**Синтаксический анализ КС-языков методом рекурсивного спуска**

Рекурсивный спуск — это эффективный и простой нисходящий алгоритм распознавания. Он состоит в следующем.

Для каждого нетерминала грамматики (понятия, конструкции языка), то есть для каждой синтаксической диаграммы, записывается отдельная распознающая процедура. При этом соблюдаются следующие соглашения:

1. Перед началом работы процедуры текущим является первый символ анализируемого понятия (см. рис. 20).

2. В процессе работы процедура считывает все символы входной цепочки, относящиеся к данному нетерминалу (выводимые из данного нетерминала) или

сообщает об ошибке. Если правила для данного нетерминала содержат в правых частях другие нетерминалы (синтаксическая диаграмма данного нетерминала содержит другие нетерминалы), то процедура обращается к распознающим процедурам этих нетерминалов для анализа соответствующих частей входной цепочки.

3. По окончании работы процедуры текущим становится первый символ, следующий во входной цепочке за данной конструкцией языка (символами, выводимыми из данного нетерминала).



*Рис. 20. Текущий символ в начале и конце работы распознающей процедуры нетерминала А*

Распознавание начинается вызовом распознающей процедуры начального нетерминала.

При этом текущим символом, как это следует из п. 1, должен быть первый символ входной цепочки.

По завершении работы начальной процедуры текущим должен быть символ «конец текста».

Таким образом, анализ методом рекурсивного спуска всегда строится по следующей схеме:

**NextCh**; {Чтение первого символа в переменную Ch}

**S**; {Вызов распознающей процедуры нач.

 нетерминала}

**if Ch <> EOT then** {Проверка исчерпания входной

 цепочки}

**Error**;

Название «рекурсивный спуск» обусловлено тем, что при наличии в грамматике самовложения вызовы распознающих процедур будут рекурсивными.

Процесс распