (T \cup N)^*$, $\psi \in (N \cup T)^*$. Множество правил подстановки P называют также *схемой грамматики*. Цепочка, стоящая в левой части правила грамматики, обязательно содержит хотя бы один нетерминальный символ. В правой же части правила в общем случае может стоять произвольная цепочка из терминальных и нетерминальных символов, включая и пустую цепочку λ.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части, вида: $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, …, $\alpha \rightarrow \beta_n$. Тогда эти правила объединяют вместе и записывают в виде: $\alpha \rightarrow \beta_1|\beta_2|\ldots|\beta_n$. Одной строке в такой записи соответствует сразу n правил.

Форма Бэкуса – Наура предусматривает, что нетерминальные символы записываются в угловых скобках: (< >). Вместо знака «→» в правилах грамматики можно использовать знак «::=».

Ниже приведен пример грамматики для целых десятичных чисел со знаком:

G = ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {<число>, <ч>, <ц>}, P, <число>)

P:
<число> → <ч> | + <ч> | - <ч>
<ч> → <ц> | <ч> <ц>
<ц> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Составляющие элементы грамматики *G*:

– множество терминальных символов *T* содержит двенадцать элементов: десять десятичных цифр и два знака;

– множество нетерминальных символов *N* содержит три элемента: символы <число>, <ч> и <ц>;

– множество правил содержит 15 правил;

– целевым символом грамматики является символ <число>.

Символ <ч> – это нетерминальный символ грамматики, такой же, как и два других. Названия нетерминальных символов не обязаны быть осмысленными. В любой грамматике можно полностью изменить имена всех нетерминальных символов, не меняя при этом заданного грамматикой языка, – точно так же, как в программе Pascal можно изменить имена идентификаторов, и при этом не изменится смысл программы. Для терминальных символов это неверно. Набор терминальных символов всегда строго соответствует алфавиту языка, определяемого грамматикой.

Та же самая грамматика для целых десятичных чисел со знаком, в которой нетерминальные символы обозначены большими латинскими буквами (это соглашение, которое обычно применяется по умолчанию), имеет вид:

$G' = <\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +\}, \{S, T, F\}, P, S>$
P:
$S \rightarrow T \mid + T \mid – T$
$T \rightarrow F \mid TF$
$F \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Здесь изменилось только множество нетерминальных символов. Теперь $N = \{S, T, F\}$. Язык, заданный грамматикой, не изменился – грамматики *G* и G' эквивалентны.

4.8. Принцип рекурсии в правилах грамматики

Особенность формальных грамматик состоит в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил. Грамматика, приведенная в примере для целых десятичных чисел со знаком, определяет бесконечное множество целых чисел с помощью 15 правил.

Возможность пользоваться конечным набором правил достигается в такой форме записи грамматики за счет рекурсивных правил. Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть явной (непосредственной), тогда символ определяется сам через себя в одном правиле, либо неявной (косвенной), тогда то же самое происходит через цепочку правил.

В рассмотренной выше грамматике G непосредственная рекурсия присутствует в правиле: <ч> → <ч> <ц>, а в эквивалентной ей грамматике G' – в правиле: $T \to TF$.

Чтобы рекурсия не была бесконечной, для участвующего в ней нетерминального символа грамматики должны существовать также и другие правила, которые определяют его не через самого себя и позволяют избежать бесконечного рекурсивного определения (в противном случае этот символ в грамматике был бы просто не нужен). Такими правилами являются: <ч> → <ц> – в грамматике G и $T \to F$ – в грамматике G'.

Смысл рекурсии можно пояснить, обращаясь к семантике языка, в рассмотренном выше примере: это язык целых десятичных чисел со знаком. Число – это любая цифра сама по себе. Любые две цифры – это тоже число и т. д. Если строить определение числа таким методом, то оно никогда не будет закончено (в математике разрядность числа ничем не ограничена). Однако можно заметить, что каждый раз, порождая новое число, мы просто дописываем цифру справа к уже написанному ряду цифр. А этот ряд цифр, начиная от одной цифры, тоже, в свою очередь, является числом. Тогда опре-

деление понятия «число» можно дать таким образом: *число* – это любая цифра либо другое число, к которому справа дописана любая цифра. Именно это и составляет основу правил грамматик G и G' и отражено в правилах:

$$< \text{ч}> \rightarrow < \text{ц}> \mid < \text{ч}> < \text{ц}> \quad \text{и} \quad T \rightarrow F \mid TF.$$

Другие правила в этих грамматиках позволяют добавить к числу знак (первая строка правил) и дают определение понятия «цифра» (третья строка правил).

Принцип рекурсии – важное понятие в представлении о формальных грамматиках. Явно или неявно рекурсия всегда присутствует в грамматиках любых реальных языков программирования. Именно она позволяет строить бесконечное множество цепочек языка, и говорить об их порождении невозможно без понимания принципов рекурсии. Как правило, в грамматике реального языка программирования содержится не одно, а целое множество правил, построенных с помощью рекурсии.

4.9. Другие способы задания грамматик

Форма Бекуса – Наура – удобный с формальной точки зрения, но не всегда доступный для понимания способ записи формальных грамматик. Рекурсивные определения хороши для формального анализа цепочек языка, но не удобны с точки зрения человека, так как они не отражают возможности для построения нового слова из уже построенного. Это неочевидно и требует дополнительного пояснения.

При создании языка программирования важно, чтобы его грамматику понимали не только те, кому предстоит создавать компиляторы для этого языка, но и пользователи языка – будущие разработчики программ.

Достаточно распространенные способы записи правил грамматики: с использованием метасимволов; в графическом виде.

4.10. Запись правил грамматик с использованием метасимволов

Запись правил грамматик с использованием метасимволов предполагает, что в строке правил грамматики могут встречаться специальные символы – символы, которые имеют особый смысл и трактуются специальным образом. В качестве таких метасимволов используются: ( ) (круглые скобки), [ ] (квадратные скобки), {} (фигурные скобки), «,» (запятая) и “ ”(кавычки).

Эти метасимволы имеют следующий смысл:

– круглые скобки означают, что из всех перечисленных внутри цепочек символов в данном месте правила грамматики может стоять только одна цепочка;

– квадратные скобки означают, что указанная в них цепочка может встречаться, а может и не встречаться в данном месте правила грамматики (т.е. может быть в нем один раз или не быть ни одного раза);

– фигурные скобки означают, что указанная внутри них цепочка может не встречаться в данном месте правила грамматики ни одного раза, встречаться один раз или сколь угодно много раз;

– запятая служит для того, чтобы разделять цепочки символов внутри круглых скобок;

– кавычки используются в тех случаях, когда один из метасимволов нужно включить в цепочку обычным образом, т.е. когда одна из скобок или запятая должны присутствовать в цепочке символов языка (если саму кавычку нужно включить в цепочку символов, то ее надо повторить дважды).

Вот так должны выглядеть правила рассмотренной выше грамматики *G*, если их записать с использованием метасимволов:

<число> → [(+, –)] < цифра>{<цифра>}

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Первое правило читается так: число есть цепочка символов, которая может начинаться с символа «+» или «–», должна содержать дальше одну цифру, за которой может идти последовательность из любого количества цифр. В отличие от формы Бэкуса – Наура, в форме, записанной с помощью метасимволов, убран из грамматики малопонятный нетерминальный символ <ч>, полностью исключена рекурсия. Таким образом, грамматика стала более понятной.

Форма записи правил с использованием метасимволов – это удобный и понятный способ представления правил грамматик. Она во многих случаях позволяет полностью избавиться от рекурсии, заменив ее символом интеграции {} (фигурные скобки). Эта форма наиболее употребительна для одного из типов грамматик – регулярных грамматик [1].

4.11. Запись правил грамматик в графическом виде

При записи правил в графическом виде вся грамматика представляется в форме набора специальным образом построенных диаграмм. Эта форма впервые была предложена при описании грамматики языка Pascal. Она доступна не для всех типов грамматик, а только для контекстно-свободных и регулярных типов, но этого достаточно, чтобы ее можно было использовать для описания грамматик известных языков программирования.

В такой форме записи каждому нетерминальному символу грамматики соответствует диаграмма, построенная в виде направленного графа. Граф имеет следующие типы вершин:

– точка входа (на диаграмме никак не обозначена, из нее просто начинается входная дуга графа);

– нетерминальный символ (на диаграмме обозначается прямоугольником, в который вписано обозначение символа);

– цепочка терминальных символов (на диаграмме обозначается овалом, кругом или прямоугольником с закругленными краями, внутрь которого вписана цепочка);

– узловая точка (на диаграмме обозначается жирной точкой или закрашенным кружком);

– точка выхода (на диаграмме никак не обозначена, в нее просто входит выходная дуга графа).

Каждая диаграмма имеет только одну точку входа или выхода, но сколько угодно вершин других трех типов. Вершины соединяются между собой направленными дугами графа (линиями со стрелками). Из входной точки дуги могут только выходить, а во входную точку – только входить. В остальные вершины дуги могут как входить, так и выходить (в правильно построенной грамматике каждая вершина должна иметь как минимум один вход и как минимум один выход).

Чтобы построить цепочку символов, соответствующую какому-либо нетерминальному символу грамматике, надо рассмотреть диаграмму для этого символа. Тогда, начав движение от точки входа, надо двигаться по дугам графа диаграммы через любые вершины до точки выхода. При прохождении через узловые точки диаграммы над результирующей цепочкой никаких действий выполнять не надо. Через любую вершину графа диаграммы, в зависимости от возможного пути движения, можно пройти один раз, ни разу или много раз. При попадании в точку выхода диаграммы построение результирующей цепочки заканчивается.

Результирующая цепочка может содержать нетерминальные символы. Чтобы заменить их на цепочки терминальных символов, нужно рассмотреть соответствующие им диаграммы. И так до тех пор, пока в цепочке не останутся только терминальные символы. Очевидно, чтобы построить цепочку символов заданного языка, надо начать рассмотрение с диаграммы стартового символа грамматики.

Описание понятия «число» из грамматики G с помощью диаграмм представлено на рис. 12.

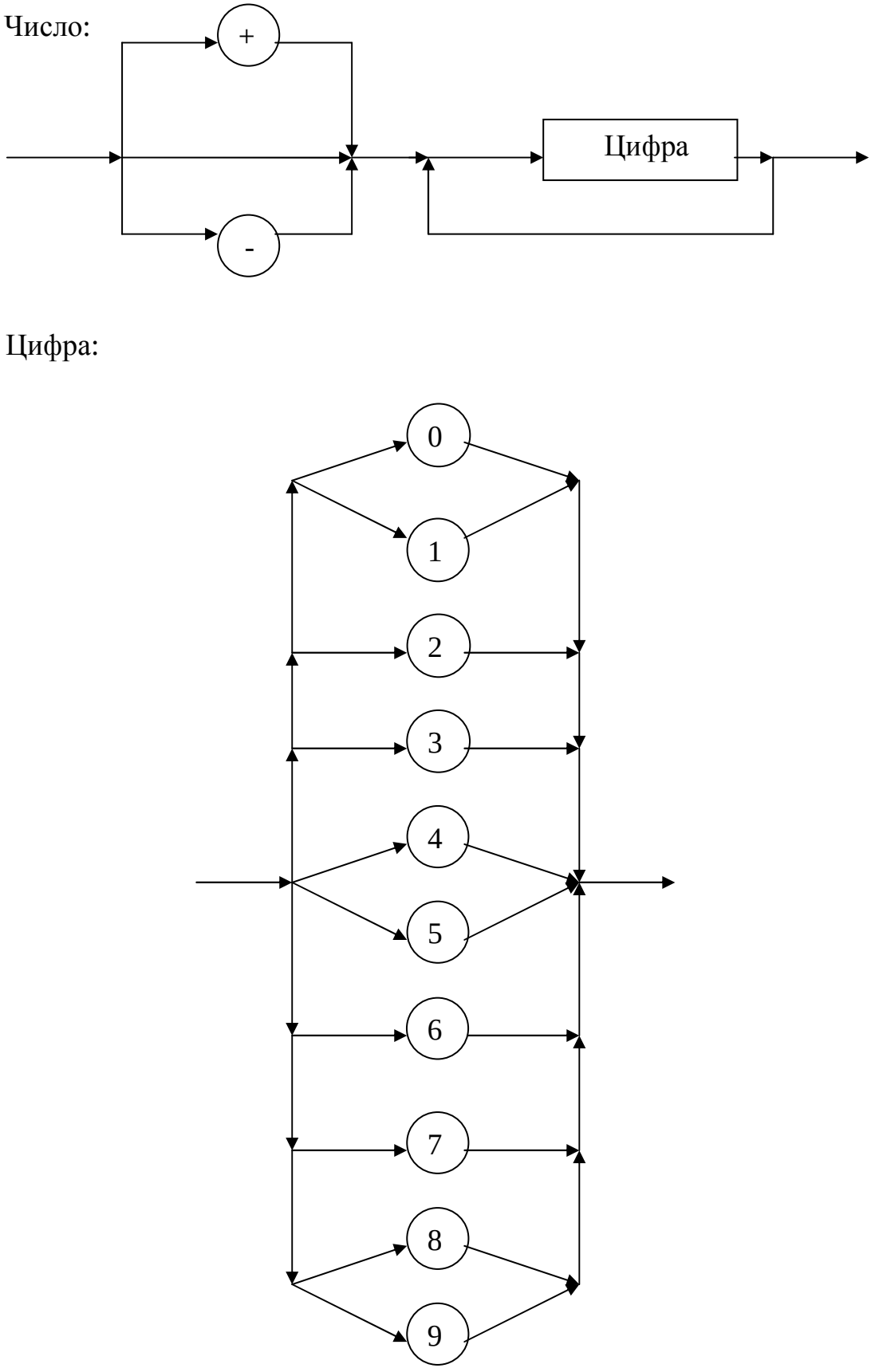


Рис. 18. Графическое представление грамматики целых десятичных чисел со знаком: вверху – для понятия «число»; внизу – для понятия «цифра»

Данный способ в основном применяется в литературе при изложении грамматик языков программирования. Для разработчиков программ он удобен, но практического применения в компиляторах пока не имеет.

4.12. Классификация языков и грамматик

Выше уже упоминались различные типы грамматик, но не указывалось, как и по какому принципу они разделяются на типы. Для человека языки бывают простые и сложные, но это сугубо субъективное мнение, которое часто зависит от личности человека.

Для компиляторов языки также можно разделить на простые и сложные, но в данном случае существуют жесткие критерии для этого разделения. От того, к какому типу относится тот или иной язык программирования, зависит сложность распознавателя для этого языка. Чем сложнее язык, тем выше вычислительные затраты компилятора на анализ цепочек исходной программы, написанной на этом языке, а следовательно, сложнее сам компилятор и его структура. Для некоторых типов языков в принципе невозможно построить компилятор, который анализировал бы исходные тексты на этих языках за приемлемое время на основе ограниченных вычислительных ресурсов (именно поэтому до сих пор не созданы программы на естественных языках, например на русском или английском).

4.12.1. Классификация грамматик по Хомскому

Формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то ее относят к определенному типу. В грамматике достаточно иметь одно правило, не удовлетворяющее требованию структуры, и она уже не попадает в заданный тип.

По классификации Хомского выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой. На структуру их

правил не накладывается никаких ограничений: для грамматики вида $G = <T, N, P, S>$, $V = N \cup T$ правила имеют вид: $\alpha \rightarrow \beta$, где $\alpha \in V^+$, $\beta \in V^*$.

Это самый общий тип грамматик. В него попадают все без исключения формальные грамматики, но часть из них может быть также отнесена и к другим классификационным типам. Грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре.

Грамматики, относящиеся только к типу 0, не имеют практического применения.

Тип 1: контекстно-зависимые и неукорачивающие грамматики. В этот тип входят два основных класса грамматик.

Контекстно-зависимые (КЗ) грамматики $G = <T, N, P, S>$, $V = N \cup T$ имеют правила вида: $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in V^*$, $A \in N$, $\beta \in V^+$.

Структура правил КЗ-грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменен на ту или иную цепочку символов в зависимости от контекста, в котором он встречается. Именно поэтому эти грамматики называются *контекстно-зависимыми*. Цепочки α_1 и α_2 в правилах грамматики обозначают контекст (α_1 – левый контекст, а α_2 – правый контекст), в общем случае любая из них (или даже обе) может быть пустой. Значение одного и того же символа может быть различным в зависимости от того, в каком контексте он встречается.

Неукорачивающие грамматики $G = <T, N, P, S>$, $V = N \cup T$ имеют правила вида: $\alpha \rightarrow \beta$, где $\alpha, \beta \in V^+$, $|\beta| \geq |\alpha|$.

Неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена на цепочку символов не меньшей длины. Отсюда и название «неукорачивающие».

Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного контекстно-зависимой грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык, и, наоборот, для любого языка,

заданного неукорачивающей грамматикой, можно построить контекстно-зависимую грамматику, которая будет задавать эквивалентный язык.

При построении компиляторов такие грамматики не применяются, поскольку языки программирования, рассматриваемые компиляторами, имеют более простую структуру и могут быть построены с помощью грамматик других типов.

Тип 2: контекстно-свободные грамматики. Контекстно-свободные (КС) грамматики $G = <T, N, P, S>$; $V = N \cup T$ имеют правила вида: $A \to \beta$, где $A \in N$, $\beta \in V^+$. Такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками (потому что в правой части правил этих грамматик всегда должен стоять как минимум один символ).

Существует также почти эквивалентный им класс грамматик – укорачивающие контекстно-свободные (УКС) грамматики $G = <T, N, P, S>$; $V = N \cup T$, правила которых могут иметь вид: $A \to \beta$, где $A \in VN$, $\beta \in V^*$.

Разница между этими двумя классами грамматик заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (λ), а в НКС-грамматиках – нет. Язык, заданный НКС-грамматикой, не может содержать пустой цепочки. Доказано, что эти два класса грамматик почти эквивалентны. В дальнейшем, говоря о КС-грамматиках, не будет уточняться, какой класс грамматик (УКС или НКС) имеется в виду, если возможность наличия в языке пустой цепочке не имеет принципиального значения.

КС-грамматики широко используются при описании синтаксических конструкций языков программирования. Синтаксис большинства известных языков программирования основан именно на КС-грамматиках.

Тип 3: регулярные грамматики. К типу регулярных относятся два эквивалентных класса грамматик: леволинейные и праволинейные.

Леволинейные грамматики $G = <T, N, P, S>$; $V = N \cup T$ могут иметь правила двух видов: $A \to B\gamma$ или $A \to \gamma$, где $A, B \in N$; $\gamma \in VT^*$.

Праволинейные грамматики $G = <T, N, P, S>$; $V = N \cup T$ могут иметь правила тоже двух видов: $A \to \gamma B$ или $A \to \gamma$, где $A, B \in N$; $\gamma \in VT^*$.

Эти два класса грамматик эквивалентны и относятся к типу регулярных грамматик.

Регулярные грамматики используются при описании простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев и т.д. Эти грамматики исключительно просты и удобны в использовании, поэтому в компиляторах на их основе строятся функции лексического анализа входного языка.

Типы грамматик соотносятся между собой особым образом. Из определения типов 2 и 3 видно, что любая регулярная грамматика является КС-грамматикой, а не наоборот. Также очевидно, что любая грамматика может быть отнесена и к типу 0, поскольку он не накладывает никаких ограничений на правила. В то же время существуют укорачивающие КС-грамматики (тип 2), которые не являются ни контекстно-зависимыми, ни неукорачивающими (тип 1), поскольку могут содержать правила вида «$A \to \lambda$», недопустимые в типе 1. В целом сложность грамматики обратно пропорциональна тому максимально возможному номеру типа, к которому может быть отнесена грамматика. Грамматики, которые относятся только к типу 0, являются самыми сложными, а грамматики, которые можно отнести к типу 3, – самыми простыми.

4.12.2. Классификация языков

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Так как один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к различным классификационным типам, то для классификации самого языка среди всех его грамматик всегда выбирается грамматика с максимально возможным классификационным типом. Например, если язык L может быть задан грамматиками G_1 и G_2, относящимся к типу 1 (контекстно-зависимые), грамматикой G_3, относящейся к типу 2 (кон-

текстно-свободный), и грамматикой G_4, относящейся к типу 3 (регулярные), то сам язык должен быть отнесен к типу 3 и является регулярным языком.

От классификационного типа языка зависит не только то, с помощью какой грамматики можно построить предложения этого языка, но и то, насколько сложно распознать эти предложения. Распознать предложения – значит построить распознаватель для языка. Сложность распознавателя языка напрямую зависит от классификационного типа, к которому относится язык.

Сложность языка убывает с возрастанием номера классификационного типа языка. Самыми сложными являются языки типа 0, самыми простыми – языки типа 3. Согласно классификации грамматик, существует также 4 типа языков.

Тип 0: языки с фразовой структурой. Это самые сложные языки, которые могут быть заданы только грамматикой, относящейся к типу 0. Для распознавания цепочек таких языков требуются вычислители, равномощные машине Тьюринга. Поэтому можно сказать, что если язык относится к типу 0, то для него невозможно построить компилятор, который гарантированно выполнял бы разбор предложений языка за ограниченное время на основе ограниченных вычислительных ресурсов.

Практически все естественные языки общения между людьми, строго говоря, относятся именно к этому типу языков. Дело в том, что структура и значение фразы естественного языка может зависеть не только от контекста данной фразы, но и от содержания того текста, в котором эта фраза встречается. Одно и то же слово в естественном языке может не только иметь разный смысл в зависимости от контекста, но и играть различную роль в предложении. Именно поэтому столь велики сложности в автоматизации перевода текстов, написанных на естественных языках, а также отсутствуют компиляторы, которые воспринимали бы программы на основе таких языков.

Тип 1: контекстно-зависимые (КЗ) языки. Тип 1 – второй по сложности тип языков. В общем случае время на распознавание предложений языка, относящегося к типу 1, экспоненциально зависит от длины исходной цепочки символов.

Языки и грамматики, относящиеся к типу 1, применяются в анализе и переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка (но они не учитывают содержание текста, поэтому в общем случае для точного перевода с естественного языка все же требуется вмешательство человека). На основе таких грамматик может выполняться автоматизированный перевод с одного естественного языка на другой, ими могут пользоваться сервисные функции проверки орфографии и правописания в языковых процессорах.

В компиляторах КЗ языки не используются, поскольку языки программирования имеют более простую структуру.

Тип 2: контекстно-свободные (КС) языки. КС-языки лежат в основе синтаксических конструкций большинства современных языков программирования, на их основе функционируют некоторые довольно сложные командные процессоры, допускающие управляющие команды цикла и условия. Эти обстоятельства определяют распространенность данного класса языков.

В общем случае время на распознавание предложений языка, относящегося к типу 1, полиноминально зависит от длины исходной цепочки символов (в зависимости от класса языка это либо кубическая, либо квадратичная зависимость). Однако среди КС-языков существует много классов языков, для которых эта зависимость линейна. Многие языки программирования можно отнести к одному из таких классов.

Тип 3: регулярные языки. Регулярные языки – самый простой тип языков. Время на распознавание предложений регулярного языка линейно зависит от длины исходной цепочки символов.

Регулярные языки лежат в основе простейших конструкций языков программирования (идентификаторов, констант и т.д.), кроме того, на их основе строятся многие мнемокоды машинных команд (языки ассемблеров), а также командные процессоры, символьные управляющие команды и другие подобные структуры.

Регулярные языки – очень удобное средство. Для работы с ними можно использовать регулярные множества и выражения, конечные автоматы [1].

На рис. 13 приведена иерархия языков и соответствующие ей иерархии грамматик и автоматов как распознающих устройств.

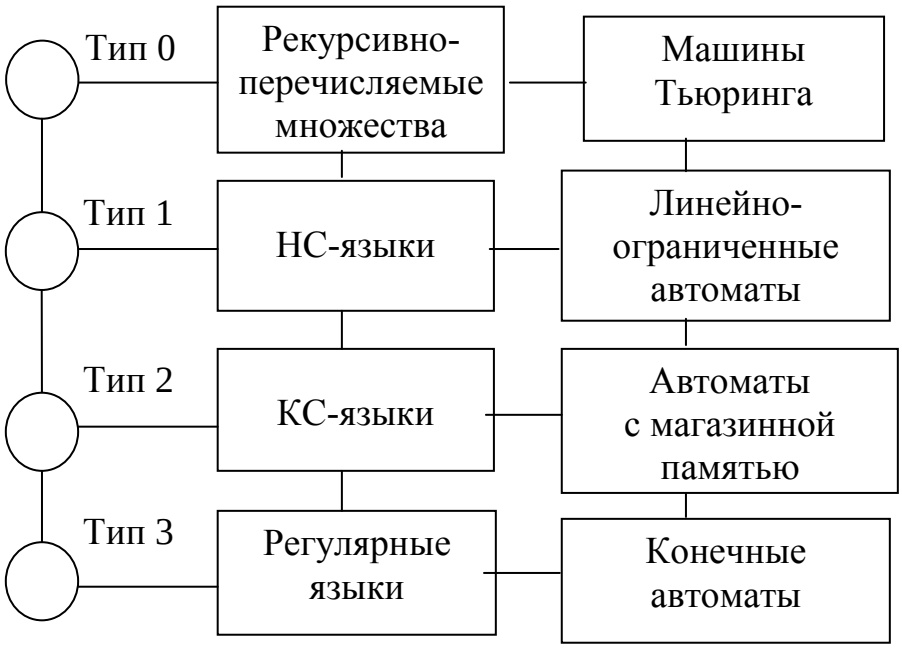


Рис. 13. Иерархия языков, грамматик и автоматов

4.12.3. Примеры классификации языков и грамматик

Классификация языков идет от простого к сложному. Если мы имеем регулярный язык, то можно утверждать, что он также является и контекстно-свободным, и контекстно-зависимым, и даже языком с фразовой структурой. В то же время известно, что существуют КС-языки, которые не являются ни регулярными, ни контекстно-свободными.

Примеры некоторых языков указанных типов

Грамматика для целых десятичных чисел со знаком
$G = <\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +\}, \{S, T, F\}, P, S>$:
P:
$S \to T + T \mid – T$
$T \to F \mid TF$
$F \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

По структуре своих правил данная грамматика G относится к контекстно-свободным грамматикам (тип 2). Ее можно отнести и к типу 0, и к типу 1, но максимально возможным является именно тип 2. К типу 3 эту грамматику отнести нельзя, так как строка $T \to F \mid TF$ содержит правило $T \to TF$, которое недопустимо для типа 3, и, хотя все остальные правила этому типу соответствуют, одного несоответствия достаточно.

Для того же самого языка (целых десятичных чисел со знаком) можно построить и другую грамматику $G' = <\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +\},\{S, T\}, P, S>$:

P:

$S \to T \mid +T \mid -T$

$T \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0\mathrm{T} \mid 1T \mid 2T \mid 3T \mid 4T \mid 5T \mid 6T \mid 7T \mid 8T \mid 9T$

По структуре своих правил грамматика G' является праволинейной и может быть отнесена к регулярным грамматикам (тип 3).

Для этого же языка можно построить эквивалентную леволинейную грамматику (тип 3) $G'' = <\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +\},\{S, T\}, P, S>$:

P:

$T \to + \mid - \mid \lambda$

$S \to T0 \mid T1 \mid T2 \mid T3 \mid 4T \mid T5 \mid T6 \mid T7 \mid T8 \mid T9 \mid S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8 \mid S9$

Следовательно, язык целых десятичных чисел со знаком, заданный грамматиками G, G' и G'', относится к регулярным языкам (тип 3).

Для произвольного языка, заданного некоторой грамматикой, в общем случае довольно сложно определить его тип. Не всегда можно построить грамматику максимально возможного типа для произвольного языка. К тому же при строгом определении типа требуется еще доказать, что две грамматики (первоначально имеющаяся и вновь построенная) эквивалентны, а также то, что не существует для того же языка грамматики с большим по номеру типом. Это нетривиальная задача, которую не так легко решить.

4.13. Цепочки вывода. Сентенциальная форма. Вывод. Цепочки вывода

Вывод – это процесс порождения предложения языка на основе правил, определяющих язык грамматики.

Цепочка $\beta = \delta_1\gamma\,\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике $G = (T, N, P, S)$, $V = T \cup N$, δ_1, γ, $\delta_2 \in V^*$, $\omega \in V^+$, если в грамматике G существует правило: $\omega \to \gamma \in P$. Непосредственная выводимость цепочки β из цепочки α обозначается так: $\alpha \Rightarrow \beta$. То есть цепочка β выводима из цепочки α, если можно взять несколько символов в цепочке α, заменить их на другие символы по некоторому правилу грамматики и получить цепочку β. В формальном определении непосредственной выводимости любая из цепочек δ_1 или δ_2 (или обе цепочки) может быть пустой. В предельном случае вся цепочка α может быть заменена на цепочку β, тогда в грамматике G должно существовать правило: $\alpha \to \beta \in P$.

Цепочка β называется *выводимой* из цепочки α (обозначается $\alpha \overset{*}{\Rightarrow} \beta$) в том случае, если выполняется одно из двух условий:

– β непосредственно выводима из α ($\alpha \Rightarrow \beta$);

– $\exists\ \gamma$ такая, что γ выводима из α и β непосредственно выводима из γ ($\alpha \overset{*}{\Rightarrow} \gamma$ и $\gamma \Rightarrow \beta$).

Это рекурсивное определение выводимости цепочки. Суть его заключается в том, что цепочка β выводима из цепочки α, если $\alpha \Rightarrow \beta$ или же если можно построить последовательность непосредственно выводимых цепочек от α к β следующего вида: $\alpha \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_I \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$, $n \geq 1$. В этой последовательности каждая последующая цепочка γ_i непосредственно выводима из предыдущей цепочки γ_{i-1}.

Такая последовательность непосредственно выводимых цепочек называется *выводом* или *цепочкой вывода*. Каждый переход от одной непосредственно выводимой цепочки к следующей в цепочке вывода называется *шагом вывода*. Очевидно, что шагов вывода в цепочке вывода всегда на один больше, чем промежуточных цепо-

чек. Если цепочка β непосредственно выводима из цепочки α (α ⇒ β), то имеется всего один шаг вывода.

Пример грамматики для целых десятичных чисел со знаком (см. п. 4.12.3)

Построим в ней несколько произвольных цепочек вывода:

1. $S \Rightarrow -T \Rightarrow -TF \Rightarrow -TFF \Rightarrow -FFF \Rightarrow -4FF \Rightarrow -47F \Rightarrow -479$
2. $S \Rightarrow T \Rightarrow TF \Rightarrow T8 \Rightarrow F8 \Rightarrow 18$
3. $T \Rightarrow TF \Rightarrow T0 \Rightarrow TF0 \Rightarrow T50 \Rightarrow F50 \Rightarrow 350$

Получим следующие выводы:

1. $S \overset{*}{\Rightarrow} -479$ или $S \Rightarrow +479$ или $S \Rightarrow 7479$
2. $S \overset{*}{\Rightarrow} 18$ или $S \Rightarrow +18$ или $S \Rightarrow 518$
3. $T \overset{*}{\Rightarrow} 350$ или $T \Rightarrow +350$ или $T \Rightarrow 6350$

Все эти выводы построены на основе грамматики *G*. В принципе в этой грамматике можно построить сколь угодно много цепочек вывода.

4.14. Сентенциальная форма грамматики. Язык, заданный грамматикой

Вывод называется *законченным*, если на основе цепочки β, полученной в результате вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод называется законченным, если цепочка β, полученная в результате вывода, пустая или содержит только терминальные символы грамматики $G = <T, N, P, S>$: $\beta \in T^*$. Цепочка β, полученная в результате законченного вывода, называется *конечной* цепочкой вывода.

В рассмотренном выше примере все построенные выводы являются законченными, а, например, вывод $S \overset{*}{\Rightarrow} -4FF$ (из первой цепочки в примере) будет незаконченным.

Сентенциальная форма грамматики $G = < T, N, P, S >$, $V = T \cup N$ – это цепочка символов $\alpha \in V^*$, выводимая из стартового символа грамматики *S*, $S \overset{*}{\Rightarrow} \alpha$. Если цепочка $\alpha \in T^*$ получена в результате законченного вывода, то она называется *конечной сентенциальной формой*, или *предложением языка*, порождаемого данной грамматикой.

В рассмотренном выше примере цепочки символов «–479» и «18» являются конечными сентенциальными формами грамматики целых десятичных чисел со знаком, так как существуют выводы $S \overset{*}{\Rightarrow} -479$ и $S \overset{*}{\Rightarrow} 18$ (примеры 1 и 2). Цепочка $F8$ из вывода 2 является сентенциальной формой, поскольку справедливо $S \overset{*}{\Rightarrow} F8$, но она не является конечной цепочкой вывода.

Язык L, заданный грамматикой $G = < T, N, P, S >$ – это множество всех конечных сентенциальных форм грамматики *G*. Язык *L*, заданный грамматикой *G*, обозначается как $L(G)$. Очевидно, что алфавитом такого языка $L(G)$ будет множество терминальных символов грамматики *T*, поскольку все конечные сентенциальные формы грамматики – это цепочки над алфавитом *T*.

Две грамматики $G = <T, N, P, S >$ и $G' = < T', N', P', S'>$ называются *эквивалентными*, если эквивалентны заданные ими языки: $L(G) = L(G')$. Эквивалентные грамматики должны иметь пересекающиеся множества терминальных символов $VT \cap VT' \neq \varnothing$ (как правило, эти множества совпадают $VT = VT'$), а вот множества нетерминальных символов, правила грамматики и их стартовые символы могут кардинально отличаться.

4.15. Левосторонний и правосторонний выводы

Левосторонний вывод – вывод, в котором на каждом шаге правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке.

Правосторонний вывод – вывод, в котором на каждом шаге правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

В цепочках вывода из того же примера, вывод 1 является левосторонним, выводы 2, 3 – правосторонними.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний и правосторонний выводы. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

4.16. Дерево вывода. Методы построения дерева вывода

Деревом вывода грамматики $G = < T, N, P, S >$ называется дерево (граф), которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

– каждая вершина дерева обозначается символом грамматики $A \in (T \cup N)$;

– корнем дерева является вершина, обозначенная стартовым символом грамматики – S;

– листьями дерева являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки λ;

– если некоторый узел дерева обозначен символом $A \in N$, а связанные с ним узлы – символами $b_1, b_2, \ldots, b_n$, где $n > 0$, $\forall\ n \geq i > 0$, $b_i \in (T \cup N \cup \{\lambda\})$, то в грамматике $G = < T, N, P, S >$ существует правило $A \rightarrow b_1, b_2, \ldots, b_n \in P$.

Из определения видно, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда (либо же оно будет иметь несколько иной вид).

На основе рассмотренного выше примера деревья вывода для цепочек вывода 1 и 2 приведены на рис. 14.

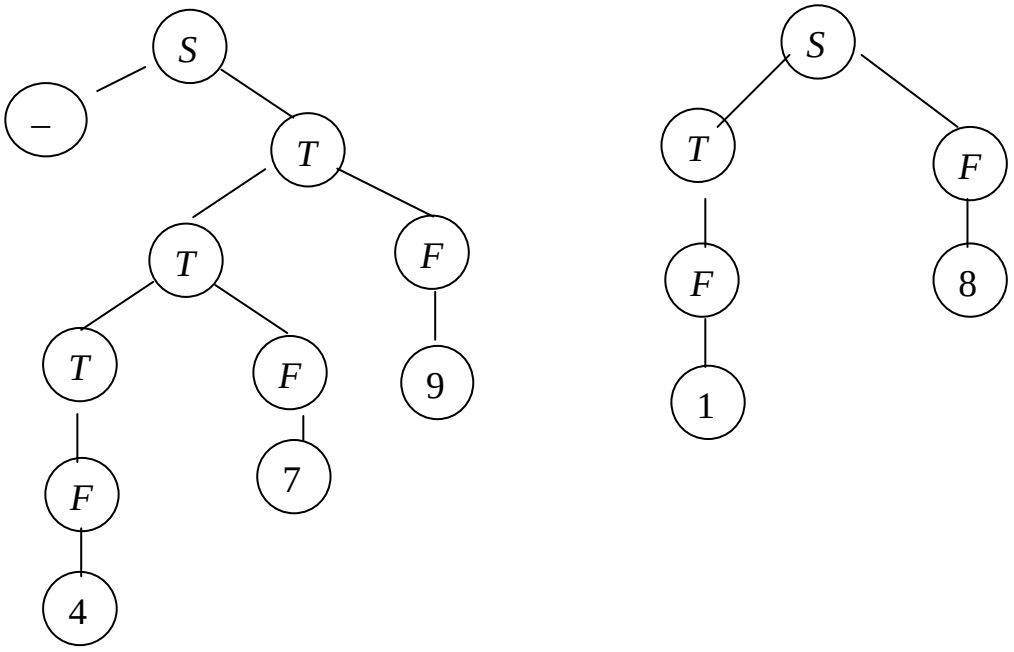


Рис. 14. Примеры деревьев вывода для грамматики целых десятичных чисел со знаком

Для того чтобы построить дерево вывода, достаточно иметь цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определенным выводом: либо левосторонним, либо правосторонним.

Построение дерева вывода сверху вниз начинается со стартового символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева выбирается крайняя вершина (крайняя левая – для левостороннего ввода, крайняя правая – для правостороннего), обозначенная нетерминальным символом. Для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода снизу вверх начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода, которые на первом шаге построения образуют последний уровень дерева. Построение дерева идет по уровням. На втором шаге построения в грамматике выбирается правило, первая часть которого соответствует крайним символам в уровне дерева (крайним правым символам при правостороннем выводе и крайним левым – при левостороннем). Выбранные вершины уровня соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в уровень дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная терминальным символом), а иначе надо вернуться ко второму шагу и повторять его над полученным уровнем дерева.

Поскольку все известные языки программирования имеют нотацию записи «слева направо», компилятор также всегда читает входную программу слева направо (и сверху вниз, если программа

разбита на несколько строк). Поэтому для построения дерева вывода методом «сверху вниз», как правило, используется левосторонний вывод, а для построения «снизу вверх» – правосторонний. Нотация чтения программ «слева направо» влияет не только на порядок разбора программы компилятором, но и на порядок выполнения операций – при отсутствии скобок большинство равноправных операций выполняются в порядке слева направо, что имеет существенное значение.

Вопросы

1. Какие существуют методы задания языков? Сравните их по эффективности.
2. Что такое грамматика языка?
3. Какие существуют формы описания грамматик?
4. Какие типы языков выделяются по классификации Хомского? Как они соотносятся между собой?
5. Что такое сентенциальная форма грамматики? Что является предложением языка?

5. СИНТАКСИЧЕСКИЙ АНАЛИЗ

5.1. Основные принципы работы синтаксического анализатора

Синтаксический анализатор (синтаксический разбор) – это часть компилятора, которая отвечает за выявление основных синтаксических конструкций входного языка. В задачу синтаксического анализа входит: найти и выделить основные синтаксические конструкции в тексте входной программы, установить тип и проверить правильность каждой синтаксической конструкции, наконец, представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

В основе синтаксического анализатора лежит распознаватель текста входной программы на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирова-

ния могут быть описаны с помощью КС-грамматик, реже встречаются языки, которые могут быть описаны с помощью регулярных грамматик. Чаще всего регулярные грамматики применимы к языкам ассемблера, а языки высокого уровня построены на основе синтаксиса КС-языков.

Распознаватель дает ответ на вопрос о том, принадлежит или нет цепочка входных символов заданному языку – это основная задача синтаксического анализатора. Кроме того, синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции всю информацию о найденных и разобранных синтаксических структурах.

Синтаксический разбор — это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора бессмысленна, в то время как лексический разбор в принципе является необязательной фазой. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Лексический анализатор только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем входной программы.

Выходом лексического анализатора является таблица лексем (или цепочка лексем). Эта таблица образует вход синтаксического анализатора, который исследует только один компонент каждой лексемы — ее тип. Остальная информация о лексемах используется на более поздних фазах компиляции при семантическом анализе, подготовке к генерации и генерации кода результирующей программы. Синтаксический анализ (или разбор) — это процесс, в котором исследуется таблица лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами. Примерами конструкций, которые обычно распознаются во

время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору. Обычно это определяет разработчик компилятора, исходя из технологических аспектов программирования, а также из синтаксиса и семантики входного языка.

Основу любого синтаксического анализатора всегда составляет распознаватель, построенный на основе какого-либо класса КС-грамматик. Поэтому главную роль в том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

Каждый язык программирования имеет правила, которые предписывают синтаксическую структуру корректных программ. В Pascal, например, программа состоит из блоков, блок – из инструкций, инструкции – из выражений, выражения – из лексем и т.д. Синтаксис конструкций языка программирования может быть описан с помощью контекстно-свободных грамматик или нотации БНФ (форма Бэкуса – Наура). Грамматики обеспечивают значительные преимущества разработчикам языков программирования и создателям компиляторов:

– грамматика дает точную и при этом простую для понимания синтаксическую спецификацию языка программирования;

– для некоторых классов грамматик можно автоматически построить эффективный синтаксический анализатор, который определяет, корректна ли структура исходной программы. Дополнительным преимуществом автоматического создания анализатора является возможность обнаружения синтаксических неоднозначностей и других сложных для распознавания конструкций языка, которые иначе могли бы остаться не замеченными на начальных фазах создания языка и его компилятора;

– правильно построенная грамматика придает языку програм-

мирования структуру, которая способствует облегчению трансляции исходной программы в объектный код, выявлению ошибок. Для преобразования описаний трансляции, основанных на грамматике языка, в рабочие программы имеется соответствующий программный инструментарий;

– со временем языки эволюционируют, обогащаясь новыми конструкциями, и выполняют новые задачи. Добавление конструкций в язык окажется более простой задачей, если существующая реализация языка основана на его грамматическом описании.

5.2. Роль синтаксического анализатора

В модели компилятора синтаксический анализатор получает строку лексем с выхода лексического анализатора (рис. 15) и проверяет, может ли эта строка порождаться грамматикой исходного языка. Он также сообщает обо всех выявленных ошибках. Кроме того, он должен уметь обрабатывать обычно часто встречающиеся ошибки и продолжать работу с оставшейся частью программы.

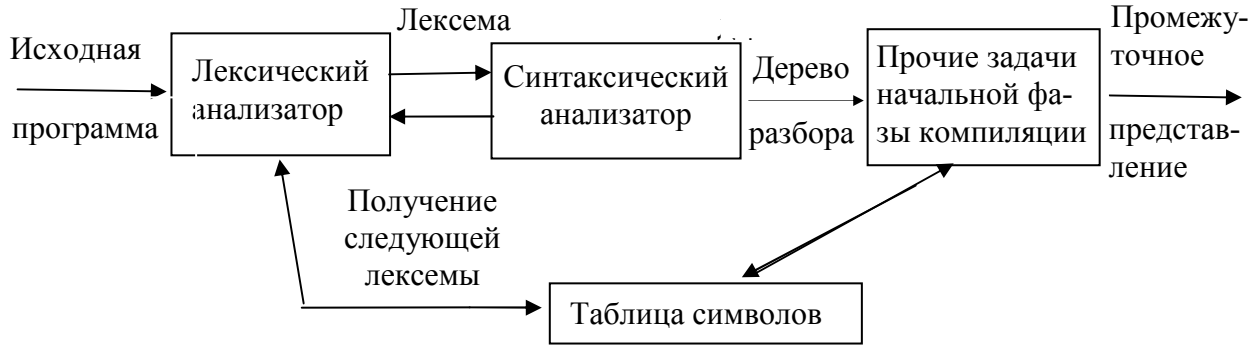


Рис. 15. Место синтаксического анализатора в модели компилятора

Имеется три основных типа синтаксических анализаторов грамматик.

1. Универсальные методы разбора, такие как алгоритмы Кока-Янгера-Касами или Эрли, могут работать с любой грамматикой. Однако эти методы слишком неэффективны для использования в промышленных компиляторах.

2. Нисходящие (сверху вниз) методы синтаксического анализа. Нисходящие синтаксические анализаторы строят дерево разбора сверху (от корня) вниз (к листьям). Входной поток синтаксического анализатора сканируется посимвольно слева направо.

3. Восходящие (снизу вверх) методы синтаксического анализа. Восходящие методы начинают построение дерева разбора с листьев и идут к корню. Входной поток также сканируется посимвольно слева направо.

Восходящие и нисходящие методы синтаксического анализа наиболее распространены в компиляторах.

Наиболее эффективные нисходящие и восходящие методы работают только с подклассами грамматик, однако некоторые из этих подклассов, такие как *LL*- и *LR*-грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования. Реализованные вручную синтаксические анализаторы чаще работают с *LL*-грамматиками. Синтаксические анализаторы для несколько большего класса *LR*-грамматик обычно создаются с помощью автоматизированных инструментов.

Выходом синтаксического анализатора является некоторое представление дерева разбора входного потока лексем, выданного лексическим анализатором. На практике имеется множество задач, которые могут сопровождать процесс разбора, – например, сбор информации о различных лексемах в таблице символов, исполнение проверки типов и других видов семантического анализа, а также создание промежуточного кода.

5.3. Обработка синтаксических ошибок

Если компилятор имеет дело исключительно с корректными программами, его разработка и реализация существенно упрощаются. Однако программисты пишут программы с ошибками, и хороший компилятор должен помочь программисту обнаружить их и локализовать. Большинство спецификаций языков программирования не определяет реакции компилятора на ошибки – этот вопрос отдается на откуп разработчикам компилятора. Однако планиро-

вание системы обработки ошибок с самого начала работы над компилятором может как упростить его структуру, так и улучшить его реакцию на ошибки.

Любая программа потенциально содержит множество ошибок самого разного уровня. Например, ошибки могут быть:

– лексическими (такими как неверно записанные идентификаторы, ключевые слова или операторы);

– синтаксическими (например, арифметические выражения с несбалансированными скобками);

– семантическими (такими как операторы, применяемые к несовместимым с ними операндам);

– логическими (например, бесконечная рекурсия).

Основные действия по выявлению ошибок и восстановлению после них решаются на этапе синтаксического анализа. Одна из причин этого состоит в том, что многие ошибки по природе своей являются синтаксическими или проявляются, когда поток лексем, идущий от лексического анализатора, нарушает определяющие язык программирования грамматические правила. Вторая причина заключается в точности современных методов разбора – они очень эффективно выявляют синтаксические ошибки в программе. Определение присутствия в программе семантических или логических ошибок – задача более сложная.

Обработчик ошибок синтаксического анализатора имеет очень просто формулируемые цели:

– он должен ясно и точно сообщать о наличии ошибок;

– он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжить поиск последующих ошибок.

5.4. Контекстно-свободные грамматики

Многие языки программирования по своей природе имеют рекурсивную структуру, которая может определяться контекстно-свободными грамматиками. Например, условная инструкция определяется следующим правилом.

Если S_1 и S_2 являются инструкциями, а E – выражением, то

$$\text{if } E \text{ then } S_1 \text{ else } S_2 \qquad (1)$$

является инструкцией

Этот тип условных инструкций не может быть определен с использованием регулярных выражений. Используя для обозначения класса инструкций синтаксическую переменную **stmt**, а для класса выражений – **expr**, можно выразить (1) с помощью продукции грамматики

$$\textbf{stmt} \rightarrow \text{if } \textbf{expr} \text{ then } \textbf{stmt} \text{ else } \textbf{stmt} \qquad (2)$$

Определение контекстно-свободной грамматики включает в себя понятия терминалов, нетерминалов, стартового символа и продукций.

1. *Терминалы* представляют собой базовые символы, из которых формируются строки. В (2) каждое из ключевых слов if, then, else является терминалом.

2. *Нетерминалы* представляют собой синтаксические переменные, которые обозначают множества строк. В (2) **stmt** и **expr** являются нетерминалами. Нетерминалы определяют множества строк, которые помогают в определении языка, порождаемого грамматикой. Кроме того, они налагают на язык иерархическую структуру, облегчающую синтаксический анализ и трансляцию.

3. Один из нетерминалов грамматики считается *стартовым символом*, и множество строк, которые он обозначает, являются языком, определяемым грамматикой.

4. *Продукции грамматики* определяют способ, которым терминалы и нетерминалы могут объединяться для создания строк. Каждая продукция состоит из нетерминала, за которым следуют стрелка (или символ : : =, если грамматика записана в форме Бэкуса – Наура) и строка нетерминалов и терминалов.

5. Общий вид правил: $A \rightarrow \beta$, где $A \in N$, $\beta \in V^+$.

Пример 9. Грамматика со следующими продукциями определяет простые арифметические выражения.

expr → expr op expr
expr → (expr)
expr → – expr
expr → id
op → +| – | * | / | ↑

В этой грамматике терминальными символами являются:

id, + , – , * , / , ↑, ( )

Нетерминальные символы грамматики – **expr** и **op**; стартовым символом грамматики является **expr.**

Соглашения по обозначениям при записи грамматик:
символы являются терминалами:
1) строчные буквы из начала алфавита, такие как *a*, *b*, *c*;
2) символы операторов, такие как +, – и т.п.;
3) символы пунктуации, такие как запятые, скобки и т.п.;
4) цифры 0, 1, …, 9;
5) строки, выделенные полужирным шрифтом, такие как **id** или **if**;
символы являются нетерминалами:
1) прописные буквы из начала алфавита, такие как *A*, *B*, *C*;
2) буква *S*, которая обычно означает стартовый символ;
3) имена из строчных букв, выделенные жирным шрифтом, такие как **stmt** или **expr**.

5. Строчные греческие буквы, такие как α, β представляют собой строки грамматических символов. Таким образом, в общем виде продукция может быть записана как $A \to \alpha$, в которой одиночный нетерминал A располагается слева от стрелки (в левой части продукции), а строка грамматических символов α – справа от стрелки (в правой части продукции).

6. Если $A \to \alpha_1$, $A \to \alpha_2$, ..., $A \to \alpha_k$ представляют собой продукции с A в левой части, то можно записать $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$. Такие правила называются *альтернативами A*.

Пример 10. Используя приведенные соглашения, можно записать грамматику из примера (2) в следующем виде:

$E \longrightarrow E\ A\ E \mid (E) \mid -E \mid \mathbf{id}$
$A \longrightarrow + \mid - \mid * \mid / \mid \uparrow \mid$

Из соглашений следует, что E и A – нетерминалы, причем E является стартовым символом. Остальные символы представляют собой терминалы.

5.5. Порождение

Существует несколько способов рассматривать процесс определения языка грамматикой: один – процесс построения деревьев разбора, другой способ, который дает точное описание нисходящего построения дерева разбора, – порождение. Основная идея состоит в том, что продукция рассматривается как переписывающее правило, в котором нетерминал из левой части замещается строкой из правой части продукции.

Рассмотрим, например, следующую грамматику арифметических выражений.

$$E \longrightarrow E+E \mid E*E \mid (E) \mid -E \mid id \qquad (3)$$

Продукция $E \longrightarrow -E$ означает, что выражение, которому предшествует знак минус, также является выражением. Эта продукция может использоваться для порождения более сложных выражений из простых, позволяя заменять любой экземпляр E на $-E$. В простейшем случае можно заместить одно E на $-E$ и описать это как E => $-E$. Такая запись читается как «E порождает $-E$» или «из E выводится $-E$». Продукция $E \longrightarrow (E)$ говорит, что можно заменить один экземпляр E в любой строке грамматики на (E). Например: $E * E$ => $(E) * E$ или $E * E$ => $E * (E)$.

Можно взять нетерминал E и неоднократно применять продукции в произвольном порядке для получения последовательности замещений. Например: E => $-E$ => $-(E)$ => $-(id)$. Такая последова-

тельность замен представляет *порождение* (*id*) из *E*. (Символ => означает «порождает за один шаг». Символ $\overset{*}{\Rightarrow}$ означает «порождает за нуль или более шагов». Символ $\overset{+}{\Rightarrow}$ используется для обозначения «порождает за один или более шагов».)

Предложение языка *L*(*G*), порождаемого грамматикой *G*, представляет собой сентенциальную форму без нетерминалов.

Например, строка –(*id* + *id*) является предложением грамматики (3), так как существует следующее порождение:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id) \qquad (4)$$

Строки *E*, –*E*, – (*E*), ..., – (*id* + *id*), появляющиеся в процессе порождения, представляют собой сентенциальные формы данной грамматики. Для указания того, что – (*id* + *id*) может быть выведено из *E*, можно записать $E \overset{*}{\Rightarrow} -(id + id)$.

На каждом шаге порождения осуществляется два выбора: во-первых, выбор заменяемого нетерминала; во-вторых, его альтернативы. Например, порождение (4) из примера 11 может продолжиться после –(*E* + *E*) следующим образом:

$$-(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id) \qquad (5)$$

Каждый нетерминал в (5) замещается той же правой частью, что и в (4), но в другом порядке.

5.6. Деревья разбора и приведения

Дерево разбора рассматривается как графическое представление порождения, из которого удалена информация о порядке замещения. Каждый внутренний узел дерева разбора помечается некоторым нетерминалом *A*, а узлы слева направо – символами из правой части продукции для этого нетерминала. Листья дерева разбора помечены нетерминалами или терминалами и, будучи прочитаны слева направо, образуют сентенциальную форму. Например, дерево разбора для строки – (*id* + *id*), полученное порождением (4), показано на рис. 16.

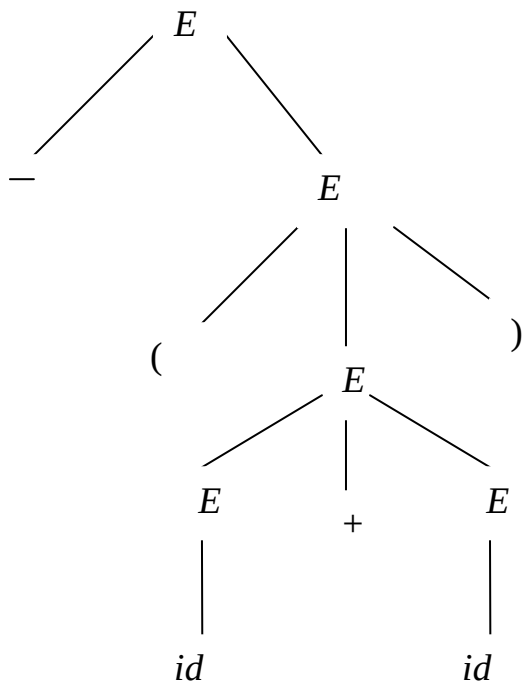


Рис. 16. Дерево разбора для – (*id* + *id*)

Пример 11. Рассмотрим порождение (4). Последовательность деревьев разбора, построенная на основе этого порождения, показана на рис. 17.

Первый шаг этого порождения представляет собой $E \Rightarrow -E$. Для моделирования этого шага добавляем к корню начального дерева два дочерних узла, помещенных как «–» и «E».

Второй шаг представляет собой $-E \Rightarrow -(E)$. Соответственно, добавляем три дочерних узла – «(», «*E*» и «)» – к листу *E* во втором дереве для получения третьего дерева, дающего – (*E*). Продолжая построения, получим шестое дерево в качестве полного дерева разбора.

Дерево разбора игнорирует порядок, в котором производилось замещение символов в сентенциальной форме. Например, если порождение (4) изменить в соответствии с (5), окончательное дерево разбора будет таким же, как на рис. 17.

Предложение может иметь не одно дерево разбора и даже не одно левое или правое порождение.

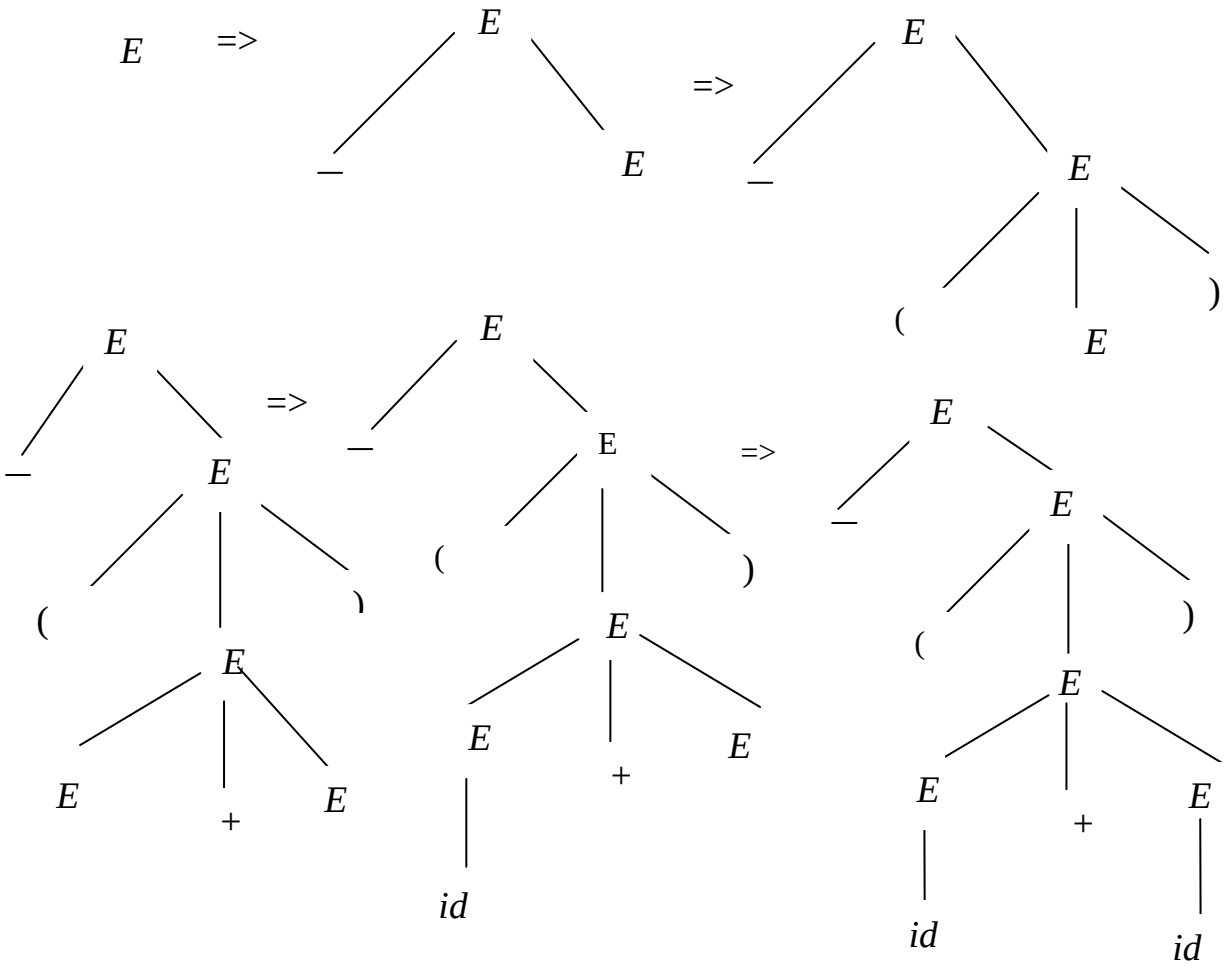
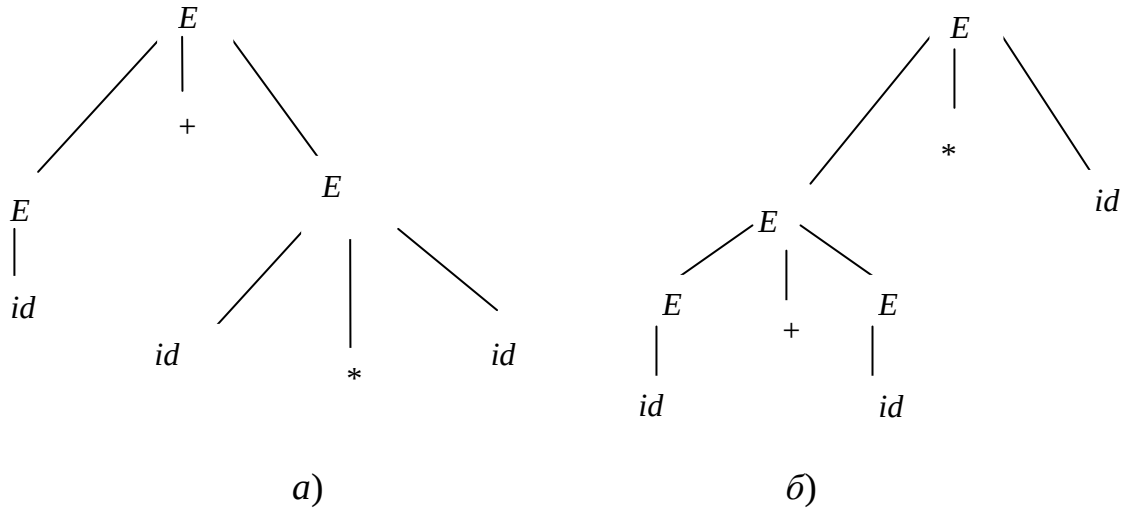


Рис. 17. Построение дерева разбора из порождения (4)

Пример 12. Обратимся вновь к грамматике арифметических выражений (3). Предложение *id + id * id* имеет два разных левых порождения

E => E + E	*E => E * E*
=> id + E	*E => E + E * E*
*=> id + E * E*	*=> id + E * E*
*=> id + id * E*	*=> id + id * E*
*=> id + id * id*	*=> id + id * id*

с двумя соответствующими им деревьями разбора (рис. 18).



а) *б*)

Рис. 18. Два дерева разбора для *id* + *id* * *id*

В отличие от дерева на рис. 18, *б*, дерево разбора на рис. 18, *а* отражает обычные приоритеты операций + и *. Обычно приоритет умножения выше, и выражение типа *a* + *b* * *c* трактуется как *a* + (*b* * *c*), а не как (*a* + *b*) * *c*.

5.7. Неоднозначность грамматик. Устранение неоднозначности

Грамматика, которая дает более одного дерева разбора для некоторого предложения, называется *неоднозначной*.

Неоднозначная грамматика – это та, которая для одного и того же предложения дает не менее двух левых или двух правых порождений. Для большинства типов синтаксических анализаторов грамматика должна быть однозначной, поскольку, если это не так, нельзя определить дерево разбора для предложения единственным образом.

Для устранения неоднозначности грамматика может быть переписана. В качестве примера устраним неоднозначность из следующей грамматики:

stmt → if **expr** then **stmt**
| if **expr** then **stmt** else **stmt** other
| other (6)

Здесь «other» означает все прочие инструкции. Грамматика (6) неоднозначна, поскольку строка

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \tag{7}$$

имеет два дерева разбора (рис. 19).

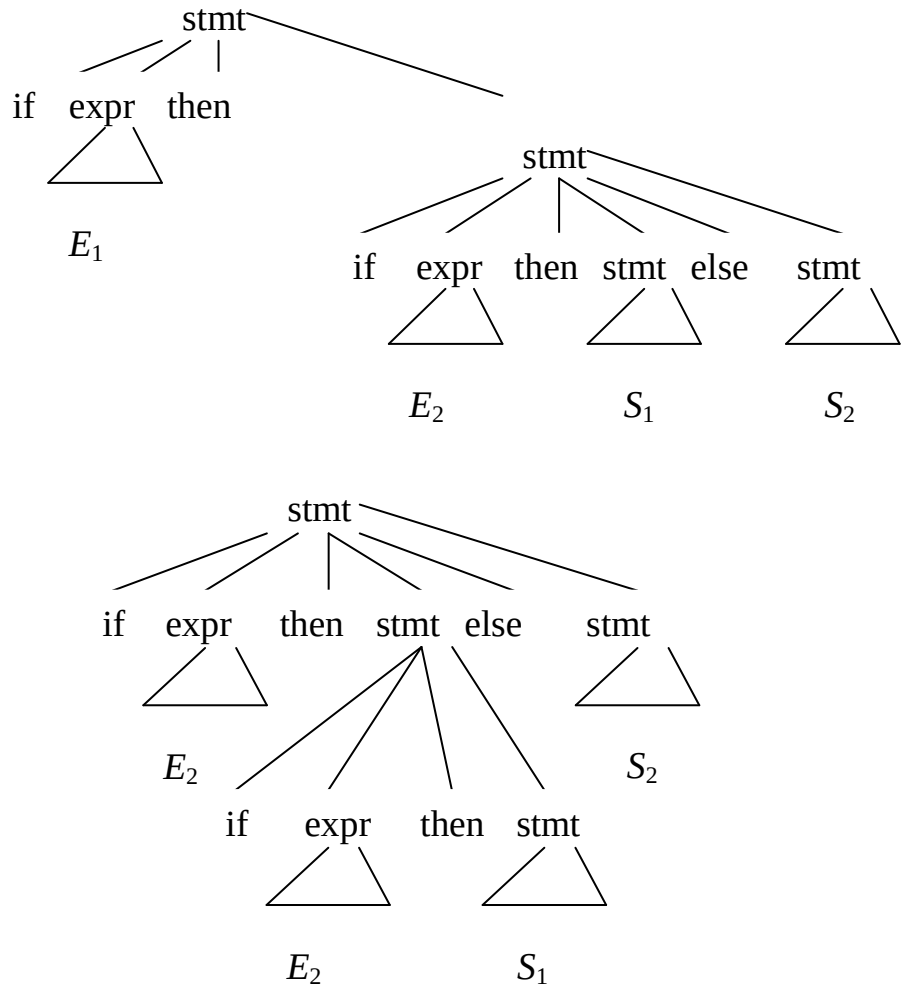


Рис. 19. Два дерева разбора для неоднозначного предложения

Во всех языках программирования с условными инструкциями такого вида предпочтительно первое дерево разбора. Общее правило гласит: «сопоставить каждое else ближайшему незанятому then».

Это правило устранения неоднозначности может быть встроено непосредственно в грамматику. Например, можно переписать грамматику, рассмотренную ранее как однозначную. Основная идея заключается в том, что инструкция, появляющаяся между then и else, должна быть «сбалансированная», т.е. не должна оканчиваться на then, не соответствующее некоторому else. Такая «сбалансированная» инструкция может представлять собой либо полную инструкцию if-then-else, содержащую только «соответствующие» инструкции, либо любые инструкции, отличающиеся от условной, т.е. получаем грамматику.

stmt → S_1 | S_2
S_1 → if **expr** then S_1 else S_1 | other
S_2 → if **expr** then **stmt** | if **expr** then S_1 else S_2

Эта грамматика генерирует то же множество строк, что и грамматика (6), но для строки (7) дает только одно дерево разбора, а именно то, которое связывает каждое else с ближайшим незанятым then.

5.8. Устранение левой рекурсии

Грамматика является леворекурсивной, если в ней имеется нетерминал A такой, что существует порождение A => $A\alpha$ для некоторой строки α. Методы нисходящего разбора не в состоянии работать с леворекурсивными грамматиками, поэтому требуется преобразование грамматики, которое устранило бы из нее левую рекурсию. Рассмотрим общий случай. Леворекурсивная пара продукций $A \rightarrow A\alpha \mid \beta$ может быть заменена нелеворекурсивными продукциями

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \lambda$$

без изменения множества строк, порождаемых из A. Этого правила достаточно для многих грамматик.

Пример 13. Рассмотрим следующую грамматику для арифметических выражений:

$E \to E + T \mid T$
$T \to t * f \mid f$
$F \to (E) \mid id$

Устранив непосредственную левую рекурсию (продукции вида $A \to A\alpha$) из продукций для E и T, получим:

$E \to TE'$
$E' \to + TE' \mid \lambda$
$T \to FT'$
$T' \to * FT' \mid \lambda$
$F \to (E) \mid id$

Общее количество A-продукций роли не играет. Можно устранить непосредственную левую рекурсию из них с помощью следующей технологии. Вначале группируем A-продукции:

.

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

где β_i не начинаются с A. Затем заменим эти A-продукции

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \lambda$$

Нетерминал A порождает те же строки, что и ранее, но без левой рекурсии. Эта процедура устраняет все непосредственные левые рекурсии из продукций для A и A' (при условии, что ни одна строка α_i не является λ), но не устраняет левую рекурсию, вызванную двумя или более шагами порождения. Например, в грамматике нетерминал S леворекурсивен, поскольку $S => Aa => Sda$, но эта рекурсия не является непосредственной.

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid \lambda$$

Алгоритм устранения левой рекурсии гарантированно работает с грамматиками, не имеющими циклов и λ-продукций. Из грамматики могут быть также удалены и циклы, и λ-продукции.

5.9. Левая факторизация

Левая факторизация представляет собой преобразование грамматики в грамматику, пригодную для предиктивного анализа. Основная идея левой факторизации заключается в том, что когда неясно, какая из двух альтернативных продукций должна использоваться для нетерминала *A*, *A*-продукции можно переписать так, чтобы отложить принятие решения до тех пор, пока из входного потока не будет прочитано достаточно символов для правильного выбора.

Например, если есть две продукции

stmt → if **expr** then **stmt** else **stmt**
if **expr** then **stmt**

то, обнаружив во входном потоке if, нельзя тут же выбрать ни одну из них. В общем случае, если $A \to \alpha\beta_1, | \alpha\beta_2$ представляют собой две *A*-продукции и входной поток начинается с непустой строки, порождаемой α, нельзя сказать, какая продукция будет использоваться (первая или вторая). Однако решение можно отложить, расширив *A* до α *A*'. В этом случае, после того как рассмотрен входной поток, выводимый из α, работаем с *A*', расширяя его до β_1 или β_2. Таким образом, будучи левофакторизованными, исходные продукции принимают вид:

$$A \to \alpha A'$$
$$A' \to \beta_1 | \beta_2$$

Пример 14. Следующая грамматика решает проблему «кочующего else»:

stmt → if **expr** then **stmt** | if **expr** then **stmt** else **stmt** | *a*
expr → *b*

Будучи левофакторизованной, эта грамматика принимает следующий вид:

stmt → if **expr** then **stmt** ***S'*** | *a*
S' → else **stmt** | λ
E → *b*

Таким образом, при получении на входе *if* используется продукция *if* **expr** then **stmt** ***S'*** и, только когда будет просмотрена if **expr** then **stmt**, можно решить, расширять ли *S* ' до else **stmt** или λ. Однако обе грамматики неоднозначны, и при входном символе else неясно, какая из альтернатив для *S*' должна быть выбрана.

5.10. Эквивалентные преобразования КС-грамматик

Цель преобразования грамматик – упрощение правил грамматики и облегчение создания распознавателя. При создании компиляторов вторая цель наиболее важная, поэтому иногда упрощением грамматик пренебрегают.

Грамматики, к которым применены преобразования в определенном порядке, называются *приведенными.*

Последовательность правил преобразований грамматик должна быть следующей:

1) удаление всех бесполезных символов (нетерминальный символ грамматики является бесполезным, если он не играет никакой роли в построении правильных цепочек языка);

2) удаление всех недостижимых символов (нетерминальный символ грамматики является недостижимым, если он никогда не появляется в выводимых цепочках);

3) удаление λ-продукций (грамматика является λ-свободной, если множество продукций не содержит λ -продукций или есть ровно одно λ-правило $S \to \lambda$ и S не встречается в правых частях остальных продукций);

4) удаление цепных правил (грамматика является грамматикой без циклов, если в ней нет выводов вида $A \overset{+}{\Rightarrow} A$).

5.11. КС-языки и магазинные автоматы

Описание КС-языка с помощью порождающей КС-грамматики не является описанием алгоритма порождения предложений этого языка. Правила подстановки грамматики – это не последовательность предписаний, а совокупность разрешений, причем порядок применения правил в грамматике произволен, тогда как в алгоритме должен быть задан жесткий порядок применения отдельных инструкций.

Получение алгоритмического описания процесса распознавания языка является одной из первоочередных задач при разработке блока синтаксического анализа транслятора.

Одним из способов описания алгоритма распознавания языка является задание его в виде некоторого распознающего устройства. Для КС-языков такими устройствами являются магазинные автоматы (автоматы с магазинной памятью, МП-автоматы).

Недетерминированным МП-автоматом называется семерка вида

$$M = <A, Q, \Gamma, \delta, q_0, Z_0, F>,$$

где A – конечное множество входных символов (входной алфавит); Q – конечное множество внутренних состояний (алфавит состояний); Γ – конечное множество магазинных символов; $q_0 \in Q$ – начальное состояние автомата; $F \in Q$ – множество заключительных состояний; δ – отображение $Q \times A \times \Gamma$ в множество $Q \times \Gamma^*$, т.е. отображение вида $\delta : Q \times A \times \Gamma \rightarrow (Q \times \Gamma^*)$.

Схема МП-автомата показана на рис. 20. Автомат имеет конечное множество состояний, конечное множество входных символов и неограниченную ленту, называемую лентой магазинной памяти. «Дно» магазина (самый нижний символ) отмечается специальным символом, называемым *маркером дна* (например, символом #). Магазинная память определяется свойством «первым пришел – последним ушел». При записи символа в магазин, его содержимое сдвигается на одну ячейку «вниз», а на освободившееся место записывается требуемый символ. В этом случае говорят, что символ «вталкивается» в магазин.

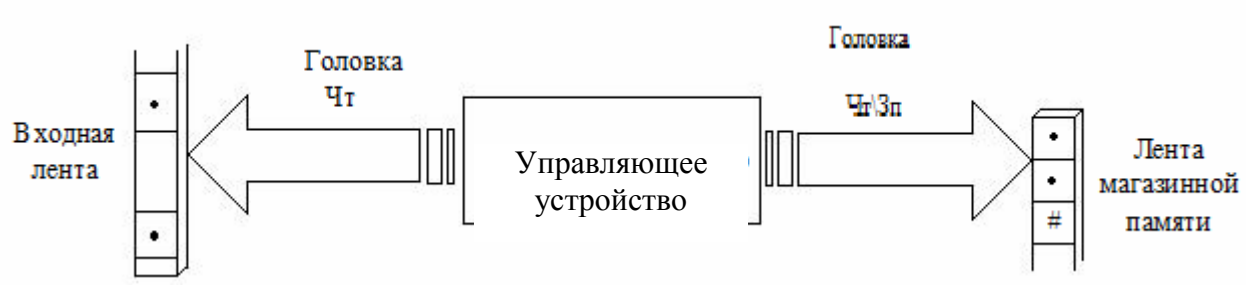


Рис. 20. Схема автомата с магазинной памятью

Для чтения доступен только самый верхний символ магазина. Этот символ после чтения может либо остаться в магазине, либо быть удален из него, т.е. «вытолкнут» из магазина. За один такт работы МП-автомата из магазина можно удалять не более одного символа.

Каждый шаг работы МП-автомата задается множеством правил перехода автомата из одних состояний в другие. Переходы в общем случае определяются:

– состоянием МП-автомата;
– верхним символом магазина;
– текущим входным символом.

Множество правил перехода называется *управляющим устройством*. В зависимости от получаемой информации управляющее устройство реализует один такт работы МП-автомата, который включает в себя три операции:

1) операции над магазином:
– втолкнуть в магазин определенный символ;
– вытолкнуть верхний символ из магазина;
– оставить содержимое магазина без изменений;
2) операции над состоянием:
– перейти в заданное новое состояние *S*;
– остаться в прежнем состоянии;
3) операции над входом:
– перейти к следующему входному символу и сделать его текущим;
– оставить данный входной символ текущим, т.е. держать его до следующего шага.

Обработку входной цепочки МП-автомат начинает в некотором выделенном состоянии при определенном содержимом магазина. Затем автомат выполняет операции, задаваемые его управляющим устройством. При этом выполняется либо завершение обработки, либо переход в новое состояние. Если выполняется переход, то он дает новый верхний символ магазина, новый текущий символ, автомат переходит в новое состояние и цикл работы автомата повторяется.

МП-автомат называется *МП-распознавателем*, если у него два выхода: ДОПУСТИТЬ и ОТВЕРГНУТЬ.

Работа МП-автомата при распознавании конкретной цепочки описывается обычно в виде последовательности конфигураций МП-автомата. Конфигурация МП-автомата определяет содержимое стека; состояние; необработанную часть входной цепочки.

Более формальный способ описания МП-автомата – в виде последовательности команд. Команда записывается в виде

$$(q, a, z) \rightarrow \{(q_1, \gamma_1), ..., (q_m, \gamma_m)\},$$

где $q, q_1, ..., q_m \in Q$; $a \in A$; $z \in \Gamma$, $\gamma_1, \gamma_2, \ldots, \gamma_m \in \Gamma^*$.

Команда интерпретируется следующим образом: МП-автомат находится в состоянии q, считывает входной символ a и верхний символ магазина z, переходит в состояние q_i, заменяя в магазине символ z на символ γ_i, $1 \leq i \leq m$ и продвигает входную головку на один символ.

Если же при выполнении команды не выполняется сдвиг входной ленты, то команда записывается в виде

$$(q, \lambda, z) \rightarrow \{(q_1, \gamma_1), ..., (q_m, \gamma_m)\}$$

и используется лишь для изменения содержимого магазина.

Различают *детерминированные и недетерминированные* МП-автоматы. Если среди команд МП-автомата нет двух, у которых совпадают левые части и не совпадают части, стоящие справа от стрелки, то МП-автомат называют *детерминированным*. В противном случае, т.е. когда для заданного состояния и текущих входном и магазинном символах возможны переходы автомата в различные состояния, его называют *недетерминированным*.

Связь между КС-языками и недетерминированными автоматами выражается теоремами, которые показывают, что класс языков, допускаемых магазинными автоматами при пустом магазине, есть в точности класс КС-языков.

Теорема. Пусть $L(G)$ – КС-язык, порождаемый грамматикой $G = < N, T, P, S >$ в нормальной форме Грейбах (правила вывода которой имеют вид $A \to b\alpha$). Тогда существует недетерминированный нисходящий МП-автомат M, допускающий все слова языка $L(G)$. Автомат $M = < A, Q, \Gamma, \delta, q_0, Z_0, F >$ строится следующим образом:

1) $A = T$;
2) $Q = \{q,\}$;
3) $T = N$;
4) $q_0 = q_q$;
5) $Z_0 = S$;
6) $F =$;
7) δ: $(q, a, B) \to (q_p \gamma)$, когда подстановка $B \to a\gamma$ принадлежит множеству правил P грамматики G, здесь $B \in N$, $a \in T$, $\gamma \in \{N \cup T\}^*$ [5].

Пример 15. Дана КС-грамматика $G = <N, T, P, S>$, у которой

$N = \{S, D, B\}$, $T = \{a, b\}$,
$P = \{S \to aB \mid bD$,
$D \to aS \mid a \mid bDD$,
$B \to aBB \mid bS \mid b\}$.

В соответствии с теоремой МП-автомат $M = <A, Q, \Gamma, \delta, q_0, Z_0, F>$, допускающий данный КС-язык, будет включать в себя компоненты $A = \{a, b\}$, $Q = \{q,\}$, $\Gamma = \{S, D, B\}$, $q_0 = q_1$, $z_0 = S$ и отображение δ, заданное в виде:

l. $(q_1, a, S) \to (q_1, B)$,	так как $(S \to aB) \in P$;
2. $(q_1, b, S) \to (q_1, D)$;	так как $(S \to bD) \in P$;
3. $(q_1, a, D) \to \{(q_1, S), (q_1, \lambda)\}$,	так как $(D \to aS)$ и $(D \to a)$ входят в множество Р;
4. $(q_1, b, D) \to (q_1, DD)$,	так как $(D \to bDD) \in P$;
5. $(q_1, a, B) \to (q_1, BB)$,	так как $(B \to aBB) \in P$;
6. $(q_1, b, B) \to \{(q_1, S), (q_1\ \lambda)\}$,	так как $(B \to bS)$ и $(B \to b)$ входят в множество P.

Построенный МП-автомат является недетерминированным, так как в его правилах (3 и 6) допускается неоднозначность перехода для одной и той же комбинации состояния входного и магазинного символов. В общем случае классы языков, допускаемых детерминированными и недетерминированными МП-автоматами, не совпадают. На практике же получили наибольшее применение детерминированные методы разбора для детерминированных КС-языков.

Вопросы

1. Какие функции выполняет синтаксический анализатор?
2. Какая грамматика является однозначной?
3. Какой цели служат преобразования правил КС-грамматик?
4. Почему из правил грамматики необходимо устранять именно левую рекурсию?
5. Почему при преобразовании КС-грамматик к приведенному виду сначала необходимо удалить бесполезные, а потом недостижимые символы?

6. НИСХОДЯЩИЙ АНАЛИЗ

Нисходящий разбор можно рассматривать как попытку найти левое порождение входной строки. Также его можно рассматривать и как попытку построить дерево разбора для входной строки, начиная с корня и создавая вершины в прямом порядке.

6.1. Анализ методом рекурсивного спуска

Рассмотрим нисходящий анализ в общем виде, а именно анализ методом рекурсивного спуска, который может использовать откаты, т.е. производить повторное сканирование входного потока. Однако такие синтаксические анализаторы с откатом встречаются редко. При анализе языков программирования технология отката не очень эффективна и предпочтительными являются табличные методы.

Рассмотрим пример, в котором откат необходим.

Пример 16. Рассмотрим грамматику

$$S \rightarrow cAd$$

$$A \rightarrow ab|a$$

и входную строку $w = cad$.

При нисходящем построении дерева разбора для этой строки вначале создаем дерево, состоящее из одного узла, помеченного как S. Указатель входа находится над первым символ строки c. Теперь воспользуемся первой продукцией для S, чтобы получить дерево, показанное на рис. 21, *а*.

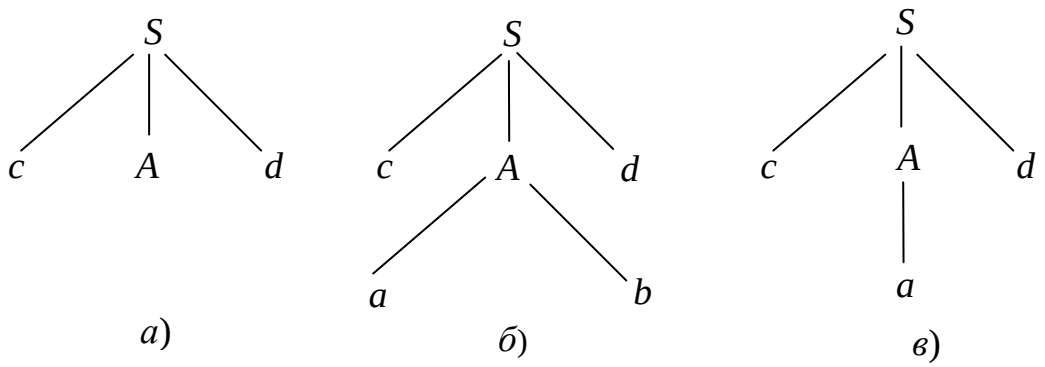


Рис. 21. Шаги нисходящего разбора

Крайний слева лист c соответствует первому символу строки w, так что переместим указатель входа к a, второму символу строки w, и рассмотрим следующий лист дерева, помеченный A. Теперь можно воспользоваться для A первой альтернативой и получить дерево, изображенное на рис. 21, *б*. Здесь обнаружено соответствие считанного символа листу дерева и осуществляется переход к следующему символу – d. Однако d не соответствуют листу дерева b, значит, надо вернуться к A, чтобы выбрать новую альтернативу для работы.

Возвращаясь к A, необходимо вернуть указатель входа в позицию 2, в которой он был, когда впервые выбирали продукцию для разложения A. Это означает, что процедура для A должна хранить указатель входа локальной переменной. При рассмотрении второй альтернативы для A получаем дерево, показанное на рис. 21, *в*. Лист

a соответствует второму символу *w*, а лист *d* – третьему. Поскольку в этот момент построено дерево разбора для *w*, прекращаем работу и сообщаем об успешном завершении разбора.

Леворекурсивная грамматика может вызвать зацикливание синтаксического анализатора, работающего методом рекурсивного спуска, даже с откатами (при разборе *A* может возникнуть необходимость вновь проанализировать *A*, находясь в той же входной позиции, что автоматически приведет к бесконечной рекурсии).

6.2. Предиктивные анализаторы

Во многих случаях аккуратная разработка грамматики, устранение из нее левой рекурсии и ее левая факторизация позволяют получить грамматику, которая может быть проанализирована синтаксическим анализатором, работающим методом рекурсивного спуска и не требующим отката. Для построения предиктивного синтаксического анализатора необходимо знать, какая из альтернатив $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$ данного анализируемого нетерминала *A* порождает строку, начинающуюся с полученного входного символа *a*. То есть правильная альтернатива должна точно определяться по первому порождаемому ею символу. Обычно в языках программирования управляющие конструкции отвечают этому правилу. Например, если есть продукции

stmt → if **expr** then **stmt** else **stmt** |while **expr** do stmt | begin **stmt**_list end

то ключевые слова if, while и begin однозначно определяют возможную альтернативу для рассматриваемой инструкции **stmt**.

6.3. Нерекурсивный предиктивный анализ

Нерекурсивный предиктивный синтаксический анализатор можно построить с помощью явного использования стека. Основная проблема предиктивного анализа заключается в определении продукции, которую следует применить к нетерминалу. Нерекурсивный синтаксический анализатор, представленный на рис. 22, ищет необходимую для анализа продукцию в таблице разбора.

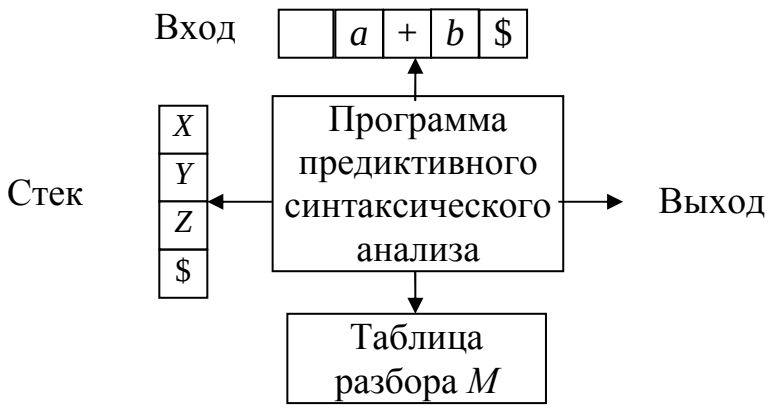


Рис. 22. Модель нерекурсивного предиктивного синтаксического анализатора

Предиктивный синтаксический анализатор включает в себя: управляющую программу, входной буфер, стек, таблицу разбора и выходной поток. Входной буфер содержит анализируемую строку с маркером ее правого конца – специальным символом. Стек содержит последовательность символов грамматики с символом $ на дне. Изначально стек содержит стартовый символ грамматики непосредственно над символом $. Таблица разбора представляет собой двухмерный массив $M[A, a]$, где A – нетерминал, а a – терминал или символ $.

Синтаксический анализатор управляется программой, которая работает следующим образом. Программа рассматривает X – символ на вершине стека и a – текущий входной символ. Эти два символа определяют действия синтаксического анализатора. Имеется три варианта:

1. Если $X = a = \$$, синтаксический анализатор прекращает работу и сообщает об успешном завершении разбора.

2. Если $X = a \neq \$$, синтаксический анализатор снимает со стека X и перемещает указатель входного потока к следующему символу.

3. Если X представляет собой нетерминал, программа рассматривает запись $M[X, a]$ таблицы разбора M. Эта запись представляет собой либо X-продукцию грамматики, либо запись об ошибке. Например, если $M[X, a] = \{X \rightarrow UVW\}$, то синтаксический анализатор замещает X на вершине стека на WVU (с U на вершине стека). В качестве выхода синтаксический анализатор просто выводит исполь-

зованную продукцию. Если $M[X, a]$ = **error**, синтаксический анализатор вызывает программу восстановления после ошибки.

Поведение синтаксического анализатора может описываться его *конфигурациями*, которые дают содержимое стека и оставшийся входной поток.

6.4. Множества FIRST и FOLLOW

При построении предиктивного синтаксического анализатора необходимо построить два множества, связанные с грамматикой *G*, – FIRST и FOLLOW, которые обеспечивают заполнение таблицы предиктивного анализа грамматики *G*. Если α – произвольная строка символов грамматики, то определим FIRST(α) как множество терминалов, с которых начинаются строки, выводимые из α. Если $\alpha \overset{+}{\Rightarrow} \lambda$, то λ входит в FIRST(α).

FOLLOW(*A*) для нетерминала *A* определяется как множество терминалов *a*, которые могут располагаться непосредственно справа от *A* в некоторой сентенциальной форме, т.е. множество терминалов *a* таких, что существуют порождения вида $S \overset{+}{\Rightarrow} \alpha A a\beta$ для некоторых α и β. В процессе приведения между *A* и *a* могут появиться символы, но они порождают λ и в конечном счете исчезают. Если *A* может оказаться крайним справа символом некоторой сентенциальной формы, то $ входит в FOLLOW(*A*).

FIRST(*X*) для всех символов грамматики *X* вычисляется с применением следующих правил до тех пор, пока ни к одному из множеств FIRST не смогут быть добавлены ни терминалы, ни λ.

1. Если *X* – терминал, то FIRST(*X*) = {*X*}.
2. Если имеется продукция $X \to \lambda$, добавим λ к FIRST(*X*).
3. Если *X* нетерминал и имеется продукция $X \to Y_1Y_2...Y_k$, то поместим *a* в FIRST(*X*), если для некоторого *i* $a \in$ FIRST(Y_i) и λ входит во все множества FIRST(Y_1)… FIRST(Y_{i-1}), т.е. $Y_1 \ldots Y_{i-1} \overset{+}{\Rightarrow} \lambda$. Если λ имеется во всех FIRST(Y_i), i = 1, k, то добавляем λ к FIRST(*X*). Например, все, что находится во множестве FIRST(Y_i), есть и в множестве FIRST(*X*). Если Y_1 не порождает λ, то больше мы ничего не добавляем к FIRST(*X*), но если $Y_1 \overset{+}{\Rightarrow} \lambda$, то к FIRST(*X*) добавляем FIRST(Y_2) и т.д.

Для любой строки $X_1X_2...X_n$ FIRST вычисляется следующим образом. Добавим к FIRST($X_1X_2...X_n$) все не λ-символы из FIRST(X_1). Добавим также все не λ-символы из FIRST(X_2), если λ є FIRST(X_1), все не λ-символы из FIRST(X_3), если λ имеется как в FIRST(X_1), так и в FIRST(X_2) и т.д. И наконец, добавим λ к FIRST($X_1X_2...X_n$), если для всех i FIRST(X_i) содержит λ.

FOLLOW(A) для всех нетерминалов A вычисляется с применением следующих правил до тех пор, пока ни к одному множеству FOLLOW нельзя будет добавить ни одного символа.

1. Поместим \$ в FOLLOW(S), где S – стартовый символ, а \$ – правый ограничитель входного потока.

2. Если имеется продукция $A \rightarrow \alpha B \beta$, то все элементы множества FIRST(β), кроме λ, помещаются в множество FOLLOW(B).

3. Если имеется продукция $A \rightarrow \alpha B$, или $A \rightarrow \alpha B \beta$, где FIRST(β) содержит λ (т.е. β => λ), то все элементы из множества FOLLOW(A) помещаются в множество FOLLOW(B) [3].

Пример 17. Для грамматики арифметических выражений

$E \rightarrow TE'$
$E' \rightarrow +\, TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *\, FT' \mid \lambda$
$F \rightarrow (E) \mid \mathbf{id}$

множества будут иметь вид:

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E') = {+, λ}
FIRST(T') = {*, λ}
FOLLOW(E) = FOLLOW(E') = {), \$}
FOLLOW(T) = FOLLOW(T') = {+,), \$}
FOLLOW(F) = {+,*,),\$}

Например, **id** и левая скобка добавляются к FIRST(F) по правилу (3) в определении FIRST, поскольку FIRST(**id**) = {**id**}, и к FIRST('(') = = {(} по правилу (1). По правилу (3) с i = 1 из продукции $T \rightarrow FT'$ следует, что **id** и левая скобка входят и в FIRST(T). В соответствии с правилом (2) λ входит в FIRST(E').

Для вычисления множеств FOLLOW помещаем $ в FOLLOW(*E*) в соответствии с правилом (1) для вычисления FOLLOW. По правилу (2), примененному к продукции $F \rightarrow (E)$, правая скобка также входит в множество FOLLOW(*E*). В соответствии с правилом (3), примененным к продукции $E \rightarrow TE'$, $ и правая скобка входят в FOLLOW(*E'*). Поскольку $E' \stackrel{*}{\Rightarrow} \lambda$, эти же символы входят и в FOLLOW(*T*). Из продукции $E \rightarrow TE'$, согласно правилу (2), следует, что все, что имеется в множестве FIRST(*E'*) (за исключением λ), должно входить в множество FOLLOW(*T*). В это же множество входит также $.

6.5. Построение таблиц предиктивного анализа

Для построения таблицы предиктивного анализа данной грамматики G используется следующий алгоритм.

1. Для каждой продукции грамматики $A \rightarrow \alpha$ выполняем шаги 2 и 3.

2. Для каждого терминала α из FIRST(*A*) добавляем $A \rightarrow \alpha$ в ячейку *M*[*A*, *a*].

3. Если в FIRST(*A*) входит λ, для каждого терминала *b* из FOLLOW(*A*) добавим $A \rightarrow b$ в ячейку *M*[*A*, *b*]. Если λ входит в FIRST(α), а $ – в FOLLOW(*A*), добавим $A \rightarrow \alpha$ ячейку *M*[*A*, $].

4. Каждая неопределенная ячейка таблицы *M* указывает на ошибку [3].

Пример 18. Применим алгоритм заполнения таблицы к грамматике из примера 16. Поскольку FIRST(*TE'*) = FIRST(*T*) = {(, **id**}, продукция $E \rightarrow TE'$ приводит к размещению в ячейках *M*[*E*, (] и *M*[*E*, **id**] записи $E \rightarrow TE'$.

Продукция $E' \rightarrow +TE'$ позволяет внести ее в ячейку *M*[*E'*, +]. Продукция *E'* приводит, с учетом FOLLOW(*E'*) = {), $}, к внесению $E' \rightarrow \lambda$ в ячейки *M*[*E'*,)] и *M[E'*, $].

Полностью таблица предиктивного анализа приведена в табл. 9.

Пустые ячейки таблицы означают ошибки; непустые ячейки указывают продукции, с помощью которых заменяются нетерминалы на вершине стека.

Таблица 9

Таблица разбора для грамматики из примера 16

Нетер-минал	Входной символ					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E′		$E' \to +\, TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T′		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to id$			$F \to (E)$		

При входном потоке **id + id * id** предиктивный синтаксический анализатор осуществляет последовательность перемещений, показанную в табл. 10.

Таблица 10

Перемещения предиктивного синтаксического анализатора при входной строке id + id * id

Стек	Вход	Выход
$E	*id* + *id* * *id*$	
$E′T	*id* + *id* * *id*$	$E \to TE'$
$E′T′F	*id* + *id* * *id*$	$T \to FT'$
$E′T′ **id**	*id* + *id* * *id*$	$F \to$ **id**
$E′T′	+ *id* * *id*$	
$E′	+ *id* * *id*$	$T' \to \varepsilon$
$E′T +	+ *id* * *id*$	$E' \to +TE'$
$E′T	*id* * *id*$	
$E′T′F	*id* * *id*$	$E \to FT'$
*$E′T′***id**	*id* * *id*$	$F \to$ **id**
$E′T	* *id*$	
$E′T′F *	* *id*$	$T' \to * FT'$
$E′T′F	*id*$	
$E′T′ **id**	*id*$	$F \to$ **id**
$E′T′	$	
$E′	$	$T' \to \lambda$
$	$	$E' \to \lambda$

Входной указатель при перемещении показывает на крайний слева символ в столбце «Вход». При рассмотрении действий синтаксического анализатора видно, что его выход совпадает с последовательностью продукций, применяемых в левом порождении. Если же к прочитанным входным символам приписать символы в стеке (от вершины до дна), то получится левосентенциальная форма в порождении.

Синтаксический анализатор находит ошибку в тот момент, когда терминал на вершине стека не соответствует очередному входному символу или на вершине стека находится нетерминал *A*, очередной входной символ – *a*, а ячейка таблицы синтаксического анализа *M*[*A*, *a*] пуста.

6.6. *LL*(1)-грамматики

Грамматика, таблица анализа которой не имеет множественных записей, называется *LL*(1). Первое *L* означает просмотр входного потока слева направо, второе *L* – левое порождение, а 1 – просмотр одного символа из входного потока на каждом шаге для принятия решения о дальнейших действиях.

Для некоторых грамматик таблица разбора *M* может иметь несколько записей в одной ячейке таблицы. Например, если грамматика *G* леворекурсивная или неоднозначная, то таблица разбора *M* будет иметь как минимум одну ячейку с несколькими записями.

LL(1)-грамматика имеет ряд отличительных свойств. Такая грамматика не может быть неоднозначной или леворекурсивной. Можно показать, что грамматика *G* является *LL*(1)-грамматикой тогда и только тогда, когда для любых двух различных ее продукций $A \rightarrow \alpha \mid \beta$ выполняются следующие условия.

1. Не существует такого терминала *a*, для которого и α, и β порождают строку, начинающуюся с *a*.

2. Пустую строку может порождать только одна из продукций: α или β.

3. Если $\beta \Rightarrow \lambda$, то *a* не порождает ни одну строку, начинающуюся с терминала из FOLLOW(*A*).

Грамматика для арифметических выражений является *LL*(1)-грамматикой. Грамматика, моделирующая инструкции **if-then-else,** таковой не является.

Если таблица анализа имеет ячейки с несколькими записями, выход состоит в преобразовании грамматики, устраняющем левую рекурсию, и левой факторизации, чтобы получить грамматику, в таблице анализа которой отсутствуют ячейки с несколькими записями. К сожалению, имеются грамматики, никакие изменения которых не приведут к *LL*(1)-грамматике. Не существует универсальных правил, с помощью которых ячейки с несколькими записями можно превратить в однозначно определенные без воздействия на язык, распознаваемый синтаксическим анализатором.

Основная сложность в использовании предиктивного анализа состоит в написании для исходного языка такой грамматики, которая позволяет построить предиктивный синтаксический анализатор.

Вопросы

1. На каком алгоритме основана работа распознавателя для *LL*(1)-грамматик?
2. Какие преобразования необходимо выполнить с грамматикой, чтобы к ней можно было применить нисходящие методы разбора?
3. Почему распознаватели с возвратами (откатами) назад не нашли широкого применения?
4. Какие грамматики являются *LL*(*k*)-грамматиками?
5. Как описывается один такт работы *LL*(1)-анализатора?

7. ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Основной метод восходящего синтаксического анализа – синтаксический анализ типа «перенос/свертка» (ПС-анализ). В процессе ПС-анализа дерево разбора для входной строки строится начиная с листа (снизу) и работает по направлению к корню дерева (вверх). Этот процесс можно рассматривать как свертку строки *w* к стартовому символу грамматики. На каждом шаге свертки некото-

рая подстрока, соответствующая правой части продукции, заменяется символом из левой части этой продукции, и если на каждом шаге подстроки выбираются корректно, то получается обращенное правое порождение.

Пример 19. Рассмотрим грамматику

$S \rightarrow aABe$
$A \rightarrow Abe \mid b$
$B \rightarrow d$

Предложение *abbcde* сводится к *S* с помощью следующих шагов:

abbcde
aAbcde
aAdle
aABe
S

Строка *abbcde* сканируется слева направо в поисках подстроки, соответствующей правой части какой-либо продукции. Такими подстроками являются *b* и *d*. Выбираем крайнее слева *b* и заменяем его нетерминалом *A*, который представляет собой левую часть продукции $A \rightarrow b$; таким образом, получаем строку *aAbcde*. Теперь правым частям продукций соответствуют подстроки *Abc*, *b* и *d*. Выбираем для замены подстроку *Abc* и заменяем ее нетерминалом *A* в соответствии с продукцией $A \rightarrow Abc$. В результате получаем строку *aAde*. Заменяя *d* на *B*, левую часть продукции $B \rightarrow d$, получаем *aABe*, которая в соответствии с первой продукцией заменяется стартовым символом *S*. Итак, последовательность из четырех сверток позволяет привести строку *abbcde* к стартовому символу *S*. Эти сокращения представляют собой обращенное (т.е. записанное в обратном порядке) правое порождение $S \Rightarrow \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$.

7.1. Понятие основы строки

Основа строки – это подстрока, которая совпадает с правой частью продукции и свертка которой в левую часть продукции представляет собой один шаг обращенного правого порождения.

Во многих случаях крайняя слева подстрока β, соответствующая правой части некоторой продукции *A*, не является основой, по-

скольку свертка в соответствии с продукцией $A \to \beta$ приводит к строке, которая не может быть свернута к стартовому символу. Если в примере 18 заменить во второй строке *aAbcde* символ *b* нетерминалом *A*, то получим строку *aAAcde*, которая не может быть свернута в *S*. Поэтому определение основы должно быть более точное.

Основа правосентенциальной формы γ является продукцией $A \to \beta$ и позицией строки β в γ, такими, что β может быть заменена нетерминалом *A* для получения предыдущей правосентенциальной формы в правом порождении γ. Таким образом, если $S \Rightarrow \alpha Aw \Rightarrow \alpha\beta w$, то $A \to \beta$ в позиции после *a* представляет собой основу строки $a\beta w$. Строка *w* справа от основы содержит только терминальные символы. Если грамматика однозначна, то каждая правосентенциальная форма грамматики имеет ровно одну основу.

Пример 20. Рассмотрим следующую грамматику:

(1) $E \to E + E$
(2) $E \to E * E$
(3) $E \to (E)$
(4) $E \to id$

и правое порождение

$$\begin{aligned} E &\Rightarrow \underline{E + E} \\ &\Rightarrow E + \underline{E * E} \\ &\Rightarrow E + E * \underline{id_3} \\ &\Rightarrow E + \underline{id_2} * id_3 \\ &\Rightarrow \underline{id_1} + id_2 * id_3 \end{aligned}$$

Отметим подстрочными индексами *id* и подчеркнем основу каждой правосентенциальной формы. Например, id_1 представляет собой основу правосентенциальной формы $id_1 + id_2 * id_3$, поскольку *id* является правой частью продукции $E \to id$, и замена id_1 на *E* приведет к предыдущей правосентенциальной форме $E + id_2 * id_3$. Строка справа от основы состоит только из терминальных символов.

Поскольку грамматика из примера 20 неоднозначна, имеется еще одно правое порождение той же строки:

$E \Rightarrow \underline{E + E}$

$\Rightarrow E * \underline{id_3}$

$\Rightarrow \underline{E + E} * id_3$

$\Rightarrow E + \underline{id_2} * id_3$

$\Rightarrow \underline{id_1} + id_2 * id_3$

Рассмотрим правосентенциальную форму $E + E * id_3$. В этом порождении $E + E$ – основа $E + E * id_3$, в то время как в ранее представленном порождении ее основой является id_3.

В этом примере рассмотрены два правых порождения. Первое порождение дает больший приоритет оператору *, чем оператору +, в то время как во втором порождении приоритет оператора + выше.

Пример 21. Рассмотрим грамматику из примера 19 и входную строку $id_1 + id_2 * id_3$. Последовательность сверток, приводящая входную строку к стартовому символу E, показана в табл. 11. Последовательность правосентенциальных форм в этом примере представляет собой обращение первой последовательности правых порождений.

Таблица 11

Свертки, выполняемые ПС-анализатором

Правосентенциальная форма	Основа	Сворачивающая продукция
$id_1 + id_2 + id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
E	$E + E$	$E \rightarrow E + E$

7.2. Стековая реализация ПС-анализа

Существует две проблемы при синтаксическом анализе методом ПС-анализа. Первая заключается в обнаружении подстроки для свертки в правосентенциальной форме, вторая – в определении, какая именно продукция должна быть выбрана, если имеется несколько продукций с соответствующей подстрокой в правой части. Достаточно удобный путь реализации ПС-анализатора состоит в использовании стека для хранения символов грамматики и входно-

го буфера для хранения анализируемой строки. В качестве маркера дна стека используется $, и этот же символ является маркером правого конца входной строки. Изначально стек пуст, а входной буфер содержит строку *w*$:

Стек	**Вход**
$	*w*$

Синтаксический анализатор работает путем переноса нуля или нескольких символов в стек до тех пор, пока на вершине стека не окажется основа β. Затем он свертывает β левой части соответствующей продукции. Синтаксический анализатор повторяет этот цикл, пока не будет обнаружена ошибка или он не придет в конфигурацию, когда в стеке находится только стартовый символ, а входной буфер пуст:

Стек	**Вход**
$*S*	$

Попав в эту конфигурацию, синтаксический анализатор прекращает работу и сообщает об успешном разборе входной строки.

Пример 22. Последовательность действий, выполняемых синтаксическим анализатором при разборе входной строки $id_1 + id_2 * id_3$ грамматики из примера 20, использующие первое порождение показана в табл. 12.

Таблица 12

Конфигурации ПС-анализатора для входной строки $id_1 + id_2 * id_3$

Стек	Вход	Действие
(1) $	$id_1 + id_2 * id_3$ $	Перенос
(2) $$id_1$	$+ id_2 * id_3$ $	Свертка по $E \rightarrow id$
(3) $$E$	$+ id_2 * id_3$ $	Перенос
(4) $$E$ +	$id_2 * id_3$ $	Перенос
(5) $$E + id_2$	$* id_3$ $	Свертка по $E \rightarrow id$
(6) $$E + E$	$* id_3$ $	Перенос
(7) $$E + E$ *	id_3 $	Перенос
(8) $$E + E * id_3$	$	Свертка по $E \rightarrow id$
(9) $$E + E * E$	$	Свертка по $E * E$
(10) $$E + E$	$	Свертка по $E + E$
(11) $$E$	$	Допуск

Основными операциями синтаксического анализатора являются *перенос* и *свертка*, но на самом деле ПС-анализатор может выполнять четыре действия: (1) перенос, (2) свертка, (3) допуск, (4) ошибка.

1. При *переносе* очередной входной символ переносится на вершину стека.

2. При *свертке* синтаксический анализатор распознает правый конец основы на вершине стека, после чего он должен найти левый конец основы и принять решение о том, каким нетерминалом заменить основу.

3. При *допуске* синтаксический анализатор сообщает об успешном разборе входной строки.

4. При *ошибке* синтаксический анализатор обнаруживает ошибку во входном потоке и вызывает программу восстановления после ошибок.

Замечание. Синтаксический анализатор для получения очередной основы переносит нуль или несколько символов в стек. *Синтаксический анализатор никогда не заглядывает внутрь стека в поисках правого края основы*. Все это делает стек особенно удобным для использования в реализации ПС-анализатора.

7.3. Конфликты в процессе ПС-анализа

Существуют контекстно-свободные грамматики, для которых ПС-анализ неприменим. Любой ПС-анализатор для такой грамматики может достичь конфигурации, в которой синтаксический анализатор по информации о содержимом стека и об очередном входном символе не в состоянии решить, должен ли он использовать перенос или свертку (конфликт *перенос*/*свертка*) либо какая из нескольких возможных сверток должна применяться (конфликт *свертка*/*свертка*).

7.4. Синтаксический анализ приоритета операторов

Принцип организации распознавателя входных цепочек языка, заданного грамматикой предшествования, основан на том, что для каждой упорядоченной пары символов в грамматике устанавливается некоторое отношение, называемое *отношением приоритетов*.

В процессе разбора входной строки расширенный МП-автомат сравнивает текущий символ входной цепочки с одним из символов, находящихся на вершине стека автомата. В процессе сравнения проверяется, какое из возможных отношений приоритетов существует между этими двумя символами. В зависимости от найденного отношения, выполняется либо перенос, либо свертка. При отсутствии отношения приоритетов между символами алгоритм сигнализирует об ошибке.

Задача состоит в том, чтобы определить эти отношения предшествования между символами грамматики. При этом грамматика может быть отнесена к одному из классов грамматик предшествования (простого предшествования, расширенного предшествования, слабого предшествования и т.д.).

Основное понятие, которое используется в грамматиках предшествования, – отношение предшествования.

Пусть имеется КС-грамматика и на некотором шаге вывода получена сентенциальная форма вида $\xi^*\ \xi_i\ \xi_j\ \xi^*$. Символы $\xi_i\ \xi_j$ стоят рядом в сентенциальной форме. Соседство этих символов характеризуется одним из отношений специального вида – отношением предшествования.

1. Между символами ξ_i и ξ_j существует отношение $\xi_i \doteq \xi_j$, если в грамматике есть правило вида $A \to \alpha\ \xi_i\ \xi_j\ \beta$, где α, β принадлежат алфавиту V^*.

2. Между символами ξ_i и ξ_j существует отношение $\xi_i <\bullet\ \xi_j$, если в грамматике есть правило вида $A \to \alpha\ \xi_i\ B\ \beta$ и вывод вида $B \overset{+}{\Rightarrow} > \xi_j\ \gamma$, где α, β, γ принадлежат алфавиту V^*.

3. Между символами ξ_i и ξ_j существует отношение $\xi_i \bullet> \xi_j$, если в грамматике есть правило вида $A \to \alpha\ BC\ \beta$ и выводы вида $B \overset{+}{\Rightarrow} \gamma_1\ \xi_j$ и $C \overset{+}{\Rightarrow} \xi_j\ \gamma_2$ или правило вывода вида $A \to \alpha\ B\ \xi_j\ \beta$ и вывод $B \overset{+}{\Rightarrow} \gamma\ \xi_i$, где α, β, γ_1, γ_2, γ принадлежат алфавиту V^*.

Тип отношения предшествования показывает, что если:

$\xi_i \stackrel{\bullet}{=} \xi_j$, то оба символа принадлежат одной основе;

$\xi_i <\bullet \xi_j$, то ξ_j – самый левый символ некоторой основы;

$\xi_i \bullet> \xi_j$, то ξ_i – самый правый символ некоторой основы.

Если между символами ξ_i и ξ_j выполняется не более одного отношения предшествования, то можно выделить основу для выполнения свертки.

7.4.1. Грамматики простого предшествования

КС-грамматика является грамматикой *простого предшествования*, если она однозначна, не содержит ε-продукций и для любой пары ее символов (терминальных и нетерминальных) выполняется не более одного отношения предшествования.

Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения предшествования – это значит, что они не могут находиться рядом в одном элементе разбора синтаксически правильной цепочки.

Отношения предшествования зависят от порядка, в котором стоят символы.

7.4.2. Грамматики операторного предшествования

Грамматики, у которых нет продукций, правые части которых представляют собой или имеют два соседних нетерминала, называются *операторными*.

Пример 23. Следующая грамматика для выражений

$$E \rightarrow \quad EAE \mid (E) \mid -E \mid id$$
$$A \rightarrow \quad + \mid - \mid * \mid / \mid \uparrow$$

не является операторной, поскольку правая часть EAE имеет два (на самом деле – даже три) последовательных нетерминала. Однако если заменим A каждой из его альтернатив, то получим операторную грамматику

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$$

При синтаксическом анализе приоритета операторов определяются три непересекающихся *отношения приоритетов* – $<\bullet$, $\doteq$ и $\bullet>$ – *между терминалами*. Эти отношения приоритетов управляют выбором основ и имеют следующее содержание (табл. 13).

Таблица 13

Отношение	Значение
$a <\bullet b$	a «уступает приоритет» b
$a \doteq b$	a имеет тот же приоритет, что и b
$a \bullet> b$	a «забирает приоритет у» b

Хотя эти отношения и выглядят похожими на арифметические отношения «меньше», «равно» и «больше», они имеют совершенно иные свойства. Например, в одном и том же языке может быть так, что и $a <\bullet b$, и $a \bullet> b$ или для некоторых терминалов a и b не выполняется ни одно из отношений $a <\bullet b$, $a \bullet> b$ и $a \doteq b$.

Имеется два способа определения отношений приоритетов, которые должны выполняться между терминалами.

Первый способ – интуитивный. Он основан на традиционных понятиях ассоциативности и приоритета операторов. Например, если * имеет приоритет выше, чем +, то * $\bullet>$ + и + $<\bullet$ *. Такой подход разрешает неоднозначности грамматик, позволяет написать для них синтаксический анализатор приоритета операторов.

Второй способ выбора отношений приоритетов операторов заключается в том, что вначале строится однозначная грамматика языка, которая отражает правильную ассоциативность и приоритеты операторов в своих деревьях разбора. Получив однозначную грамматику, можно воспользоваться механическим методом построения на ее основе отношений приоритетов операторов. Эти отношения могут пересекаться и задавать язык, отличающийся от порождаемого исходной грамматикой, но со стандартными арифметическими выражениями проблем практически не бывает.

7.4.3. Использование отношений приоритетов операторов

Цель отношений приоритетов состоит в определении границ основы правосентенциальной формы: $<\bullet$ отмечает ее левый конец, $\bullet>$

– правый, $a \doteq$ находится внутри основы. Предположим, что есть правосентенциальная форма операторной грамматики. Из того, что у продукции не может быть двух смежных нетерминалов в правой части, следует, что и правосентенциальная форма не может иметь двух смежных нетерминалов. Таким образом, правосентенциальную форму можно записать в виде $\beta_0 a_1 \beta_1 ... a_n \beta_n$, где каждое β является либо пустой строкой, либо одиночным нетерминалом, а каждое a представляет собой одиночный терминал.

Предположим, что между a_i и a_{i+1} выполняется ровно одно отношение – $\bullet>$, $<\bullet$, $\doteq$. Используем для маркировки концов строки символ \$ и определим, что \$ $<\bullet$ b для всех терминалов b. Теперь предположим, что из строки удалили нетерминалы и поместили одно из отношений $\bullet>$, $<\bullet$ или $\doteq$ между каждой парой терминалов и между крайними терминалами и маркерами \$. В правосентенциальной форме *id* + *id* * *id* отношения приоритетов показаны в табл. 14 (эти отношения выбраны при рассмотрении грамматики из примера 22).

Таблица 14

Отношения приоритетов операторов

	id	+	*	\$
id		$\bullet>$	$\bullet>$	$\bullet>$
+	$<\bullet$	$\bullet>$	$<\bullet$	$\bullet>$
*	$<\bullet$	$\bullet>$	$\bullet>$	$\bullet>$
\$	$<\bullet$	$<\bullet$	$<\bullet$	

Тогда строка с отношениями приоритетов принимает вид

$$\$ <\bullet id \bullet> + <\bullet id \bullet> * <\bullet id \bullet> \$ \qquad (8)$$

Например, $<\bullet$ вставлен между крайним слева \$ и *id*, поскольку в ячейке в строке и столбце *id* находится отношение $<\bullet$. Основа может быть найдена следующим образом:

1. Сканируем строку слева направо, пока не встретим первый символ $\bullet>$. В (8) символ располагается между первым *id* и +.

2. Затем сканируем строку в обратном направлении (влево), пропуская все $\doteq$, пока не встретим <•. В (8) сканирование идет до символа $.

3. Основа содержит все, что находится слева от первого •> и справа от <•, найденного на шаге (2), включая все промежуточные и окружающие нетерминалы (включение окружающих нетерминалов необходимо для того, чтобы в правосентенциальной форме не появлялись два смежных нетерминала). В (8) основой является первый *id*.

Работая с грамматикой (8) свертываем *id* в *E*, в результате получаем правовосентенциальную форму *E* + *id* * *id*. После свертки двух оставшихся *id* в *E* получаем правосентенциальную форму *E* + *E* * *E*. Следующая строка $ +*$ получена удалением нетерминалов. Вставка отношений приоритетов дает строку $ <• + <• * •> $, из которой ясно, что левый конец основы находится между + и *, а правый – между * и $. Эти отношения приоритетов указывают, что в правосентенциальной форме *E* + *E* * *E* основой является *E* * *E*. Окружающие * нетерминалы *E* также являются частью основы.

Поскольку нетерминалы не влияют на процесс анализа, можно не беспокоиться об их распознавании. В стеке синтаксического анализатора приоритета операторов достаточно одного маркера «нетерминал», чтобы обозначать места для хранения значений атрибутов.

Если между терминалом на вершине стека и очередным входным символом выполняется отношение <• или $\doteq$, то синтаксический анализатор выполняет перенос (это означает, что правый конец основы еще не найден). Если отношение – •>, то выполняется свертка. В этот момент синтаксический анализатор обнаружил правый конец основы, и для поиска ее левого конца в стеке можно воспользоваться отношениями приоритета.

Если между парами терминалов не выполняется ни одно из отношений приоритетов (пустые ячейки в табл. 14), то обнаружена синтаксическая ошибка, и должна быть вызвана программа восстановления после ошибки.

7.4.4. Нахождение отношений приоритетов операторов

Отношения приоритета операторов можно получить любым подходящим способом. Для языка арифметических выражений, на подобие генерируемого грамматикой (8), можно использовать правила для выбора «правильных» основ, отражающие ассоциативность и приоритет бинарных операторов.

1. Если оператор θ_1 имеет более высокий приоритет, чем оператор θ_2, определяем, что $\theta_1 > \theta_2$ и $\theta_2 <\bullet\ \theta_1$. Например, если оператор * имеет более высокий приоритет, чем оператор +, то * •> + и + <• *. Эти отношения гарантируют, что при получении выражения вида $E + E * E + E$ основой является $E * E$ и именно $E * E$ будет свернуто первым.

2. Если θ_1 и θ_2 представляют собой операторы равного приоритета (это может быть один и тот же оператор), то устанавливаем, что $\theta_1 \bullet> \theta_2$ и $\theta_2 \bullet> \theta_1$, если операторы левоассоциативны, и $\theta_1 <\bullet\ \theta_2$ и $\theta_2 <\bullet\ \theta_1$, если правоассоциативны. Например для левоассоциативных + и – устанавливаем, что + •> – , – •>– и – •> +. Для правоассоциативного оператора ↑ следует принять, что ↑ <• ↑. Такие отношения гарантируют, что в выражении $E – E + E$ основой является $E – E$, а в $E ↑ E ↑ E$ – последнее выражение $E ↑ E$.

3. Для всех операторов θ определяем отношения θ <• *id*, *id* •> θ, θ <• (, (<•θ ,) •> θ •>), θ •> $ и $ <• θ. Кроме того, считаем, что

(=)	$ <•(	$ <• *id*
(<• (	*id* •> $	) •> $
(<• *id*	*id* •>)	) •>)

Эти правила гарантируют, что и *id*, и (*E*) будут приведены к *E*. Кроме того, $ служит как левым, так и правым маркером конца строки, что заставляет основы находиться между ними.

Пример 23. В табл. 15 приведены отношения приоритета операторов для грамматики (7.1) в предположении, что:

1. Оператор ↑ имеет высший приоритет и правоассоциативен;

2. Операторы * и / имеют приоритет следующего уровня и левоассоциативны;

3. Операторы + и - имеют низший приоритет и левоассоциативны.

Пустые ячейки означают ошибки.

Таблица 15

Отношения приоритета операторов

	+	-	*	/	↑	*id*	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
↑	•>	•>	•>	•>	<•	<•	<•	•>	•>
id	•>	•>	•>	•>	•>			•>	•>
(	<•	<•	<•	<•	<•	<•	<•	≐	
)	•>	•>	•>	•>	•>			•>	•>
$	<•	<•	<•	<•	<•	<•	<•		

7.4.5. Обработка ошибок переноса/свертки

При обнаружении ошибок синтаксический анализатор приоритета операторов вызывает программу восстановления после ошибок. При обращении к матрице приоритетов для принятия решения о переносе или свертке можно обнаружить, что между символом на вершине стека и очередным входным символом отношение отсутствуют. Предположим, что на вершине стека находятся *a* и *b* (*b* на вершине стека), во входном потоке – *c* и *d* (*c* – первый), а между *b* и с нет отношений приоритетов. Для восстановления после этой ошибки нужно изменить стек или входной буфер (а может быть, и тот и другой). Можно изменить символы, вставив их в стек или во входной буфер или удалить оттуда. При вставке и изменении следует избегать бесконечного цикла, в котором постоянно вставляются символы в буфер, перенести или свернуть которые невозможно.

Один из способов гарантии отсутствия зацикливания – обеспечение возможности переноса символа после восстановления (если текущий символ – $, следует убедиться, что во входной поток не вставляются новые символы, а стек в конце концов прекращается). Например, если *a* <• *c*, то при входной строке *cd* и в стеке строки *ab* можно снять со стека *b*. Другой способ – удаление из входного потока *c* при *b* <• *d*. Третий способ заключается в поиске такого *e*, что

b <• e <• c, и вставке e во входной поток c. В более общем виде, если единственный символ для вставки найти не удается, можно вставить строку символов, такую, что b <• e_1 <• e_2 <• …<• e_n <• c.

Выбор конкретного действия в конечном счете должен определяться интуицией разработчика компилятора. Для каждой пустой ячейки матрицы приоритетов необходимо определить подпрограмму восстановления после ошибок; одна и та же подпрограмма может использоваться в нескольких местах. Когда синтаксический анализатор рассматривает запись для a и не обнаруживает отношений приоритета между a и b, он наводит указатель на подпрограмму восстановления после этой ошибки.

Пример 24. В матрице приоритетов (табл. 16) показаны только строки и столбцы, имеющие пустые ячейки, которые заполнены именами подпрограмм восстановления после ошибки.

Таблица 16

Матрица приоритета операторов с подпрограммами обработки ошибок

	id	(	)	$
id	*e*3	*e*3	•>	•>
(	<•	<•	≐	*e*4
)	*e*3	*e*3	•>	•>
$	<•	<•	*e*2	*e*1

Сущность этих подпрограмм состоит в следующем.

*e*1: Вызывается при отсутствии выражения в целом. Вставляет *id* во входной поток.

Диагностическое сообщение: *отсутствует операнд*

*e*2: Вызывается, когда выражение начинается с правой скобки. Удаляет «)» из входного потока.

Диагностическое сообщение: *несбалансированная правая скобка*

*e*3: Вызывается, когда *id* или «)» следует за *id* или «(». Вставляет «+» во входной поток.

Диагностическое сообщение: *отсутствует оператор*

*e*4: Вызывается при завершении выражения левой скобкой. Снимает «(» со стека.

Диагностическое сообщение: *отсутствует правая скобка*

Рассмотрим работу этого механизма с неверной входной строкой *id*+). Первые действия, предпринимаемые синтаксическим анализатором – перенос *id*, свертка в *E* и перенос +. После этих действий получаем следующую конфигурацию:

Стек	Вход
$*E*+	)$

Поскольку + •>), вызывается свертка для основы +. Подпрограмма проверки ошибок при свертке должна просмотреть наличие *E* слева и справа от оператора. Обнаружив, что один из них отсутствует, она выдает сообщение об ошибке «*отсутствует операнд*» и все равно выполняет свертку.

Теперь конфигурация принимает вид:

Стек	Вход
$*E*	)$

Между $ и) нет отношений приоритетов (табл. 16), на рис. 35 и запись для этой пары символов – *e*2. Подпрограмма *e*2 выводит диагностическое сообщение «*несбалансированная правая скобка*» и удаляет правую скобку из входного потока. При этом достигается конечная конфигурация синтаксического анализатора:

Стек	Вход
$*E*	$

В качестве технологии синтаксического анализа общего назначения синтаксический анализ приоритета операторов имеет ряд недостатков. Например, с его помощью сложно обрабатывать лексемы вроде знака «минус», который имеет два разных приоритета – в зависимости от того, является ли он унарным или бинарным. Кроме того, класс грамматик, с которыми может работать такой синтаксический анализатор, весьма невелик.

Тем не менее из-за своей простоты эта технология используется для разбора выражений многими компиляторами.

7.4.6. Алгоритм синтаксического анализа простого предшествования

Каждый такт работы анализатора начинается с определения отношения предшествования, которое выполняется для пары, образованной из верхнего символа стека и крайнего левого символа входной строки.

Возможны следующие варианты.

1. Ни одно из отношений не определено. Тогда входная цепочка отвергается.

2. Для пары символов определено отношение $<\bullet$ или $\doteq$, тогда выполняется перенос.

При переносе анализатор записывает символ $<\bullet$ в стек, если соответствующее отношение выполняется для данной пары, после чего переносит в стек левый символ входной цепочки. Во входной цепочке осуществляется сдвиг (переход) к следующему символу.

3. Для пары определено отношение $\bullet>$, тогда выполняется свертка.

При выполнении свертки анализатор работает следующим образом.

3.1. Просматривает символы в стеке, пока не обнаружит символ $<\bullet$. Выделяет подцепочку, состоящую из просмотренных символов (без знака $<\bullet$). Снимает со стека найденные символы.

3.2. Ищет правило вывода грамматики, правая часть которого совпадает с подцепочкой, выделенной на предыдущем шаге. Если такое правило найдено, то переходят к следующему шагу, иначе – к последнему шагу 4.

3.3. Исключает из стека найденные на первом шаге подцепочку и символ <•. В итоге образуется пара, состоящая из символа на верхушке стека и нетерминального символа из левой части продукции, найденной на шаге 3.2.

3.4. Ищет отношение предшествования, выполняющееся для пары, образованной на предыдущем шаге (это только <• или $\doteq$).

Если ни одно отношение предшествования не выполнено, входная цепочка отвергается.

В противном случае в стек записывается символ <•, если для пары выполнено данное отношение, и символ из левой части продукции. На этом свертка заканчивается.

4. Анализируются цепочки, записанные в стеке и на входной ленте. Если в стеке находится цепочка \$ <• *S*, где *S* – стартовый символ грамматики, а на входной ленте только маркер конца строки \$, то входная цепочка принимается (допускается). В противном случае цепочка отвергается.

7.4.7. Алгоритм синтаксического анализа приоритета операторов

Алгоритм аналогичен рассмотренному ранее в п. 7.4.6.

Разница состоит в том, что отношения приоритетов рассматриваются только между терминальными символами грамматики.

При выполнении свертки с вершины стека выбираются все терминальные символы, связанные отношением $\doteq$, совместно с окружающими их нетерминалами.

Так как нетерминальные символы не играют роли в определении отношений между терминалами, можно заменить все нетерминалы одним символом.

7.5. *LR*-анализаторы

LR-технология – наиболее эффективная технология восходящего синтаксического анализа, которая может быть использована для анализа большого класса контекстно-свободных грамматик. Эта технология называется *LR*(*k*)-*анализом*; *L* – сканирование входного потока слева направо, *R* – построение обращенных правых порож-

дений, *k* – число входных символов, которые могут быть просмотрены для принятия решения о способе проведения разбора. Если (*k*) не указывается, подразумевается, что *k* равно 1. *LR*-анализ привлекателен по следующим причинам:

– *LR*-анализаторы могут быть созданы для распознавания, по сути, всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика;

– метод *LR*-анализа – наиболее общий известный метод ПС-анализа без отката, который не уступает в эффективности другим методам этого типа;

– класс грамматик, которые могут быть разобраны с использованием *LR*-метода, представляет собой собственное надмножество класса грамматик, которые могут быть разобраны предиктивными синтаксическими анализаторами;

– *LR*-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока.

Основной недостаток этого метода состоит в том, что ручное построение *LR*-анализатора для грамматики типичного языка программирования требует большого объема работы. Для решения этой задачи нужен специализированный инструмент – генератор *LR*-анализаторов. С помощью такого генератора для разработанной контекстно-свободной грамматики можно автоматически построить ее синтаксический анализатор. Если грамматика содержит неоднозначности или другие конструкции, трудные для разбора сканированием слева направо, генератор в состоянии их локализовать и сообщить о них разработчику.

7.5.1. Алгоритм LR-анализа

Схематически *LR*-анализатор представлен на рис. 23. Он состоит из входного потока, выхода, стека, управляющей программы и таблицы синтаксического анализа, состоящей из двух частей: *действие* (*action*) и *переход* (*goto*). Управляющая программа для всех *LR*-анализаторов одна и та же; изменяются только таблицы синтаксического анализа. Программа синтаксического анализа считывает

символы из входного буфера по одному и использует стек для хранения строк вида $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$ (s_m находится на вершине стека). Каждый символ X является символом грамматики, а каждый s_i – символом *состояния*. Каждый символ состояния обобщает информацию, содержащуюся в стеке ниже его. Комбинация символа состояния на вершине стека и текущего входного символа используется в качестве индекса таблицы синтаксического анализа и определяет дальнейшее действие – перенос или свертку. При реализации грамматические символы не обязаны появляться в стеке, однако для облегчения понимания принципов работы *LR*-анализатора их лучше рассматривать.

Таблица синтаксического анализа состоит из двух частей – функции действий синтаксического анализа a*ction* и функции переходов *goto*. Управляющая программа *LR*-анализатора функционирует следующим образом. Она определяет s_m, текущее состояние на вершине стека, и a_i – текущий входной символ. Затем программа обращается к части *action*[s_m, a_i] – ячейке таблицы действий синтаксического анализа, определяемой состоянием s_m и символом a_i, которая может иметь одно из четырех значений:

1) перенос s, где s – состояние;
2) свертка в соответствии с продукцией $A \rightarrow \beta$;
3) допуск;
4) ошибка.

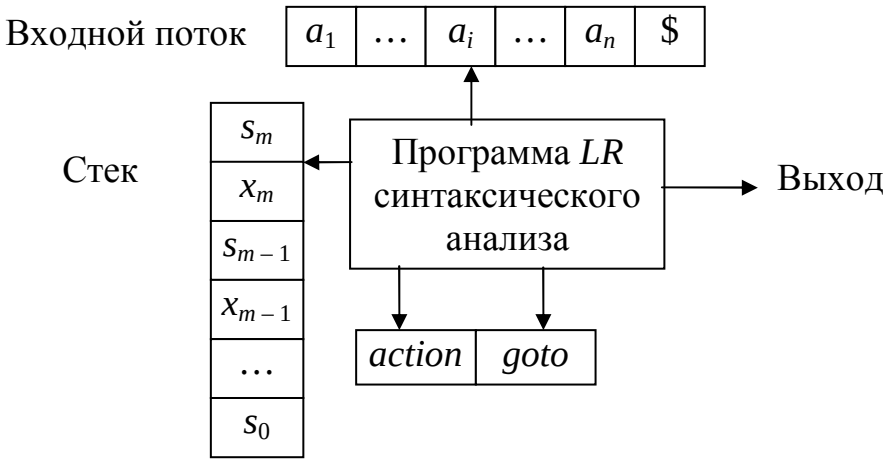


Рис. 23. Модель *LR*-анализатора

Функция *goto* получает в качестве аргументов состояние и символ грамматики и возвращает новое состояние. Функция *goto* таблицы синтаксического анализа, построенная на основе грамматики G с использованием *LR*-метода, представляет собой функцию переходов детерминированного конечного автомата. Начальным состоянием этого ДКА является состояние, изначально размещаемое на вершине стека *LR*-анализатора.

Конфигурация LR-анализатора представляет собой пару, первый компонент которой – содержимое стека, а второй – непросмотренная часть входного потока: ($s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$, $a_i a_{i+1} \ldots a_n$ \$). Эта конфигурация представляет собой правосентенциальную форму $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$, по сути, тем же способом, что и ПС-анализатор; новым является только наличие в стеке символов состояний.

Следующий шаг синтаксического анализатора определяется текущим входным символом a и состоянием на вершине стека s_m в соответствии со значением ячейки таблицы *action*$[s_m, a_i]$. Конфигурации, получаемые после каждого из четырех типов действий, следующие.

1. Если *action*$[s_m, a_i]$ = «перенос s», синтаксический анализатор выполняет перенос, переходя в конфигурацию ($s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i s$, $a_i a_{i+1} \ldots a_n$ \$). Синтаксический анализатор переносит в стек текущий входной символ a_i и очередное состояние s, определяемое значением *action*$[s_m, a_i]$; текущим входным символом становится a_{i+1}.

2. Если *action*$[s_m, a_i]$ = «свертка $A \rightarrow \beta$», то синтаксический анализатор выполняет свертку, переходя в конфигурацию ($s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s$, $a_i a_{i+1} \ldots a_n$ \$), где s = *goto*$[s_{m-r}, A]$, а r – длина β правой части продукции. Здесь синтаксический анализатор вначале снимает со стека $2r$ символов (r символов состояний и r символов грамматики), выводя на вершину стека состояние s_{m-r}. Затем он вносит в стек символ A (левую часть продукции) и s – запись из ячейки *goto*$[s_{m-r}, A]$. Текущий входной символ при этом не изменяется. Последовательность снимаемых со стека символов грамматики $X_{m-r+1} \ldots X_m$ всегда соответствует правой части продукции свертки.

3. Если *action*[s_m, a_i] = «допуск», синтаксический анализ завершает свою работу.

4. Если *action*[s_m, a_i] = «ошибка», синтаксический анализатор обнаружил ошибку и вызывает подпрограмму восстановления после нее.

Все *LR*-анализаторы ведут себя одинаково; единственная разница между ними заключается в таблицах *action* и *goto*.

7.5.2. Построение таблиц SLR-анализа

Существуют разные методы, отличающиеся по мощности и сложности реализации. Самый простой в реализации метод «простого *LR*» (simple *LR*, *SLR*) анализа, самый слабый по количеству грамматик, с которыми он работает. Таблица, построенная таким методом, называется *SLR*-таблицей, а синтаксический анализатор, работающий с *SLR*-таблицей, – *SLR*-анализатором.

При построении таблиц вводится понятие пункта.

LR(0)-*пункт*, или *элемент* грамматики *G* – продукция *G* с точкой в некоторой позиции правой части.

Например, продукция $A \rightarrow XYZ$ дает четыре пункта:

$A \rightarrow \bullet\, XYZ$
$A \rightarrow X \bullet YZ$
$A \rightarrow XY \bullet Z$
$A \rightarrow XYZ \bullet$

Продукция $A \rightarrow \lambda$ генерирует только один пункт: $A \rightarrow \bullet$.

Пункт указывает, какую часть продукции мы уже увидели в данной точке в процессе синтаксического анализа.

Например, первый пункт, приведенный выше, определяет, что во входном потоке мы ожидаем встретить строку, порождаемую *XYZ*. Второй пункт указывает, что уже есть строка, порожденная *X*, и мы ожидаем получить из входного потока строку, порождаемую *YZ*.

Основная идея *SLR*-метода состоит в том, чтобы вначале на базе грамматики построить детерминированный конечный автомат для распознавания активных префиксов. Группируем пункты в множества, которые приводят к состояниям *SLR*-анализатора. Пункты могут рассматриваться как состояния недетерминированного конеч-

ного автомата, распознающего активные префиксы. Система *LR*(0)-пунктов, которая называется *канонической,* обеспечивает основу для построения *SLR*-анализаторов.

Для построения канонической *LR*(0)-системы грамматики необходимо определить *расширенную грамматику* и две функции – *closure* и *goto*.

Если *G* – грамматика со стартовым символом *S*, то *G'* – *расширенная грамматика* грамматики *G*, представляет собой *G* с новым стартовым символом *S'* и продукцией $S' \rightarrow S$. Назначение этой новой стартовой продукции – указать синтаксическому анализатору, когда он должен прекратить разбор и объявить о допущении входной строки. Таким образом, допуск строки происходит тогда, когда синтаксический анализатор выполняет свертку, соответствующую продукции $S' \rightarrow S$.

7.5.3. Операция замыкания

Если *I* – множество пунктов грамматики *G*, то *closure*(*I*) – множество пунктов, построенное из *I* по следующим правилам.

1. Изначально в *closure*(*I*) входят все пункты из *I*.

2. Если $A \rightarrow \alpha \bullet B\beta$ входит в *closure*(*I*) и $B \rightarrow \gamma$ представляет собой продукцию, то добавляем в *closure*(*I*) пункт $B \rightarrow \bullet\gamma$ (если его там еще нет). Это правило применяется до тех пор, пока не будут внесены все возможные пункты в *closure*(*I*).

Наличие $A \rightarrow \alpha \bullet B\beta$ в *closure*(*I*) указывает, что в некоторый момент в процессе синтаксического анализа можно встретить во входном потоке подстроку, выводимую из $B\beta$. Но если имеется продукция $B \rightarrow \gamma$, то, естественно, можно встретить в этот момент строку, выводимую из γ, поэтому $B \rightarrow \bullet\gamma$ включается в *closure*(*I*) [3].

Пример 24. Рассмотрим расширенную грамматику арифметических выражений:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned} \qquad (9)$$

Эту грамматику следует записать в ином виде:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $T \rightarrow (E)$
(6) $T \rightarrow id$

Если I представляет собой множество из одного пункта $\{[E' \rightarrow \bullet E]\}$, то *closure*($I$) содержит пункты:

$E' \rightarrow E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

Здесь $E' \rightarrow \bullet E$ помещается в *closure*(I) в соответствии с правилом (1). Поскольку E расположено непосредственно за точкой, то, согласно правилу (2), добавляются также E-продукции с точкой слева, т.е. $E \rightarrow \bullet E + T$ и $E \rightarrow \bullet T$. Так как среди пунктов имеется T, следующее за точкой, мы добавляем $T \rightarrow \bullet T * F$ и $T \rightarrow \bullet F$ и аналогично $F \rightarrow \bullet (E)$ и $F \rightarrow \bullet id$. Больше добавлять нечего в *closure* (I).

7.5.4. Операция goto

Второй функцией является *goto*(I, X), где I является множеством пунктов, а X – символом грамматики, *goto*(I, X) определяется как замыкание множества всех пунктов $[A \rightarrow \alpha X \bullet \beta]$, таких, что $[A \rightarrow \alpha \bullet X \beta]$ принадлежит множеству I.

Пример 25. Если I представляет собой множество из двух пунктов $\{[E' \rightarrow E\bullet], [E \rightarrow E\bullet + T]\}$, то *goto*($I$, +) состоит из:

$E \rightarrow E + \bullet T$
$E \rightarrow \bullet T * F$
$E \rightarrow \bullet F$
$E \rightarrow \bullet (E)$
$E \rightarrow \bullet id$

Вычислим *goto*(*I*, +), рассмотрев пункты из *I*, у которых сразу за точкой следует символ +. В отличие от пункта $E' \to E\bullet$, таким пунктом является $E \to E\bullet + T$. Мы перемещаем точку за символ +, получая $\{E \to E + \bullet T\}$, и рассматриваем замыкание этого множества.

7.5.5. Построение множеств пунктов

Строим *C* – каноническую систему множества *LR*(0)-пунктов для расширенной грамматики *G'*.

Пример 26. Каноническая система множеств *LR*(0)-пунктов для грамматики (9) представлена на рис. 24, а функция *goto* в виде диаграммы переходов детерминированного конечного автомата – на рис. 25.

I_0: $E' \to \bullet E$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet(E)$
$F \to \bullet id$

I_1: $E' \to E\bullet$
$E \to E\bullet + T$

I_2: $E \to T\bullet$
$T \to T\bullet * F$

I_3: $T \to F\bullet$

I_4: $F \to (\bullet E)$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet(E)$
$F \to \bullet id$

I_5: $F \to id\bullet$

I_6: $E \to E + \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet(E)$
$F \to \bullet id$

I_7: $T \to \bullet T * F$
$F \to \bullet(E)$
$F \to \bullet id$

I_8: $F \to \bullet(E)$
$E \to \bullet E + T$

I_9: $E \to E + T\bullet$
$T \to T\bullet * F$

I_{10}: $T \to T * F\bullet$

I_{11}: $F \to (E)\bullet$

Рис. 24. Каноническая *LR*(0)-система для грамматики (9)

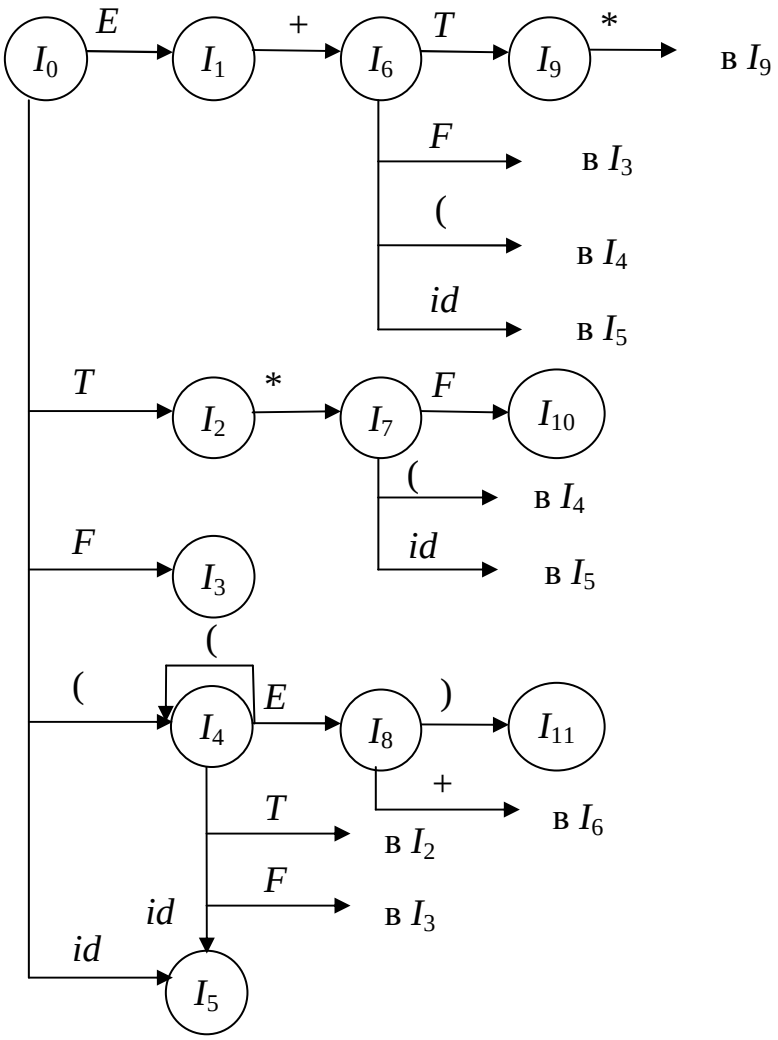


Рис. 25. Диаграмма переходов детерминированного конечного автомата для активных префиксов

7.5.6. Построение таблицы разбора SLR-анализа

Рассмотрим, как построить функции *action* и *goto SLR*-анализа по детерминированному конечному автомату, распознающему активные префиксы. Грамматику G расширяем до грамматики G' и на основе G' строим каноническую систему множеств пунктов для G'. Строим *action*, функцию действий синтаксического анализа, и *goto*, функцию переходов. Потребуется множество FOLLOW(A) для каждого нетерминала A грамматики.

Алгоритм построение таблицы *SLR*-анализа

Вход. Расширенная грамматика G'.

Выход. Функции *action* и *goto* таблицы *SLR*-анализа для грамматики G'.

Метод.

1. Построим $C = \{I_0, I_1, ..., I_n\}$ – систему множеств *LR*(0)-пунктов для грамматики G'.

2. Состояние i строится на основе I_i. Действия синтаксического анализа для состояния i определяются следующим образом:

а) если $[A \to \alpha \bullet a\beta] \in I_i$, и $goto(I_i, a) = I_j$, то определить $action[i, a]$ как «перенос j»; здесь a – терминальный символ;

б) если $[A \to \alpha\bullet] \in I_i$, то определить $action[i, a]$ как «свертка $A \to \alpha$» для всех a из FOLLOW(A); здесь A не должно быть S';

в) если $[S' \to S\bullet] \in I_i$, то определить $action[i, \$]$ как «допуск».

Если по этим правилам генерируются конфликтующие действия, то грамматика не является *SLR*(l). Алгоритм не в состоянии построить синтаксический анализатор для нее.

3. Переходы *goto* для состояния i и всех нетерминалов A строятся по правилу: если $goto(I_i, A) = I_j$, то $goto[i, A] = j$.

4. Все записи, не определенные по правилам (2) и (3), указываются как «ошибка».

5. Начальное состояние синтаксического анализатора представляет собой состояние, построенное из множества пунктов, содержащего $[S' \to S\bullet]$ [3].

Пример 27. Построим *SLR*-таблицу для грамматики (9).

Вначале рассматривает множество пунктов I_0:

$E' \to \bullet E$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

Пункт $F \rightarrow \bullet(E)$ дает запись *action*[0, ( ] = «перенос 4», пункт $F \rightarrow \bullet\ id$ – запись *action*[0, *id*] = «перенос 5». Другие пункты I_0 к записям *action* не приводят.

Рассматриваем I_1:

$E' \rightarrow E\bullet$
$E \rightarrow E\bullet + T$

Первый пункт дает *action*[1, $] – «допуск», а второй – *action*[1, +] – «перенос б». Переход к I_2 дает:

$E \rightarrow T\bullet$
$T \rightarrow T\bullet * F$

Поскольку FOLLOW(*E*) ={$, +,)}, из первого пункта следует *action*[2, $] = *action*[2, +] = *action*[2,)] = «свертка $E \rightarrow T$». Второй пункт дает *action*[2, *] = «перенос». Продолжая таким образом рассмотрение множества пунктов, получаем табл. 17. В ней номера продукций в свертках те же, что и номера, под которыми продукции приведены в исходной грамматике, т.е. $E \rightarrow E + T$ имеет номер 1, $E \rightarrow T$ – номер 2 и т.д.

Таблица 17

Таблица разбора синтаксического анализа грамматики арифметических выражений

Состояние	*action*						*goto*		
	Id	+	*	(	)	$	*E*	*T*	*F*
0	*p*5			*p*4			1	2	3
1		*p*6				acc			
2		*s*2	*s*7		*s*2	*s*2			
3		*s*4	*s*4		*s*4	*s*4			
4	*P*5			*p*4			8	2	3
5		*s*6	*s*6		*s*6	*s*6			

Продолжение табл. 17

Состояние	*action*						*goto*		
6	*p*5							9	3
7	*p*5			*p*4					10
8		*s*6		*p*4	*p*11				
9		*s*1	*s*7		*s*1	*s*1			
10		*s*3	*s*3		*s*3	*s*3			
11		*s*5	*s*5		*s*5	*s*5			

Примечание. *pi* – перенос и *i*-е состояние на вершине стека; *sj* – свертка в соответствии с продукцией с номером *j*; acc – допуск входной строки; пустая ячейка – ошибка.

Значение *goto*[*s*, *a*] для терминала *a* находится в поле *action*, связанном с переносом для входного символа *a* и состояния *s*. Поле *goto* дает значения *goto*[*s*, *A*] для нетерминалов *A*.

Для входной строки *id* * *id* + *id* последовательность содержимого стека и входной строки показана в табл. 18. Например, в строке (1) *LR*-анализатор находится в состоянии 0 с первым входным символом *id*. Действие в строке 0 и столбце *id* части *action* (см. табл. 17) – *p*5, означает перенос и внесение в стек состояния 5. В строке (2) выполняется внесение в стек *id* и *p*5 и удаление *id* из входного потока.

После этого текущим входным символом становится *; действие для состояния *p*5 и входного символа * – *s*6, т.е. свертка согласно продукции $F \rightarrow id$. При этом со стека снимаются два символа (символ состояния и символ грамматики), и на вершине стека появляется состояние 0. Поскольку *goto*[0, *F*] равно *p*3, в стек вносятся *F* и 3; получается конфигурация, показанная в строке (3). Остальные строки в табл. 18 получены аналогично.

Таблица 18

Действия *LR*-анализатора по разбору строки *id * id + id*

№ п/п	Стек	Входной поток	Действие
1	0	*id* * *id* + *id* $	Перенос
2	0 *id* 5	* *id* + *id* $	Свертка по $F \to id$
3	0 *F* 3	* *id* + *id* $	Свертка по $T \to F$
4	0 *T* 2	* *id* + *id* $	Перенос
5	0 *T* 2 * 7	*id* + *id* $	Перенос
6	0 *T* 2 *7 *id* 5	+ *id* $	Свертка по $F \to id$
7	0 *T* 2 * 7 *F* 10	+ *id* $	Свертка по $T \to T * F$
8	0 *T* 2	+ *id* $	Свертка по $E \to T$
9	0 *E* 1	+ *id* $	Перенос
10	0 *E* 1 + 6	*id* $	Перенос
11	0 *E* 1 + 6 *id* 5	$	Свертка по $F \to id$
12	0 *E* 1 + 6 *F* 3	$	Свертка по $T \to F$
13	0 *E* 1 + 6 *T* 9	$	$E \to E + T$
14	0 *E* 1	$	Допуск

Пример 28. Любая *SLR*(l)-грамматика однозначна, но имеется множество однозначных грамматик, не являющихся *SLR*(l). Рассмотрим грамматику с продукциями

$$\begin{aligned} S &\to L = R \\ S &\to R \\ L &\to * R \\ L &\to \mathbf{id} \\ R &\to L \end{aligned} \qquad (10)$$

Каноническая система множеств *LR*(1) пунктов для этой грамматики показана рис. 26.

I_0: $S' \to \bullet S$
$S \to \bullet L = R$
$S \to \bullet R$
$L \to \bullet * R$
$L \to \bullet id$
$R \to \bullet L$

I_1: $S' \to \bullet S$

I_2: $S \to L \bullet = R$
$R \to L \bullet$

I_3: $S \to R \bullet$

I_4: $L \to * \bullet R$
$R \to \bullet L$
$L \to \bullet * R$
$L \to \bullet id$

I_5: $L \to id \bullet$

I_6: $S \to L = \bullet R$
$R \to \bullet L$
$L \to \bullet * R$
$L \to \bullet id$

I_7: $L \to * R \bullet$

I_8: $R \to L \bullet$

I_9: $S \to L = R \bullet$

Рис. 26. Каноническая *LR*(0)-система грамматики (10)

Рассмотрим множество пунктов I_2. Первый пункт в этом множестве устанавливает *action*[2, =] как «перенос 6». Поскольку FOLLOW(*R*) содержит «=» (чтобы увидеть это, рассмотрите $S \Rightarrow L = = R \Rightarrow *R = R$), второй пункт определяет *action*[2, =] как «свертка $R \to L$». Следовательно, запись *action*[2, =] определена дважды, а поскольку для нее имеется и запись переноса, и запись свертки, состояние 2 приводит к конфликту «перенос/свертка» при входном символе =.

Грамматика (10) не является неоднозначной. Обнаруженный конфликт порождается тем, что метод *SLR* недостаточно мощный, чтобы запомнить левый контекст, необходимый для принятия решения о действиях синтаксического анализатора при входном символе = и просмотренной строке, выводимой из *L*.

Пример 29. В примере 28 в состоянии 2 был пункт $R \to L \bullet$, который мог соответствовать приведенной выше продукции $A \to \alpha$, и *a* мог быть знаком «=», принадлежащим FOLLOW(*R*). Таким образом, *SLR*-анализатор в состоянии 2 с очередным входным символом = вызывает свертку по продукции $R \to L$ (вызывается также и перенос – в соответствии с пунктом $S \to L \bullet = R$ в состоянии 2). Однако в грамматике из примера 28 правосентенциальная фор-

ма, начинающаяся с R = ..., отсутствует. Следовательно, состояние 2, соответствующее только активному префиксу L, не должно вызывать свертку этого L в R.

Состояние может содержать дополнительную информацию, которая позволит исключить некоторые из таких некорректных сверток по продукции $A \rightarrow \alpha$. Разделяя при необходимости состояния, можно заставить каждое состояние *LR*-анализатора точно указывать, какой входной символ может следовать за основой α (для которой существует возможная свертка в A).

Дополнительная информация внедряется в состояние путем переопределения пунктов для включения терминального символа в качестве второго компонента. Общий вид пункта: $[A \rightarrow \alpha \bullet \beta, a]$, где $A \rightarrow \alpha\beta$ представляет собой продукцию, a – терминал или маркер конца строки \$. Такой объект назовем *LR*(1)-*пунктом*. Здесь 1 означает длину второго компонента, называемого *предпросмотром пункта*. Предпросмотр не играет роли в пункте вида $[A \rightarrow \alpha \bullet \beta, a]$, где $\beta \neq \lambda$, однако пункт $[A \rightarrow \alpha\bullet, a]$ вызывает свертку по продукции $A \rightarrow \alpha$ только тогда, когда очередной входной символ – a. Следовательно, свертка по продукции $A \rightarrow \alpha$ выполняется только при тех входных символах a, для которых $[A \rightarrow \alpha\bullet, a]$ является *LR*(1)-пунктом в состоянии на вершине стека. Множество таких a всегда будет подмножеством FOLLOW(A), но может быть и собственным подмножеством.

7.5.7. LR-грамматики

Грамматика, для которой можно построить таблицу синтаксического анализа, называется *LR-грамматикой*. Для того чтобы грамматика была *LR*-грамматикой, достаточно, чтобы *ПС*-анализатор, читающий поток слева направо, был способен распознавать основы при их появлении в стеке.

LR-анализатор не должен просматривать весь стек, чтобы распознать появление основы на вершине стека; более того, символ состояния на вершине стека содержит всю необходимую информацию. Если можно распознать основу, зная только символы грамматики в стеке, то можно построить конечный автомат, который в состоянии определить основу на вершине стека путем считывания

грамматических символов в стеке сверху вниз. По сути, таким автоматом является функция *goto* таблицы *LR*-анализа. Однако автомату также не нужно читать стек при каждом переходе. Символ состояния на вершине стека представляет собой состояние распознающего основы конечного автомата, в котором бы он оказался после считывания грамматических символов в стеке снизу вверх. Следовательно, *LR*-анализатор может определить по состоянию на вершине стека все, что надо знать о содержимом стека.

Другой источник информации, который используется *LR*-анализатором для принятия решения о переносе/свертке, – очередные k символов входного потока. Практический интерес представляют случаи, когда $k = 0$ и $k = 1$. Грамматика, для разбора которой *LR*-анализатору требуется просмотр до k символов входного потока на каждом шаге, называется *LR*(k)-*грамматикой*.

Существует важное отличие между *LL*- и *LR*-грамматиками. Для того чтобы грамматика представляла собой *LR*(k)-грамматику, необходимо распознать появление правой части продукции, видя все, что из нее порождено, а также k входных символов. Это требование существенно менее строго, чем требование *LL*(k)-грамматики, где нужно распознать использование продукции, видя только k первых символов того, что порождено ее правой частью. Поэтому *LR*-грамматики могут описывать больше языков, чем *LL*-грамматики.

Вопросы

1. Какой алгоритм лежит в основе всех восходящих методов синтаксического анализа?
2. Какие преимущества имеют распознаватели на основе грамматик операторного предшествования перед распознавателями на основе грамматик простого предшествования?
3. Почему любая *LL*-грамматика является *LR*-грамматикой?
4. Как определяются отношения предшествования? Какие средства существуют для определения отношений предшествования?
5. Как используются отношения предшествования при выполнении синтаксического анализа?
6. Что является пунктом грамматики? Что показывает тот или иной пункт?
7. В чем особенности *LR*-технологий, используемых при разработке компиляторов?
8. Сравните рассмотренные методы синтаксического анализа. Отметьте достоинства и недостатки каждого метода.

8. ГЕНЕРАЦИЯ КОДА. МЕТОДЫ ГЕНЕРАЦИИ КОДА

8.1. Общие принципы генерации кода

Генерация объектного кода – это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. Генерация объектного кода порождает результирующую объектную программу на языке машинных команд. Внутреннее представление программы может иметь любую структуру в зависимости от реализации компилятора, в то время как результирующая программа всегда представляет собой линейную последовательность команд. Поэтому генерация объектного кода (объектной программы) в любом случае должна выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки.

Генерация объектного кода выполняется после того, как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: распределено адресное пространство под функции и переменные, проверено соответствие имен и типов переменных, констант.

В идеале компилятор должен выполнить синтаксический разбор всей входной программы, затем выполнить семантический анализ, после чего приступить к подготовке генерации и непосредственно к генерации кода. Однако такая схема работы компилятора практически почти никогда не применяется. В общем случае ни один семантический анализатор и ни один компилятор не способны проанализировать и оценить смысл всей входной программы в целом. Формальные методы анализа семантики применимы только к очень незначительной части возможных программ. Поэтому у компилятора нет практической возможности порождать эквивалентную выходную программу на основании всей входной программы.

Как правило, компилятор выполняет генерацию результирующего кода поэтапно, на основе законченных синтаксических конструкций входной программы:

1) выделяет законченную синтаксическую конструкцию из текста входной программы;

2) порождает для нее фрагмент результирующего кода и помещает его в текст выходной программы;

3) переходит к следующей синтаксической конструкции.

Так продолжается до тех пор, пока не будет разобрана вся входная программа. В качестве анализируемых законченных синтаксических конструкций выступают операторы, блоки операторов, описания процедур и функций. Их конкретный состав зависит от входного языка и реализации компилятора.

Смысл (семантику) каждой такой синтаксической конструкции входного языка можно определить, исходя из ее типа, а тип определяется синтаксическим анализатором на основании грамматики входного языка. Примерами типов синтаксических конструкций могут служить операторы цикла, условные операторы, операторы выбора и т.д. Одни и те же типы синтаксических конструкций характерны для различных языков программирования, при этом они различаются синтаксисом (который задается грамматикой языка), но имеют сложный смысл (который определяется семантикой). В зависимости от типа синтаксической конструкции выполняется генерация кода результирующей программы, соответствующего данной синтаксической конструкции. Для семантически схожих конструкций различных входных языков может порождаться типовой результирующий код.

8.2. Внутреннее представление программы

Результатом работы распознавателя КС-грамматики – входного языка является последовательность правил грамматики, примененных для построения входной цепочки. Зная тип распознавателя по найденной последовательности, можно построить цепочку вывода или дерево вывода. В этом случае дерево вывода выступает в качестве дерева синтаксического разбора и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Дерево вывода содержит массу избыточной информации, которая для дальнейшей работы компилятора не требуется. Эта информация включает в себя все нетерминальные символы, содержащиеся в узлах дерева, – после того как дерево построено, они не несут никакой смысловой нагрузки и не представляют интереса для дальнейшей работы.

Для полного представления о типе и структуре найденной и разобранной синтаксической конструкции входного языка в принципе достаточно знать последовательность номеров правил грамматики, примененных для ее построения. Однако форма представления этой информации может быть различной в зависимости от реализации самого компилятора и от фазы компиляции. Эта форма называется *внутренним представлением программы* (иногда используются также термины «промежуточное представление» или «промежуточная программа»).

8.3. Способы внутреннего представления программ

Все внутренние представления программы обычно содержат в себе две принципиально различные вещи – операторы и операнды. Различия между формами внутреннего представления заключаются в том, как операторы и операнды соединяются между собой. Операторы и операнды должны отличаться друг от друга, если они встречаются в любом порядке. Что будет выступать в роли операндов и операторов, решает разработчик компилятора, который руководствуется семантикой входного языка.

Известны следующие формы внутреннего представления языка:

– связочные списочные структуры, представляющие собой синтаксические деревья;

– многоадресный код с явно именуемым результатом (тетрады);

– многоадресный код с неявно именуемым результатом (триады);

– косвенные триады;

– обратная (постфиксная) польская запись операций;

– ассемблерный код или машинные команды.

В каждом конкретном компиляторе может использоваться одна из этих форм, выбранная разработчиками. Но чаще всего компилятор не ограничивается использованием только одной формы для внутреннего представления программы. На различных фазах компиляции могут использоваться различные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую. Не все из перечисленных форм широко используются в современных компиляторах. Некоторые компиляторы, незначительно оптимизирующие результирующий код, генерируют объектный код по мере разбора исходной программы. В этом случае применяется *схема СУ-компиляции*, когда фазы синтаксического разбора, семантического анализа, подготовки и генерации объектного кода совмещены в одном проходе компилятора. Тогда внутреннее представление программы существует только в виде последовательности шагов алгоритма. В любом случае компилятор всегда будет работать с представлением программы в форме машинных команд, иначе он не сможет построить результирующую программу.

8.4. Синтаксические деревья

Синтаксическое дерево (дерево операций) – это структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций.

8.4.1. Дерево разбора. Преобразование дерева разбора в дерево операций

В синтаксическом дереве внутренние узлы (вершины) соответствуют операциям, а листья представляют собой операнды. Как правило, листья синтаксического дерева связаны с записями в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка программирования, на котором написана исходная программа.

Синтаксические деревья могут быть построены компилятором для любой части входной программы. Не всегда синтаксическому

дереву должен соответствовать фрагмент кода результирующей программы (например, возможно построение синтаксических деревьев для декларативной части языка). В этом случае операции, имеющиеся в дереве, не требуют порождения объектного кода, но несут информацию о действиях, которые должен выполнить сам компилятор над соответствующими элементами. В том случае, когда синтаксическому дереву соответствует некоторая последовательность операций, влекущая порождение фрагмента объектного кода, говорят о дереве операций.

Дерево операций можно непосредственно построить из дерева вывода, порожденного синтаксическим анализатором. Для этого достаточно исключить из дерева вывода цепочки нетерминальных символов, а также узлы, не несущие семантической (смысловой) нагрузки при генерации кода. Примером таких узлов могут служить различные скобки, которые изменяют порядок выполнения операций и операторов, но после построения дерева никакой смысловой нагрузки не несут, так как им не соответствует никакой объектный код.

То, какой узел в дереве является операцией, а какой – операндом, невозможно определить из грамматики, описывающей синтаксис входного языка. Также ниоткуда не следует, каким операциям должен соответствовать объектный код в результирующей программе, а каким – нет. Все это определяется только исходя из семантики «смысла» языка входной программы. Поэтому только разработчик компилятора может четко определить, как при построении дерева операций должны различаться операнды и сами операции, а также то, какие операции являются семантически не значащими для порождения объектного кода.

Алгоритм преобразования дерева вывода в дерево операций

1. Если в дереве больше не содержится узлов, помеченных нетерминальными символами, то выполнение алгоритма завершено, иначе – перейти к шагу 2.

2. Выбрать крайний левый узел дерева, помеченный нетерминальным символом грамматики и сделать его текущим. Перейти к шагу 3.

3. Если текущий узел имеет только один нижележащий узел, то текущий узел необходимо удалить из дерева, а связанный с ним узел присоединить к узлу вышележащего уровня (исключить из дерева цепочку) и вернуться к шагу 1; иначе – перейти к шагу 4.

4. Если текущий узел имеет нижележащий узел (лист дерева), помеченный терминальным символом, который не несет семантической нагрузки, тогда этот лист нужно удалить из дерева и вернуться к шагу 3; иначе – перейти к шагу 5.

5. Если текущий узел имеет один нижележащий узел (лист дерева), помеченный терминальным символом, обозначающим знак операции, а остальные узлы помечены как операнды, то узел, помеченный знаком операции, надо удалить из дерева, текущий узел пометить этим знаком операции и перейти к шагу 1; иначе – перейти к шагу 6.

6. Если среди нижележащих узлов для текущего узла есть узлы, помеченные нетерминальными символами грамматики, то необходимо выбрать крайний левый среди этих узлов, сделать его текущим узлом, перейти к шагу 3; иначе – выполнение алгоритма завершено.

Этот алгоритм всегда работает с узлом дерева, который считается текущим и стремится исключить из дерева все узлы, помеченные нетерминальными символами. То, какие из символов считать семантически незначащими, а какие считать знаками операций, решает разработчик компилятора. Если семантика языка задана корректно, то в результате работы алгоритма из дерева будут исключены все нетерминальные символы.

Пример синтаксического дерева, построенного для цепочки $(a + a) * b$ из языка, заданного различными вариантами грамматики арифметических выражений, представлен на рис. 27.

В результате применения алгоритма преобразования деревьев синтаксического разбора, в дерево операций, получим дерево операции, представленное на рис. 27. Несмотря на то, что исходные

синтаксические деревья имели различную структуру, зависящую от используемой грамматики, результирующее дерево операций всегда имеет одну и ту же структуру, зависящую только от семантики входного языка.

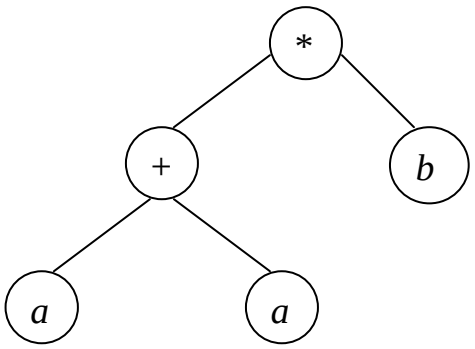


Рис. 27. Пример дерева операций для языка арифметических выражений

Дерево операций является формой внутреннего представления программы, которой удобно пользоваться на этапах синтаксического разбора, семантического анализа и подготовки к генерации кода, когда еще нет необходимости работать непосредственно с кодами команд результирующей программы.

Преимущества внутреннего представления в виде дерева операций:

1) четко отражает связь всех операций между собой, поэтому его удобно использовать для преобразований, связанных с перестановкой и переупорядочиванием операций без изменений конечного результата;

2) это машинно-независимая форма внутреннего представления программы.

Недостаток синтаксических деревьев заключается в том, что они представляют собой сложные связанные структуры, а поэтому не могут быть тривиальным образом преобразованы в линейную последовательность команд результирующей программы. Тем не менее они удобны при работе с внутренним представлением программы на тех этапах, когда нет необходимости непосредственно обращаться к командам результирующей программы.

Синтаксические деревья могут быть преобразованы в другие формы внутреннего представления программы, представляющие собой линейные списки, с учетом семантики входного языка. Эти преобразования выполняются на основе принципов СУ-компиляции.

8.5. Трехадресный код. Типы трехадресных инструкций

Трехадресный код представляет собой последовательность инструкций вида

$$x := y \; op \; z,$$

где x, y и z – имена, константы или временные переменные, генерируемые компилятором; op – некоторый оператор (например, арифметический оператор для работы с числами с фиксированной или плавающей точкой или оператор для работы с логическими значениями). Выражение исходного языка наподобие $x + y * z$ может быть транслировано в следующую последовательность:

$$t_1 := y * z;$$
$$t_2 := x + t_1.$$

Здесь t_1 и t_2 – сгенерированные компилятором временные имена. Использование имен для вычисленных программой промежуточных значений обеспечивает трехадресному коду, в отличие от постфиксной записи, возможность легкого переупорядочения.

Термин «трехадресный код» отражает тот факт, что каждая инструкция обычно содержит три адреса – два для операндов и один для результата.

Список некоторых основных трехадресных инструкций, используемых в большинстве языков программирования:

1. Инструкции присвоения вида $x := y \; op \; z$, где op – бинарная арифметическая или логическая операция.

2. Инструкция присвоения вида *x* := *op y*, где *op* – унарная операция. Основные унарные операции включают унарный минус, логическое отрицание, операторы сдвига и операторы преобразования, которые, например, преобразуют число с фиксированной точкой в число с плавающей точкой.

3. Инструкции копирования вида *x* : = *y*, в которых значение *y* присваивается *x*.

4. Безусловный переход *goto L*. После этой инструкции будет выполнена трехадресная инструкция с меткой *L*.

5. Условный переход типа *if x relop y goto L*. Эта инструкция применяет оператор отношения *relop* (<, >= и т.п.) к *x* и *y*, и следующей выполняется инструкция с меткой *L*, если соотношение *x relop y* верно. В противном случае выполняется следующая за условным переходом инструкция.

6. Индексированные присвоения типа *x* := *y*[*i*]; *x*[*i*] := *y*. Первая инструкция присваивает *x* значение, находящееся в *i*-й ячейке памяти по отношению к *y*. Инструкция *x*[*i*] : = *y* заносит в *i*-ю ячейку памяти по отношению к *x* значение *y*. В обеих инструкциях *x*, *y* и *i* ссылаются на объекты данных.

7. Присвоение адресов и указателей вида *x* : = &*y*, *x* : = **y* и **x* : = *y*. Первая инструкция устанавливает значение *x* равным положению *y* в памяти. Предположительно, *y* представляет собой имя, возможно временное, обозначающее выражение с *l*-значением типа *A* [*i*, *j*], а *x* – имя указателя или временное имя. Таким образом, *r*-значение *x* представляет собой значение некоторого объекта. Во второй инструкции под *y* подразумевается указатель или временная переменная, *l*-значение которой представляет собой местоположение ячейки памяти. В результате *l*-значение *x* становится равным содержимому этой ячейки. И наконец, инструкция **x* := *y* устанавливает *l*-значение объекта, указываемого *x*, равным *l*-значению *y*.

Выбор приемлемых операторов представляет собой важный вопрос в создании промежуточного представления. Очевидно, что множество операторов должно быть достаточно богатым, чтобы позволить реализовать все операции исходного языка. Небольшое множество операторов легче реализуется на новой целевой машине,

однако ограниченное множество инструкций может привести к генерации длинных последовательностей инструкций промежуточного представления для некоторых конструкций исходного языка и добавить работы оптимизатору и генератору целевого кода.

8.6. Тетрады – многоадресный код с явно именуемым результатом

Тетрады представляют собой запись операций в форме из четырех составляющих: операция, два операнда и результат операции. Например, тетрады могут выглядеть так: <операция1> (<операнд1>, <операнд2>, <результат>).

Тетраду можно рассматривать как запись с четырьмя полями: *op*, *arg*1, *arg*2 и *result.* Поле *op* содержит внутренний код оператора. Трехадресная инструкция $x := y\ op\ z$ представляется размещением y в *arg*1, z – в *arg*2 и x – в *result.* Инструкции с унарным оператором наподобие $x := -y$ или $x := y$ не используют *arg*2. Условные и безусловные переходы помещают в *result* целевую метку. Присвоение вида $a := b^* - c + b^* - c$ представлено четырехадресным кодом в табл. 19. Содержимое полей *arg*1, *arg*2 и *result* обычно является указателем на записи в таблице символов для имен, представленных этими полями. В этом случае временные имена должны быть внесены в таблицу символов при их создании.

Таблица 19

Представление трехадресных инструкций тетрадами

	op	*arg*l	*arg*2	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Тетрады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме тетрад, они вычисляются последовательно одна за другой. Каждая тетрада вычисляется так: выполняется операция, заданная тетрадой над операндами; ее результат помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации).

Результат вычисления тетрады никогда не может быть опущен, иначе тетрада полностью теряет смысл. Порядок вычисления тетрад может быть изменен, но только если допустить наличие тетрад, целенаправленно изменяющих этот порядок.

Преимущество тетрад:

1. Тетрады представляют собой линейную последовательность. Поэтому для них несложно написать алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы (либо последовательность команд ассемблера).

2. Тетрады представляют собой машинно-независимую форму внутреннего представления программы, так как не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа.

Недостатки представления в форме тетрад заключаются в том, что они требуют бóльше памяти, чем триады, они также не отражают явно взаимосвязь операций между собой. Кроме того, есть сложности с преобразованием тетрад в машинный код, так как они плохо отображаются в команды ассемблера и машинные коды, поскольку в наборах команд большинства современных компьютеров редко встречаются операции с тремя операндами.

Пример. Выражение *A*:= *B***C* + *D* – *B* *10, записанное в виде тетрад, имеет вид:

1) * (*B*, *C*, *T*1)
2) + (*T*1, *D*, *T*2)
3) * (*B*, 10, *T*3)
4) – (*T*2, *T*3, *T*4)
5) : = (*T*4, 0, *A*)

Здесь операции обозначены соответствующими знаками (при этом присвоение также является операцией). Идентификаторы *T*1, …, *T*4 обозначают временные переменные, используемые для хранения результатов вычисления тетрад. В последней тетраде (присвоение) требуется только один операнд, поэтому в качестве второго операнда выступает незначащий операнд «0».

8.7. Триады – многоадресный код с неявно именуемым результатом

Триады представляют собой запись вида <операция> (<операнд1>, <операнд2>).

Для того чтобы избежать вставки временных имен в таблицу символов, можно ссылаться на временные значения по номеру инструкции, которая вычисляет значение, соответствующее этому имени. Тогда трехадресные инструкции можно представить записями только с тремя полями: *op*, *arg*1 и *arg*2 (табл. 20). Поля *arg*1 и *arg*2 для аргументов *op* представляют собой либо указатели в таблице символов (для определенных программистом имен или констант), либо указатели на тройки (для временных значений).

В табл. 19 числа в скобках представляют собой указатели на тройки; указатели на таблицу символов представлены соответствующими именами. На практике информация, необходимая для интерпретации различных типов записей в полях *arg*1 и *arg*2, может быть закодирована в поле *op* или в дополнительных полях. Триады в табл. 20 соответствуют тетрадам в табл. 19.

Таблица 20

Представление трехадресных инструкций триадами

	op	*Arg*l	*arg*2
(0)	uminus	*c*	
(1)	*	*b*	(0)
(2)	uminus	*c*	
(3)	*	*b*	(2)
(4)	+	(1)	(3)
(5)	assign	*a*	(4)

Инструкция копирования $a := t_5$ закодирована триадой, в которой *a* размещено в поле *arg*1 и использован оператор *assign*.

Операции типа *x*[*i*] := *y* требуют две триады (табл. 21), операции типа *x*:= *y*[*i*] – две триады (табл. 22).

Таблица 21

	op	*arg*l	*arg*2
(0)	[] =	*x*	*i*
(1)	assign	(0)	*y*

Таблица 22

	op	arg1	arg2
(0)	= []	*y*	*i*
(1)	assign	*x*	(0)

Особенность триад состоит в том, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому при записи последовательно триады нумеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера триад, а непосредственно ссылки в виде указателей – тогда при изменении нумерации и порядка следования триад ссылок не требуется).

Триады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме триад, они вычисляются последовательно одна за другой. Каждая триада в последовательности вычисляется так: операция, заданная триадой, выполняется над операндами. Если в качестве одного из операндов

(или обоих операндов) выступает ссылка на другую триаду, то берется результат вычисления той триады. Результат вычисления триады нужно сохранять во временной памяти, так как он может потребоваться последующим триадам. Если какой-то из операндов в триаде отсутствует, то он может быть опущен или заменен пустым операндом. Порядок вычисления триад, как и для тетрад, может быть изменен, но только если допустить наличие триад, целенаправленно изменяющих этот порядок.

Преимущество триад:

1. Триады представляют собой линейную последовательность. Поэтому для них несложно написать алгоритм, который будет преобразовывать последовательность триад в последовательность команд результирующей программы (либо последовательность команд ассемблера). Однако здесь требуется также и алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов вычисления, так как временные переменные для этой цели не используются. В этом отличие триад от тетрад;

2. Триады представляют собой машинно-независимую форму внутреннего представления программы, так как не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа (в этом они похожи на тетрады);

3. Триадам требуется меньше памяти для своего представления, чем тетрадам;

4. Триады явно отражают взаимосвязь операций между собой, что делает их применение удобным.

Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как результаты удобно распределять не только по доступным ячейкам временной памяти, но и по имеющимся регистрам процессора. Это дает определенные преимущества. Триады ближе к двухадресным машинным командам, чем тетрады, именно эти команды более всего распространены в наборах команд большинства современных компьютеров.

Пример. Выражение *A*: = *B* **C* + *D* – *B* *10, записанное в виде триад, будет иметь вид:

1) * (*B*, *C*)
2) + (^1, *D*)
3) * (*B*, 10)
4) – (^2, ^3)
5) := (*A*, ^4)

Здесь операции обозначены соответствующим знаком (при этом присвоение также является операцией), а знак «^» означает ссылку операнда одной триады на результат другой.

8.8. Косвенные триады

Еще одно представление трехадресного кода состоит в использовании списка указателей на триады вместо списка самих триад. Такая реализация называется *косвенными тройками* (*косвенными триадами*). В качестве примера воспользуемся массивом *statement* для перечисления указателей на тройки в требуемом порядке. Тройки в табл. 20 могут быть представлены, как в табл. 23, 24.

Таблица 23

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Таблица 24

	op	*arg*1	*arg*2
(0)	uminus	*c*	
(1)	*	*b*	(14)
(2)	uminus	*c*	
(3)	*	*b*	(16)
(4)	+	(15)	(17)
(5)	assign	*a*	(18)

8.9. Сравнение представлений: использование косвенного обращения

Разницу между триадами и тетрадами можно рассматривать как вопрос о наличии в представлении определенной степени косвен-

ности. Когда окончательно генерируется целевой код, каждое имя - временное или определенное программистом - получает некоторый адрес в памяти. Этот адрес помещается в запись таблицы символов для данного имени. При использовании тетрад трехадресные инструкции, определяющие или использующие временные переменные, могут непосредственно обращаться к памяти для этих переменных с помощью таблицы символов.

Более важное преимущество тетрад проявляется в оптимизирующем компиляторе, где инструкции зачастую перемещаются. При использовании тетрад таблица символов добавляет дополнительную степень косвенности между вычислением значения и его использованием. При перемещении инструкций, вычисляющих *x*, инструкция, использующая это значение, не требует внесения никаких изменений. В случае же использования триад перемещения инструкции, определяющие временное значение, требуют изменения всех ссылок на эту инструкцию в массивах *arg*1 и *arg*2. Эта проблема затрудняет использование триад в оптимизирующих компиляторах.

При использовании косвенных триад такой проблемы не возникает – перемещение инструкций осуществляется простым переупорядочением списка *statement*. Поскольку указатели на временные значения ссылаются на массив значений *op–arg*1–*arg*2, который остается неизменным, ни один из этих указателей не нуждается в изменениях при перемещении триад. Таким образом, косвенные триады очень похожи на тетрады с точки зрения их использования. Обе системы записи требуют примерно одинакового количества памяти, они одинаково эффективны при переупорядочении кода. Как и в случае с обычными триадами, выделение памяти для временных переменных должно быть отложено до фазы генерации целевого кода. Однако косвенные триады могут сэкономить определенный объем памяти по сравнению с тетрадами, если некоторое временное значение используется несколько раз. Причина этого заключается в том, что две или более записи таблицы *statement* могут указывать на одну и ту же строку в структуре *op*-*arg*1-*arg*2.

8.10. Ассемблерный код и машинные команды

Машинные команды используют внутреннее представление программы полностью, соответствующее объектному коду, и сложных преобразований не требуют. Команды ассемблера представляют собой форму записи машинных команд, поэтому формы внутреннего представления программы практически ничем не отличаются от них.

Однако использование команд ассемблера или машинных команд для внутреннего представления программы требует дополнительных структур для отображения взаимосвязи операций. В этом случае внутреннее представление программы получается зависимым от архитектуры вычислительной системы, на которую ориентирован результирующий код. Значит, при ориентировании компилятора на другой результирующий код необходимо перестраивать и внутреннее представление программы и методы его обработки (при использовании триад или тетрад этого не требуется).

Тем не менее машинные команды – это язык, на котором должна быть записана результирующая программа. Поэтому компилятор так или иначе должен работать с ними. Кроме того, только обрабатывая машинные команды (или их представление в форме команд ассемблера), можно добиться наиболее эффективной результирующей программы. Любой компилятор работает с представлением результирующей программы в форме машинных команд, однако их обработка происходит, как правило, на завершающих этапах фазы генерации кода.

8.11. Обратная польская запись операций

Обратная польская запись – это удобная для вычисления выражений форма записи операций и операндов. Эта форма предусматривает, что знаки операций записываются после операндов.

Обратная польская запись – это постфиксная запись операций. В этой записи знаки операций записываются непосредственно за операндами. По сравнению с обычной записью операций в поль-

ской записи операнды следуют в том же порядке, а знаки операций – строго в порядке их выполнения. Так как в этой форме записи все операции выполняются в том порядке, в котором они записаны, это делает ее чрезвычайно удобной для вычисления выражения на компьютере. Польская запись не требует учета приоритета операций, в ней употребляются скобки, и в этом ее основное преимущество.

Постфиксная запись выражения E может быть определена следующим образом.

1. Если E является переменной или константой, то постфиксная запись E представляет собой E.

2. Если E – выражение вида $E_1\ op\ E_2$, где op – произвольный бинарный оператор, то постфиксная запись E представляет собой $E'_1\ E'_2\ op$, где E'_1 и E'_2 – постфиксные записи для E_1 и E_2 соответственно.

3. Если E – выражение вида E_1, то постфиксная запись для E_1 представляет собой также и постфиксную запись для E.

4. Скобки в постфиксной записи не используются.

Например, постфиксная запись для выражения (8 – 3) + 5 имеет вид 83 – 5 +.

Она особенно эффективна в тех случаях, когда для вычислений используется стек. Главный *недостаток* обратной польской записи заключается в следующем: поскольку используется стек, то для работы с ним всегда доступна только верхушка стека, а это делает крайне затруднительной оптимизацию выражений в форме обратной польской записи. Практически выражения в форме обратной польской записи почти не поддаются оптимизации.

Там, где оптимизации вычисления выражений не требуется или она не имеет большого значения, обратная польская запись оказывается очень удобным методом внутреннего представления программы.

Обратная польская запись была предложена первоначально для записи арифметических выражений. Однако этим ее применение не ограничивается. В компиляторе можно порождать код в форме обратной польской записи для вычисления практически любых выражений. Для этого достаточно ввести знаки, предусматривающие

вычисление соответствующих операций. В том числе можно ввести операции условного и безусловно перехода, предполагающие изменение последовательности хода вычислений и перемещение вперед или назад на некоторое количество шагов в зависимости от результата на верхушке стека. Такой подход позволяет очень широко применять форму обратной польской записи.

Преимущества и недостатки обратной польской записи определяют и сферу ее применения. Так, она очень широко используется для вычисления выражений в интерпретаторах и командных процессорах, где оптимизация вычислений либо отсутствует вовсе, либо не имеет существенного значения.

8.11.1. Вычисление выражений с помощью обратной польской записи

Вычисление выражений в обратной польской записи выполняется достаточно просто с помощью стека. Для этого выражение просматривается в порядке слева направо, и встречающиеся в нем элементы обрабатываются по следующим правилам:

1. Если встречается операнд, то он помещается в стек (попадает в верхушку стека).
2. Если встречается знак унарной операции (операции, требующей одного операнда), то операнд выбирается с верхушки стека, операция выполняется и результат помещается в стек (попадает в верхушку стека).
3. Если встречается знак бинарной операции (операции, требующей двух операндов), то два операнда выбираются с верхушки стека, операция выполняется, и результат помещается в стек (попадает в верхушку стека).

Вычисление выражения заканчивается, когда достигается конец записи выражения. Результат вычисления при этом всегда находится на верхушке стека. Очевидно, что данный алгоритм можно легко расширить и для более сложных операций, требующих три и более операнда [1].

На рис. 28 представлены примеры вычисления выражений в обратной польской записи.

Рис. 28. Вычисление выражений в обратной польской записи с использованием стека

Вопросы

1. Какие задачи в компиляторе решает семантический анализ?
2. Сравните между собой основные способы внутреннего представления программ.
3. На каких этапах компиляции лучше всего использовать каждый из этих способов?
4. Какие способы (или способ) внутреннего представления программы обязательно должен уметь обрабатывать компилятор?
5. Почему линейные участки легче всего поддаются оптимизации, чем все другие части внутреннего представления программы и объектного кода?

9. СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ТРАНСЛЯЦИЯ

Для трансляции конструкций языка программирования компилятору, помимо генерации кода, может потребоваться отследить множество различных параметров. Например, компилятору может

понадобиться информация о типе конструкции, о расположении первой инструкции в целевом коде или количестве сгенерированных инструкций. Таким образом, можно говорить о некоторых абстрактных атрибутах, связанных с языковыми конструкциями: типах, строках, адресах памяти.

Значения атрибутов вычисляются согласно семантическим правилам, связанным с продукциями грамматики.

Существует два вида записи для связанных с продукциями семантических правил:

1) *синтаксически управляемые определения* – представляют собой высокоуровневые спецификации трансляции, скрывающие множество деталей реализации и освобождающие пользователя от явного указания порядка выполнения трансляции;

2) *схемы трансляции* – указывают порядок, в котором выполняются семантические правила; так что эти схемы показывают определенную часть деталей реализации.

Концептуально при обоих методах разбирается входной поток лексем, строится дерево разбора и обходится так, как необходимо для выполнения семантических правил в узлах дерева разбора (рис. 29). Выполнение семантических правил может генерировать код, сохранять информацию в таблице символов, выводить сообщения об ошибках или выполнять какие-либо другие действия. Результат трансляции потока лексем будет получен путем выполнения указанных семантических правил.

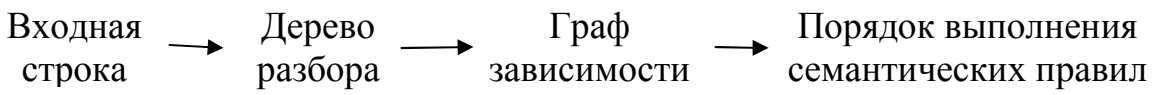


Рис. 29. Концепция синтаксически управляемой трансляции

Реализация не всегда следует приведенной на рис. 29 схеме. Частные случаи синтаксически управляемых определений могут быть реализованы за один проход выполнением семантических правил в процессе синтаксического анализа, без явного построения дерева разбора или графа, показывающего взаимосвязи между атрибутами.

Например, подкласс СУ-определений, именуемый *L*-атрибутными определениями, включает практически все этапы трансляции, которые могут выполняться без явного построения дерева разбора.

9.1. Синтаксически управляемые определения

Синтаксически управляемое определение представляет собой обобщение контекстно-свободной грамматики, в которой каждый грамматический символ имеет связанное множество атрибутов – синтезируемые и наследуемые атрибуты. Если рассматривать узел грамматического символа в дереве разбора как запись с полями для хранения информации, то атрибут соответствует имени поля.

Атрибут может представлять собой все, что угодно: строку, число, тип, адрес памяти и т.д. Значение атрибута в узле дерева разбора определяется семантическими правилами, связанными с используемой в данном узле продукцией.

Значение синтезируемого атрибута в узле вычисляется по значениям атрибутов в дочерних по отношению к данному узлах.

Значения наследуемых атрибутов определяются значениями атрибутов соседних (т.е. узлов, дочерних по отношению к родительскому узлу данного) и родительского узлов.

Семантические правила определяют зависимости между атрибутами, которые представляются графом. Граф зависимости определяет порядок выполнения семантических правил, что, в свою очередь, дает значения атрибутов в узлах дерева разбора входной строки. Семантические правила могут иметь и побочные действия, например вывод значения или изменение глобальной переменной. Естественно, при реализации не обязательно явным образом строить дерево разбора или граф зависимостей; главное, чтобы для каждой входной строки выполнялись верные действия в правильном порядке.

Дерево разбора, показывающее значения атрибутов в каждом узле, называется *аннотированным*, а процесс вычисления значений атрибутов в узлах дерева – *аннотированием* дерева разбора.

9.2. Вид синтаксически управляемого определения

В синтаксически управляемом определении каждая продукция грамматики $A \rightarrow \alpha$ имеет связанное с ней множество *семантических правил* вида

$$b := f(c_1, c_2, \ldots, c_k),$$

где f – функция: $c_1, c_2, \ldots, c_k$ – атрибуты грамматических символов продукции; b – синтезируемый атрибут символа A или наследуемый атрибут одного из грамматических символов правой части продукции.

В любом случае атрибут b *зависит* от атрибутов $c_1, c_2, \ldots, c_k$.

Атрибутная грамматика является синтаксически управляемым определением, в котором функции в семантических правилах не имеют побочных эффектов.

Функции в семантических правилах зачастую записываются как выражения. Иногда единственная цель семантического правила в синтаксически управляемом определении состоит именно в создании побочного эффекта. Такие семантические правила записываются как вызовы процедур или фрагменты программ. Их можно рассматривать как правила, определяющие значения фиктивных синтезируемых атрибутов нетерминала в левой части связанной продукции; фиктивный атрибут и знак присвоения «: =» при этом не указываются.

Пример 30. В табл. 25 приведено синтаксически управляемое определение программы настольного калькулятора. Это определение связывает с каждым из нетерминалов *E*, *T* и *F* целочисленный синтезируемый атрибут *val*. Для каждой *E*-, *T*- и *F*-продукции семантическое правило вычисляет значение атрибута *val* нетерминала из левой части продукции по значениям атрибутов нетерминалов правой части.

Таблица 25

Синтаксически управляемое определение простого калькулятора

Продукция	Семантические правила
$L \to E\ n$	*print(E.val)*
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val \times F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to$ digit	$F.val :=$ digit.*lexval*

Лексема digit имеет синтезируемый атрибут *lexval*, значение которого предоставляется лексическим анализатором. Правило, связанное с продукцией $L \to En$ для стартового нетерминала L, представляет собой процедуру вывода значения арифметического выражения, порождаемого E.

В синтаксически управляемом определении предполагается, что терминалы могут иметь только синтезируемые атрибуты, поскольку определение не дает никаких семантических правил для терминалов (обычно значения атрибутов терминалов предоставляются лексическим анализатором). Кроме того, если не оговорено особо, стартовый символ не имеет наследуемых атрибутов.

9.3. Синтезируемые атрибуты

Синтезируемые атрибуты часто используются на практике.

S-атрибутным определением называется синтаксически управляемое определение, использующее только синтезируемые атрибуты.

Дерево разбора для *S*-атрибутного определения всегда может быть аннотировано путем выполнения семантических правил для атрибутов в каждом узле снизу вверх, от листьев к корню.

Пример 31. *S*-атрибутное определение в примере 30 описывает калькулятор, считывающий арифметическое выражение из цифр, скобок, операторов + и *, за которым следует символ новой строки **n**, и выводит значение выражения. Например, получив выражение 3 * 5 + 4, за которым следует символ новой строки, программа выводит значение 19. На рис. 30 показано

аннотированное дерево разбора для входной строки 3 * 5 + 4**n**. Выход программы, печатаемый в корне дерева, представляет собой значение *E.val* в первом дочернем узле корня дерева.

Вычисление значений атрибутов производится следующим образом. Крайнему слева снизу внутреннему узлу соответствует использование продукции $F \rightarrow$ digit. Семантическое правило *F.val: = = *digit.*lexval* определяет атрибут *F.val* в этом узле, имеющий значение 3, поскольку значение digit.*lexval* в дочернем узле равно 3. Аналогично в узле, родительском по отношению к данному узлу, атрибут *T.val* также имеет значение 3.

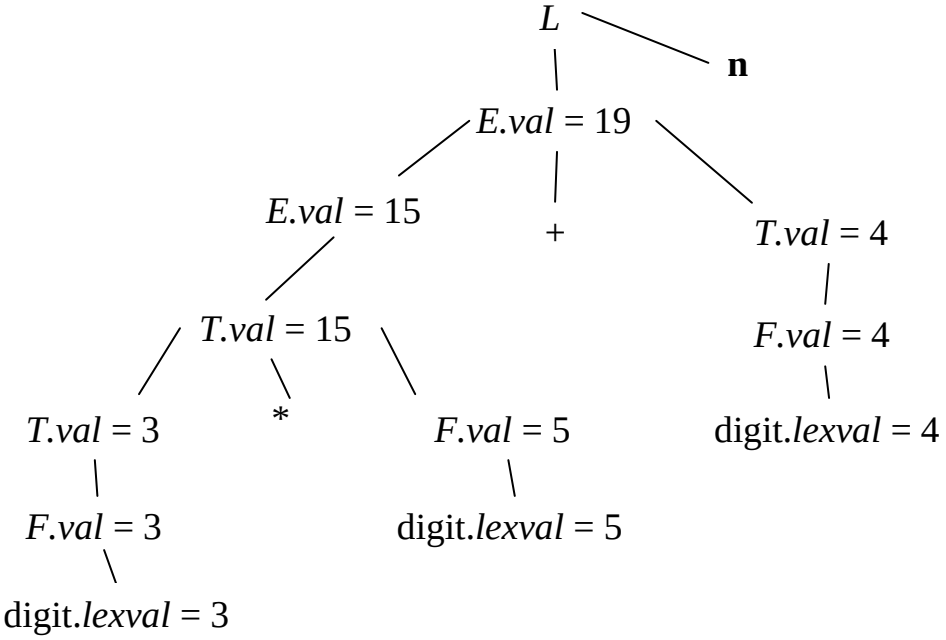


Рис. 30. Аннотированное дерево разбора для 3 *5 + 4*n*

Теперь рассмотрим узел продукции $T \rightarrow T * F$. Значение атрибута *T.val* в этом узле определяется следующим образом:

Продукция	Семантическое правило
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

При использовании этого семантического правила в данном узле T_1.*val* имеет значение 3, полученное от левого наследника, а *F.val* – значение 5, полученное от правого наследника. Следовательно, в этом узле *T.val* составляет 15.

Правило, связанное с продукцией для стартового нетерминала $L \rightarrow E$ **n**, выводит значение выражения, порожденного E.

9.4. Наследуемые атрибуты

Наследуемые атрибуты представляют собой атрибуты, значения которых в узле дерева разбора определяются атрибутами родительского и/или дочерних по отношению к родительскому узлов.

Наследуемые атрибуты удобны для выражения зависимости конструкций языка программирования от контекста, в котором они появляются. Например, наследуемые атрибуты используются для отслеживания, появляется ли идентификатор слева или справа от знака присвоения, чтобы определить, потребуется ли адрес или значение данного идентификатора. Хотя всегда можно переписать синтаксически управляемое определение таким образом, чтобы использовать только синтезируемые атрибуты, зачастую более естественно воспользоваться синтаксически управляемым определением с наследуемыми атрибутами.

В следующем примере наследуемый атрибут распространяет информацию о типе на различные идентификаторы в объявлении.

Пример 32. Рассмотрим синтаксически управляемое определение, представленное в табл. 26.

Объявление, порождаемое нетерминалом D в синтаксически управляемом определении, состоит из ключевого слова *int* или *real*, за которым следует список идентификаторов. Нетерминал T имеет синтезируемый атрибут *type*, значение которого определяется ключевым словом объявления. Семантическое правило *L.in := T.type*, связанное с продукцией $D \rightarrow T\ L$, определяет наследуемый атрибут *L.in* как тип объявления. Затем приведенные правила распространяют этот тип вниз по дереву разбора с использованием атрибута *L.in*. Правила, связанные с продукциями для L, вызывают процедуру *addtype* для добавления типа каждого идентификатора к его записи в таблице символов (определяемой атрибутом *entry*).

Таблица 26

Синтаксически управляемое определение с наследуемым атрибутом *L.in*

Продукция	Семантические правила
$D \rightarrow T\ L$	*L.in := T.type*
$T \rightarrow int$	*T.type := integer*
$T \rightarrow real$	*T.type := real*
$L \rightarrow L_1\ ,\ id$	L_1 *:= L.in* *addtype*(*id.entry, L.in*)
$L \rightarrow id$	*addtype*(*id.entry, L.in*)

На рис. 31 показано аннотированное дерево разбора для предложения *real* id_1, id_2, id_3. Значение *L.in* в трех *L*-узлах дает тип идентификаторов id_1, id_2 и id_3. Эти значения определяются вычислением значения атрибута *T.type* в левом дочернем по отношению к корню узле, а затем вычислением *L.in* в нисходящем порядке в трех *L*-узлах правого поддерева корневого узла. В каждом *L*-узле вызывается также процедура *addtype* для записи в таблицу символов информации о том, что идентификатор в правом дочернем узле имеет тип *real*.

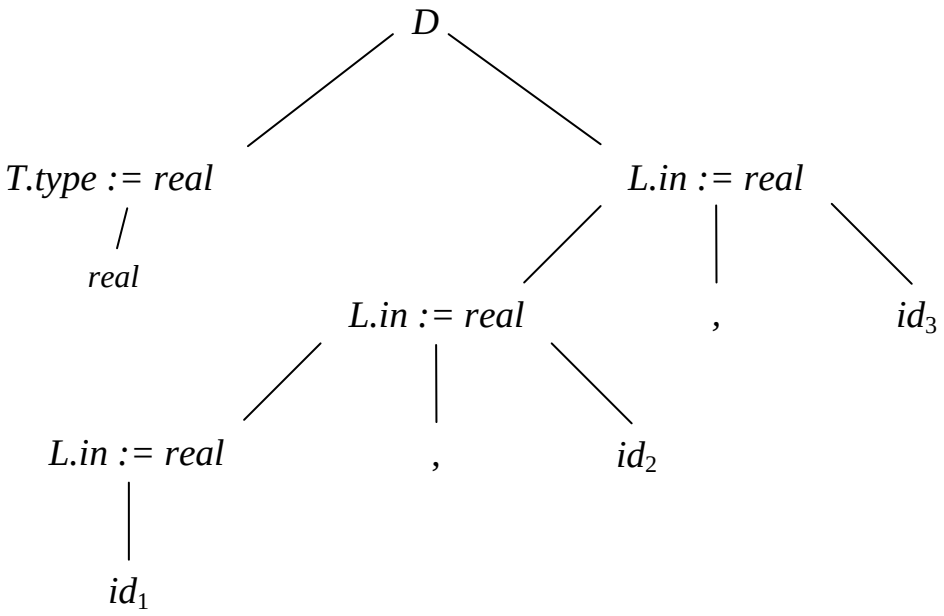


Рис. 31. Дерево разбора с наследуемыми атрибутами *in* в каждом узле, помеченном *L*

9.5. Графы зависимости

Если атрибут b в узле дерева разбора зависит от атрибута c, то семантическое правило для b в этом узле должно выполняться после семантического правила, определяющего c. Зависимости между наследуемыми и синтезируемыми атрибутами в узлах дерева разбора могут быть показаны с помощью направленного графа, называемого *графом зависимости.*

Перед построением графа зависимости для дерева разбора каждое семантическое правило приводится к виду $b := f(c_1, c_2, \ldots, c_k)$ введением фиктивного синтезируемого атрибута b для каждого семантического правила, состоящего из вызова процедуры (или представляющего собой фрагмент кода). Граф имеет узел для каждого атрибута и дугу, ведущую в узел b из узла c, если атрибут b зависит от атрибута c.

Предположим, что продукция $A \rightarrow XY$ имеет связанное с ней семантическое правило $A.a := f(X.x, Y.y)$. Это правило определяет синтезируемый атрибут $A.a$, который зависит от атрибутов $X.x$ и $Y.y$. Если эта продукция используется в дереве разбора, то в графе зависимости будет три узла $A.a$, $X.x$ и $Y.y$ с дугами от $X.x$ к $A.a$, поскольку $A.a$ зависит от $X.x$, и от $Y.y$ к $A.a$, поскольку $A.a$ зависит от $Y.y$.

Если продукция $A \rightarrow XY$ имеет семантическое правило $X.i := g(A.a, Y.y)$, связанное с ней, то в графе зависимости будут дуги от $A.a$ к $X.i$ и от $Y.y$ к $X.i$, поскольку $X.i$ зависит как от $A.a$, так и $Y.y$.

Пример 33. Когда в дереве разбора используется следующая продукция, мы добавляем показанные на рис. 32 дуги в граф зависимости.

Продукция	Семантическое правило
$E \rightarrow E_1 * E_2$	$E.val := E_1.val + E_2.val$

Три узла, помеченные в графе зависимости символом •, представляют собой синтезируемые атрибуты $E.val$, $E_1.val$ и $E_2.val$ в соответствующих узлах дерева разбора. Дуги, направленные от $E_1.val$, $E_2.val$ к $E.val$, показывают, что $E.val$ зависит от $E_1.val$, $E_2.val$. Пунктирные линии на рис. 32 представляют дерево разбора и не являются частью графа зависимости.

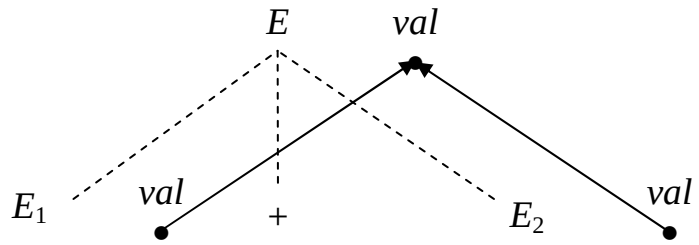


Рис. 32. Атрибут *E*.val, синтезированный из E_1.*val* и E_2.*val*

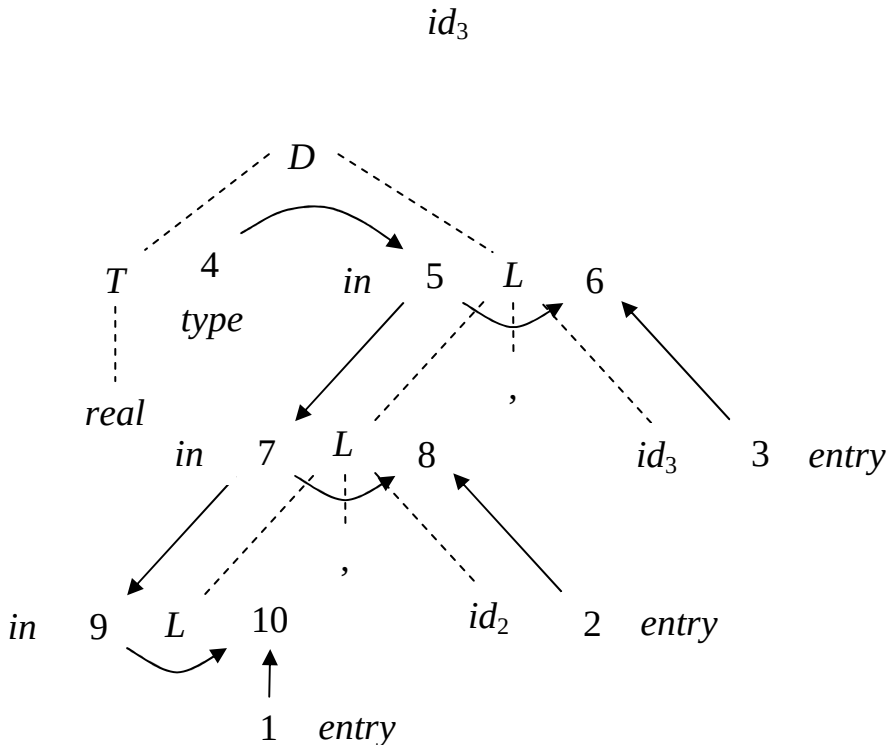


Рис. 33. Граф зависимости для дерева разбора (см. табл. 18)

Пример 34. На рис. 33 показан граф зависимости для дерева разбора на рис. 31. Узлы в графе зависимости помечены числами. Имеется дуга узла 4 для *T.type* в узел 5 *L.in*, поскольку наследуемый атрибут *L.in* зависит от атрибута *T.type* согласно семантическому правилу *L.in := T.type* для продукции $D \rightarrow TL$. Две нисходящие дуги в узлы 7 и 9 появляются в силу зависимости L_1.*in* от *L.in* в соответствии с семантическим правилом L_1.*in := L.in* для продукции $L \rightarrow L_1$, *id*. Каждое из семантических правил *addtype*(*id.entry*, *L.in*), связанных с *L*-продукциями, приводит к созданию фиктивного атрибута. Узлы 6, 8 и 10 построены для таких фиктивных атрибутов.

9.6. Порядок выполнения семантических правил

Топологическая сортировка направленного ациклического графа является упорядочением узлов графа m_1, m_2, …, m_k, таким, что дуги направлены от узлов, расположенных в упорядоченной последовательности раньше, к узлам, расположенным поэтапно, т.е. если $m_i \rightarrow m_j$ – дуга от m_i к m_j, то m_i встречается в упорядоченной последовательности раньше m_j.

Любая топологическая сортировка графа зависимости дает правильный порядок пополнения семантических правил, связанных с узлами дерева разбора, т.е. к моменту пополнения семантического правила b: = $f(c_1, c_2, \ldots, c_k)$ атрибуты $c_1, c_2, \ldots, c_k$ доступны для вычисления f.

Трансляция, определяемая синтаксически управляемым определением, может быть уточнена. Вначале для построения дерева разбора входного потока используется грамматика, лежащая в основе синтаксически управляемого определения. Затем строится граф зависимости, после топологической сортировки которого получается порядок выполнения семантических правил. Выполнение этих правил в полученном порядке приводит к трансляции входной строки.

Пример 35. Каждая из дуг графа зависимости на рис. 33 направлена от узла с меньшим номером к узлу с большим номером. Следовательно, топологическая сортировка графа зависимости может быть выполнена просто записью узлов в порядке их номеров. После такой топологической сортировки получим следующую программу (a_n означает атрибут, связанный с узлом номер n в графе зависимости).

a_4 : = *real*;
a_5 : = a_4 ;
addtype (id_3.*entry*, a_5);
a_7 : = a_5;
addtype (id_2.*entry*, a_7,);
a_9 : = a_7,
addtype (id_1.*entry*, a_9);

Выполнение этих семантических правил вносит тип *real* в записи таблицы символов для каждого из идентификаторов.

Для выполнения семантических правил существует ряд методов.

1. *Методы дерева разбора*. Порядок выполнения семантических правил определяется во время компиляции с помощью топологической сортировки графа зависимости, построенного по дереву разбора для каждой входной строки. Эти методы не позволяют определить порядок выполнения только в том случае, когда граф зависимости для конкретного дерева разбора имеет цикл.

2. *Методы, основанные на правилах*. В процессе создания компилятора анализируются семантические правила, связанные с продукциями либо вручную, либо с помощью специализированного инструментария. Порядок вычисления атрибутов, связанных с каждой продукцией, предопределяется в процессе разработки компилятора.

3. *Игнорирующие методы*. Выбор порядка выполнения происходит без рассмотрения семантических правил. Например, если трансляция происходит в процессе синтаксического анализа, то порядок выполнения задается методом синтаксического анализа независимо от семантических правил. Игнорирующие методы ограничивают класс реализуемых синтаксически управляемых определений.

Методы, основанные на правилах, и игнорирующие методы не требуют явного построения графа зависимости в процессе компиляции, поэтому они более эффективны с точки зрения времени работы и используемой памяти.

9.7. Восходящее выполнение *S*-атрибутных определений

Для определения трансляций можно использовать синтаксически управляемые определения. Рассмотрим реализацию таких трансляторов. Создание транслятора для произвольного синтаксически управляемого определения может оказаться сложной задачей; однако имеются большие классы полезных синтаксически управляемых определений, трансляторы для которых строятся дос-

таточно просто. К таким классам относятся *S-атрибутные определения*, т.е. синтаксически управляемые определения, в которых применяются исключительно синтезируемые атрибуты.

Синтезируемые атрибуты могут быть вычислены восходящим синтаксическим анализатором в процессе разбора входной строки. Синтаксический анализатор может хранить значения синтезируемых атрибутов, связанных с грамматическими символами, в своем стеке. При выполнении свертки по хранящимся в стеке атрибутам символов из правой части сворачиваемой продукции вычисляются значения новых синтезируемых атрибутов. Можно расширить стек синтаксического анализатора для хранения значений этих синтезируемых атрибутов.

9.7.1. Синтезируемые атрибуты в стеке синтаксического анализатора

Восходящий синтаксический анализатор для хранения информации о разобранных поддеревьях использует стек. Можно использовать дополнительные поля в стеке синтаксического анализатора для хранения значений синтезируемых атрибутов. Пример стека синтаксического анализатора с пространством для хранения одного значения атрибута показан в табл. 27. Стек реализован с помощью пары массивов – *state* и *val*. Каждая запись *state* является указателем (или индексом) на запись в таблице *LR*(1)-анализа. Если символ *i*-го состояния – *A*, то *val*[*i*] содержит значение атрибута, связанного с узлом дерева разбора, соответствующим этому *A*.

Таблица 27

Стек синтаксического анализатора с полем для синтезируемого атрибута

	state	*Val*
	…	…
	X	*X.x*
	Y	*Y.y*
top →	*Z*	Z.z
	…	…

Текущая вершина стека определяется указателем *top*. Считаем, что синтезируемые атрибуты вычисляются непосредственно перед каждой сверткой. Предположим, что с продукцией $A \rightarrow XYZ$ связано семантическое правило

$$A.a := f(X.x, Y.y, Z.z).$$

Перед сверткой *XYZ* в *A* значение атрибута *Z.z* находится в *val*[*top*], значение *Y.y* – в *val*[*top*-1] и *X.x* – в *val*[*top*-2]. Если символ не имеет атрибутов, то соответствующая запись в массиве *val* не определена. После свертки *top* уменьшается на 2, состояние, соответствующее *A*, помещается в *state*[*top*] (т.е. на место *X.x*), а значение синтезируемого атрибута *A.a* – в *val*[*top*].

Пример 36. Рассмотрим синтаксически управляемое определение калькулятора, приведенное на табл. 25. Синтезируемые атрибуты в аннотированном дереве разбора на рис. 30 могут быть вычислены *LR*-анализатором в процессе восходящего разбора входной строки 3 * 5 + 4*n*. Как и ранее, полагаем, что лексический анализатор обеспечивает значение атрибута digit.*lexval* – числовое значение каждой лексемы, представляющей цифру. При переносе синтаксическим анализатором лексемы digit в стек, он помещается в *state*[*top*], а значение атрибута – в *val*[*top*].

Для вычисления атрибутов модифицируем синтаксический анализатор, чтобы он выполнял фрагменты кода, показанные в табл. 28, непосредственно перед выполнением соответствующей свертки. Можно связать вычисление атрибутов со свертками, поскольку каждая свертка определяет используемую продукцию. Фрагменты кода получены из семантических правил на табл. 18 путем замены каждого атрибута позицией в массиве *val*.

Таблица 28

Реализация калькулятора с помощью *LR*-анализатора

Продукция	Фрагмент кода
$L \rightarrow E\ n$	*print*(*val*[*top*])
$E \rightarrow E_1 + T$	*val*[*ntop*]: = *val*[*top* – 2] + *val*[*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*ntop*] : = *val*[*top* – 2] * *val*[*top*] *i*
$T \rightarrow F$	
$F \rightarrow (E)$	*val*[*ntop*] : = *val*[*top* – 1]
$F \rightarrow$ digit	

Фрагменты кода не показывают, каким образом происходит работа с *ntop* и *top*. При свертке продукции с *r* символами в правой части значение *ntop* устанавливается равным *top* – *r* + 1, а после выполнения фрагмента кода *top* становится равным *ntop*.

Последовательность действий, выполняемых синтаксическим анализатором для входной строки 3 * 5 + 4*n*, показана в табл. 29, где представлено содержимое полей *state* и *val* стека после каждого действия синтаксического анализатора. Состояние стека заменяется соответствующим грамматическим символом, и вместо лексемы digit используется введенная цифра.

Таблица 29

Действия транслятора для входной строки 3 * 5 + 4*n*

Вход	*state*	*val*	Используемая продукция
3 * 5 + 4 *n*	_	_	
* 5 + 4 *n*	3	3	
* 5 + 4 *n*	*F*	3	$F \rightarrow digit$
* 5 + 4 *n*	*T*	3	$T \rightarrow F$
* 5 + 4 *n*	*T* *	3 _	
+ 4 *n*	*T* * 5	3 _	$F \rightarrow digit$
+ 4 *n*	*T* * *F*	3 _ 5	
+ 4 *n*	*T*	15	$T \rightarrow T * F$
+ 4 *n*	*E*	15	$E \rightarrow T$
+ 4 *n*	*E* +	15 _	
n	*E* + 4	15 _ 4	
n	*E* + *F*	15 _ 4	$F \rightarrow digit$
	E + *T*	15 _ 4	$T \rightarrow F$
n	*E*	19	$E \rightarrow E + T$
	E n	19 _	
	L	19	$L \rightarrow E\ n$

Рассмотрим последовательность событий для входного символа 3. На первом шаге синтаксический анализатор переносит в стек состояние, соответствующее лексеме digit (значение ее атрибута равно 3). Состояние представлено тройкой; в поле *val* находится значение 3. На следующем шаге синтаксический анализатор выполняет свертку в соответствии с продукцией $F \rightarrow$ digit и семантическим правилом *F.val* := digit.*lexval*. Третий шаг состоит в свертке по продукции $T \rightarrow F$. С этой продукцией не связан никакой фрагмент кода, поэтому массив *val* остается неизменным. После каждой свертки вершина стека *val* содержит значение атрибута, связанное с левой частью сворачивающей продукции. При реализации фрагменты кода выполняются непосредственно перед сверткой. Таким образом, можно связать с продукцией действие, выполняемое в процессе свертки в соответствии с данной продукцией.

9.8. *L*-атрибутные определения

Другой класс синтаксически управляемых определений – *L*-атрибутные определения.

Синтаксически управляемое определение является *L-атрибутным*, если каждый наследуемый атрибут символа X_j, $1 \le j \le n$, из правой части продукции $A \rightarrow X_1 X_2 \ldots X_n$ зависит только от:

1) атрибутов символов $X_1, X_2, \ldots, X_{j-1}$, расположенных в продукции слева от X_{j-1};

2) наследуемых атрибутов A.

Например, каждое *S*-атрибутное определение является *L*-атрибутным, так как ограничения (1) и (2) относятся только к наследуемым атрибутам.

Пример 37. Синтаксически управляемое определение (табл. 30) не является *L*-атрибутным, поскольку наследуемый атрибут *Q.i* грамматического символа *Q* зависит от атрибута *R.s* грамматического символа справа.

Таблица 30

Синтаксически управляемое определение, не являющееся *L*-атрибутным

Продукция	Семантические правила
$A \to L\ M$	$L.i := l(A.i)$
	$M.i := m(L.s)$
	$A.s := f(M.s)$
$A \to Q\ R$	$R.i := r(A.i)$
	$Q.i := q(R.s)$
	$A.s := f(Q.s)$

9.9. Схемы трансляции

Другой способ записи семантических правил – это схемы трансляции. Для определения трансляции используем процедурную спецификацию.

Схема трансляции – это контекстно-свободная грамматика, в которой атрибуты, связанные с символами грамматики, и семантические действия заключены в фигурные скобки ({ }) и вставлены в правые части продукций. Схемы трансляции являются удобным способом записи определения трансляции, выполняемой в процессе синтаксического анализа.

Схемы трансляции имеют как наследуемые, так и синтезируемые атрибуты. Схема трансляции подобна СУ-определению, однако здесь явно указан порядок применения семантических правил. Выполнение семантических правил указывается с помощью фигурных скобок в правой части продукции.

Пример 38. Схема трансляции, преобразующая инфиксные выражения со сложением и вычитанием в соответствующие постфиксные выражения:

$E \to TR$
$R \to addop\ T\ \{print\ (addop.lexeme)\}\ R_1 \mid \lambda$
$T \to num\{print(num.val)\}$

На рис. 34 показано дерево разбора для входной строки 9 – 5 + 2, на котором семантические действия показаны как дочерние узлы по отношению к узлам, представляющим левые части соответствующих продукций. По сути, действия рассматриваются как терминальные символы, что удобно для определения момента выполнения этих действий. Вместо лексем *num* и *addop* указываются реальные числа и операция сложения. При выполнении действий в порядке обхода дерева вывода в глубину действия (рис. 34) приводят к выводу 95 – 2 +.

При разработке схемы трансляции необходимо соблюдать некоторые ограничения для того, чтобы гарантировать, что значение атрибута доступно при обращении к нему. Эти ограничения, накладываемые *L*-атрибутными определениями, гарантируют, что действие не использует атрибут, который еще не вычислен.

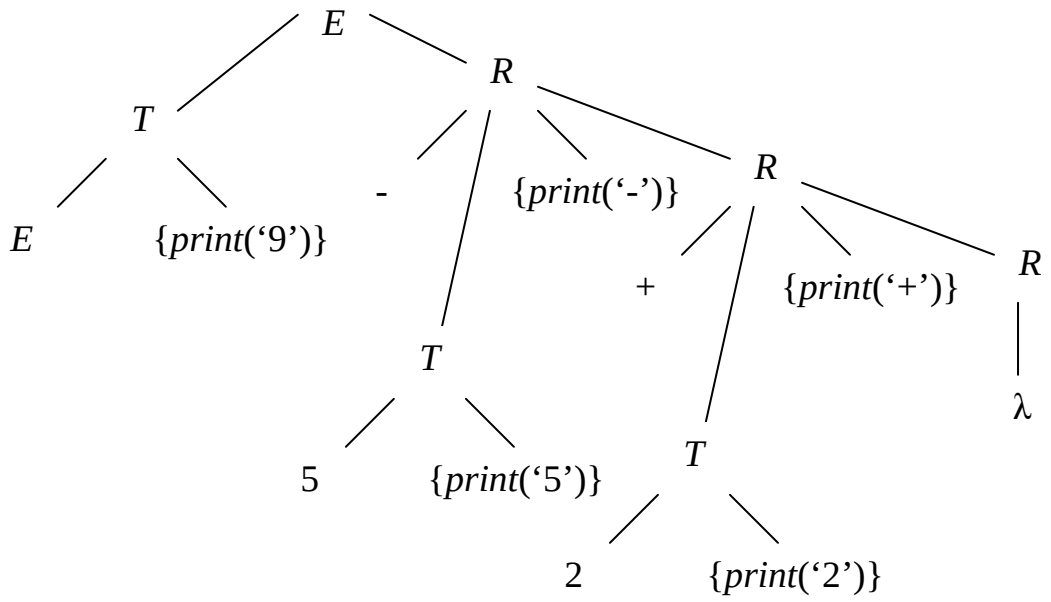


Рис. 34. Дерево разбора с семантическими действиями для выражения 9 – 5 + 2

Простейший случай – когда используются только синтезируемые атрибуты. В этом случае можно построить схему трансляции просто путем создания действий для каждого семантического правила, состоящих из присвоения, и размещения этих действий в конце правой части связанных продукций.

Пример. Продукция $T \rightarrow T_1 * F$ и семантическое правило $T_1.val1 := T_1.val \times F.val$ дают следующую продукцию и семантическое действие:

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

Если имеются и синтезируемые, и наследуемые атрибуты, необходимо соблюдать следующие правила.

1. Наследуемый атрибут для символа из правой части продукции должен вычисляться в действии перед этим символом.

2. Действие не должно обращаться к синтезируемому атрибуту символа справа от действия.

3. Синтезируемый атрибут для нетерминала в левой части продукции может вычисляться только после того, как будут вычислены все атрибуты, от которых он зависит. Действие, вычисляющее такой атрибут, обычно может размещаться в конце правой части продукции.

Пример. Следующая схема трансляции не удовлетворяет первому из трех требований:

$S \rightarrow A_1 A_2 \ \{A_1.in := 1; A_2.in := 2\}$
$A \rightarrow a \ \{print(A.\ in)\}$

При попытке вывести входную строку *aa* в процессе обхода в глубину дерева разбора наследуемый атрибут $A.in$ во второй продукции не определен. Обход в глубину начинается в S и проходит поддеревья A_1 и A_2 до того, как устанавливаются значения $A_1.in$ и $A_2.in$. Если действие, определяющее значения $A_1.in$ и $A_2.in$, вставить перед символами A в правой части продукции $S \rightarrow A_1\ A_2$, то $A.in$ будет определено при каждом вызове $print(A.\ in)$.

Начав с L-атрибутного синтаксически управляемого определения, всегда можно построить схему трансляции, удовлетворяющую трем приведенным выше требованиям.

9.9.1. Восходящее вычисление наследуемых атрибутов. Удаление внедренных действий из схемы трансляции

Все действия при восходящей трансляции должны находится в правом конце продукции. Чтобы понять, каким образом наследуе-

мые атрибуты могут обрабатываться снизу вверх, необходимо рассмотреть преобразование, по которому все вставленные действия в схеме трансляции располагаются в правых концах их продукций.

Это преобразование вставляет в базовую грамматику новые нетерминалы-маркеры, порождающие λ, т.е. каждое вставленное действие заменяется отдельным маркером M и добавляется действие в конец продукции $M \to \lambda$.

П р и м е р . Схема трансляции

$E \to T\ R$
$R \to +\ T\ \{print('+')\}\ R \mid -\ T\ \{print('-')\}\ R \mid \lambda$
$T \to num\ \{\ print(num.val)\ \}$

преобразуется с помощью нетерминалов M и N в

$E \to T\ R$
$R \to +\ T\ M\ R \mid -\ T\ N\ R \mid \lambda$
$T \to num\ \{print(num.val)\}$
$M \to \lambda\ \{print('+')\}$
$N \to \lambda\ \{print('-')\}$

Грамматики в этих двух схемах трансляции задают один и тот же язык, и, построив дерево разбора с дополнительными узлами для действий, можно показать, что действия выполняются в одном и том же порядке. Действия в преобразованной схеме трансляции завершают продукции, так что они могут быть выполнены непосредственно перед сверткой правой части в процессе восходящего разбора.

9.9.2. Наследование атрибутов в стеке синтаксического анализатора

Восходящий синтаксический анализатор сворачивает правую часть продукции $A \to XY$ удалением X и Y с вершины стека синтаксического анализатора и заменой их на A. Предположим, что X имеет синтезируемый атрибут $X.s$ (который хранится в стеке вместе с X).

Поскольку значение *X.s* находится в стеке перед выполнением любой свертки в поддереве ниже *Y*, это значение может быть наследовано *Y*. Таким образом, если наследуемый атрибут *Y.i* определен правилом копирования *Y.i*: = *X.s*, то значение *X.s* может быть использовано там, где требуется *Y.i*. Правила копирования играют важную роль в вычислении наследуемых атрибутов в процессе восходящего синтаксического анализа.

Пример 39. Тип идентификатора может быть передан с помощью правил копирования и наследуемых атрибутов (рис. 35). Рассмотрим действия восходящего синтаксического анализатора при входной строке *real p, q, r*.

Значение атрибута *T.type* можно получить при использовании продукции для *L*. Схема трансляции имеет следующий вид:

$D \rightarrow T$ $\quad L$	$\{L.in := T.type\}$
$T \rightarrow int$	$\{T.type := integer\}$
$T \rightarrow real$	$\{T.type := real\}$
$L \rightarrow$ $\quad L_1, id$	$\{L_1.in := L.in\}$ $\{addtype\ (id.entty, L.in)\}$
$L \rightarrow id$	$\{addtype\ (id.entry, L.in)\}$

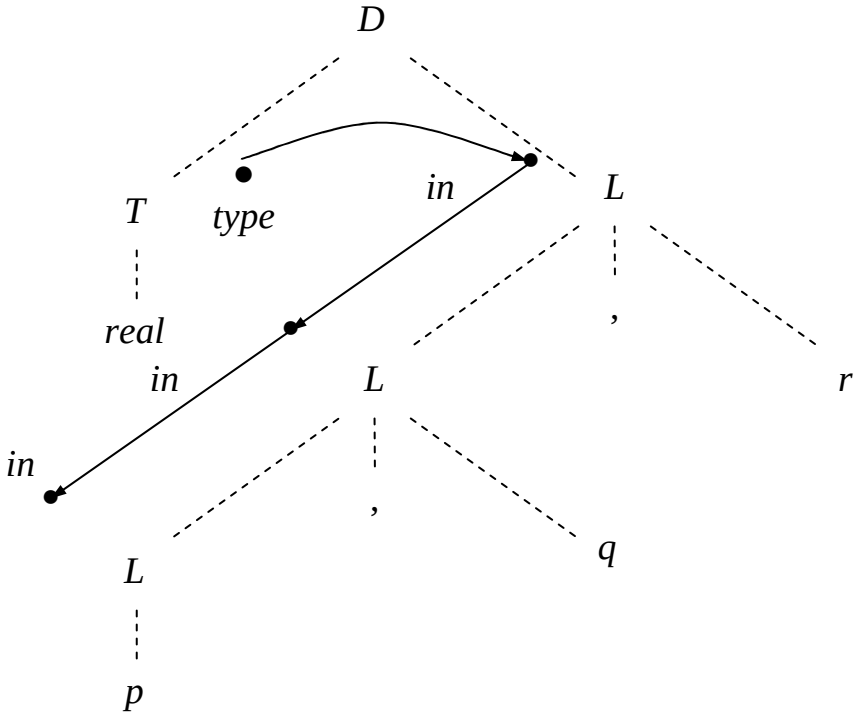


Рис. 35. *L.in* = *T*.type в каждом узле для *L*

Если проигнорировать действия в приведенной выше схеме трансляции, то последовательность шагов синтаксического анализатора при входном потоке *real p, q, r* будет иметь вид, представленный табл. 31.

Для ясности вместо состояния стека здесь показан соответствующий символ грамматики, а вместо лексем *id* – соответствующие им идентификаторы.

Предположим, что стек синтаксического анализатора реализован в виде пары массивов – *state* и *val*. Если *state*[*i*] – грамматический символ *X*, то в *val*[*i*] хранится синтезируемый атрибут *X.s*.

Таблица 31

При свертке правой части продукции для *L*, *T* находится непосредственно под правой частью

Входной Поток	Состояние стека	Использованная продукция
real p, q, r	–	
p, q, r	*real*	
p, q, r	*T*	$T \rightarrow real$
, q, r	*T p*	
, q, r	*T L*	$L \rightarrow id$
q, r	*T L,*	
, r	*T L, q*	
, r	*T L*	$L \rightarrow L, id$
r	*T L,*	
	T L, r	
	T L	$L \rightarrow L, id$
	D	$D \rightarrow T L$

Содержимое массива *state* показано в табл. 31. Всякий раз, когда сворачивается правая часть продукции *L*, в стеке непосредственно под правой частью располагается *T*. Этот факт можно использовать для доступа к значению атрибута *T.type*.

Другая реализация (табл. 32) использует информацию о местоположении *T.type* в стеке *val* относительно его вершины. Пусть *top* и *ntop* указывают верхний элемент стека непосредственно перед сверткой и после нее.

Таблица 32

Значение *T.type*, используемое вместо *L.in*

Продукция	Фрагмент кода
$D \rightarrow TL;$	
$T \rightarrow int$	$val[ntop] := integer$
$T \rightarrow real$	$val[ntop] := real$
$L \rightarrow L, id$	$addtype(val[top], val[top-3])$
$L \rightarrow id$	$addtype(val[top], val[top-1])$

Из правил копирования, определяющих *L.in*, получаем, что вместо *L.in* можно использовать *T.type*.

При применении продукции $L \rightarrow id$ на вершине стека *val* находится *id.entry*, а непосредственно под ним – *T.type*. Следовательно, $addtype(val[top], val[top-1])$ эквивалентно *addtype*(*id.entry, T.type*). Поскольку правая часть продукции $L \rightarrow L, id$ имеет три символа, *T.type* при ее свертке находится в $val[top-3]$. Таким образом, правила копирования для атрибута *L.in* устраняются, поскольку вместо него используется значение *T.type* из стека.

9.9.3. Замена наследуемых атрибутов синтезируемыми

Иногда можно избежать использования наследуемых атрибутов путем изменения грамматики.

Например, объявление в PASCAL может состоять из списка идентификаторов, за которым следует тип: *m*, *n*: integer;

Грамматика такого объявления может включать в себя продукции вида:

$D \rightarrow L{:}T$
$T \rightarrow integer \mid char$
$L \rightarrow L, id \mid id$

Поскольку идентификаторы порождаются *L*, но тип в поддереве *L* не содержится, нельзя связать тип с идентификатором с помощью только синтезируемых атрибутов. Если нетерминал *L* наследует тип

от *T*, то получается *СУ*-определение не являющееся *L*-атрибутным, так что трансляция на его основе не может быть выполнена в процессе синтаксического анализа.

Решение этой проблемы состоит в реструктуризации грамматики для включения типа в качестве последнего элемента в список идентификаторов.

$D \rightarrow id\ L$
$L \rightarrow id\ L\ /\ :\ T$
$T \rightarrow integer\ /\ char$

Теперь тип может рассматриваться как синтезируемый атрибут *L.type*, и в таблицу символов можно внести тип каждого идентификатора, порождаемого *L*.

9.9.4. Память для значений атрибутов во время компиляции

При заданном порядке вычисления атрибутов *время жизни* атрибута начинается, когда атрибут впервые вычисляется, и заканчивается, когда вычислены все атрибуты, зависящие от него. Можно сэкономить память, сохраняя значения атрибутов только на протяжении их времени жизни.

Рассмотрим не являющееся *L*-атрибутным синтаксически управляемое определение, предназначенное для передачи информации о типе идентификаторам в объявлении.

Пример 40. Синтаксически управляемое определение (табл. 33) представляет собой объявления вида

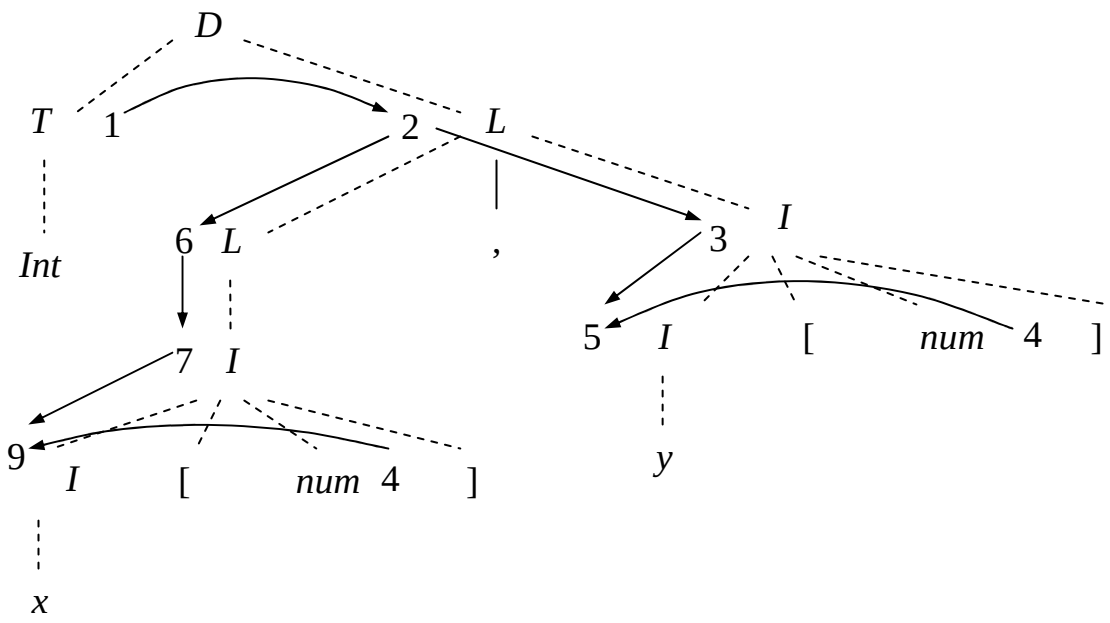
real c[12] [31];
int x[3], *y*[5]. (11)

Дерево разбора для (11) показано на рис. 36, *а* пунктирными линиями. Тип, полученный из *T*, наследуется *L* и передается вниз идентификаторам в объявлении. Дуга от *T.type* к *L.in* показывает, что *L.in* зависит от *T.type*. Синтаксически управляемое определение в табл. 32 не является *L*-атрибутным, поскольку I_1.*in* зависит от *num.val*, а *num* располагается справа от I_1, в $I \rightarrow I_1$, [*num*].

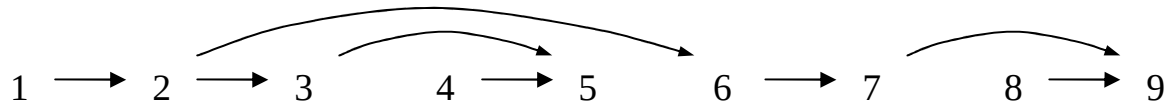
Таблица 33

Передача типа идентификаторам в объявлении

Продукция	Правила
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow int$	$T.type := integer$
$T \rightarrow real$	$T.type := real$
$L \rightarrow L_1, I$	$L_1.in := L.in$
	$l.in := L.in$
$L \rightarrow I$	$l.in := L.in$
$I \rightarrow I_1 [num]$	$I_1.in := array(num.val, l.in)$
$I \rightarrow id$	$addtype\{id.entry, I.in\}$



а)



б)

Рис. 36. Определение времени жизни значений атрибутов: *а* – граф зависимости для дерева разбора; *б* – узлы в порядке вычисления (*а*)

9.9.5. Назначение памяти атрибутам во время компиляции

Предположим, дана последовательность регистров для хранения значений атрибутов. Для удобства считаем, что каждый регистр может хранить любое значение атрибута. Для определения регистров, в которых будут вычисляться атрибуты, используется информация о времени жизни атрибутов.

Пример 41. Предположим, что атрибуты вычисляются в порядке, заданном номерами узлов в графе зависимости (рис. 36, *б*). Время жизни каждого узла начинается в момент вычисления его атрибута и заканчивается, когда этот атрибут используется в последний раз. Например, время жизни узла 1 заканчивается, когда вычислен атрибут узла 2, поскольку 2 – единственный узел, зависящий от 1. Аналогично время жизни узла 2 заканчивается при вычислении атрибута узла 6.

На рис. 37 показано вычисление атрибутов с использованием минимально возможного количества регистров. Узлы графа зависимости D дерева разбора рассматриваются в порядке их вычисления. Изначально имеется пул регистров $r_1, r_2, \ldots, r_n$. Если атрибут b определен семантическим правилом $b := f(c_1, c_2, \ldots, c_k)$, то при вычислении b может закончиться время жизни одного или нескольких c_i. Соответствующие им регистры освобождаются после вычисления b. По возможности b вычисляется в регистре, хранившем один из $c_1, c_2, \ldots, c_k$.

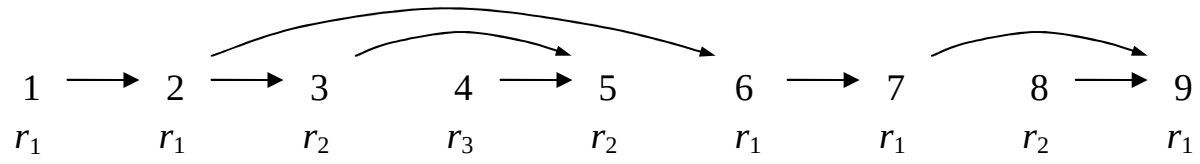


Рис. 37. Регистры, используемые для значений атрибутов на рис. 36

Регистры, используемые в процессе вычисления графа зависимостей на рис. 36, *б*, показаны на рис. 37. Вычисления начинаются с узла 1 в регистре r_1. Время жизни узла 1 заканчивается при вычислении узла 2, поэтому узел 2 использует тот же регистр r_1. Узел

3 получает новый регистр r_2, поскольку значение узла 2 потребуется при вычислении узла 6.

9.9.6. Устранение копий

Можно усовершенствовать метод, приведенный на рис. 37, рассматривая правила копирования как специальный случай. Правило копирования имеет вид $b := c$. Так что если значение c находится в регистре r, то там же автоматически размещается и значение b. Количество атрибутов, определяемых правилами копирования, может оказаться значительным, и явного копирования желательно избегать.

При рассмотрении узла m вначале проверяется, не определяется ли его значение правилом копирования. Если это так, следовательно, его значение уже находится в регистре и m присоединяется к классу эквивалентности со значением в этом регистре. Кроме того, такой регистр возвращается в пул только в конце времени жизни всех узлов со значениями в этом регистре.

Пример 42. На рис. 38 использован знак равенства для указания узлов, определяемых правилом копирования. Исходя из синтаксически управляемого определения в табл. 33, находим, что тип, определенный в узле 1, копируется в каждый элемент списка идентификаторов; в результате узлы 2, 3, 6 и 7 на рис. 36 являются копиями 1.

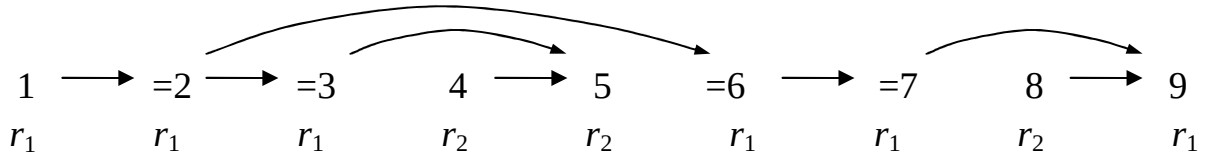


Рис. 38. Использование регистров с учетом правил копирования

Поскольку 2 и 3 являются копиями 1, их значения могут быть получены из регистра r_1 (см. рис. 38). Время жизни 3 заканчивается при вычислении 5, но регистр r_1, хранящий значение 3, не возвращается при этом в пул, поскольку еще не завершено время жизни 2 из класса эквивалентности.

Вопросы

1. Какие существуют способы записи семантических правил в компиляторах?

2. Какие атрибуты называются синтезированными и наследуемыми?

3. В чем основные различия между наследуемыми и синтезируемыми атрибутами?

4. Как используется стек для вычисления значений атрибутов?

5. Что представляет собой правило вычисления атрибутов?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Гордеев А.В.*, *Молчанов А.Б.* Системное программное обеспечение. – СПб.: Питер, 2002. – 736 с.

2. *Карпов Ю.Г.* Теория и технология программирования. Основы построения трансляторов. – СПб.: БХВ-Петербург, 2005. – 272 с.

3. *Ахо А.*, *Сети Р.*, *Ульман Дж.* Компиляторы: принципы, технологии и инструменты: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 768 с.

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

5. *Компаниец Р.И.*, *Маньков Е.В.*, *Филатов Н.Е.* Системное программирование. Основы построения трансляторов. – СПб.: КОРОНА-принт, 2000. – 256 с.

Оксана Георгиевна **ГАНИЧЕВА**

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Редактор *Н.А. Бачурина*
Компьютерная верстка: *Т.С. Камыгина*
Лицензия А № 165724 от 11 апреля 2006 г.

Подписано в печать 17.01.11. Тираж 300 экз.
Уч.-изд. л. 9,82. Формат 60×84 $^{1}/_{16}$. Усл. 11,16 п. л.
Гарнитура Таймс. Заказ 14.

ГОУ ВПО «Череповецкий государственный университет»
162600, г. Череповец, пр. Луначарского, 5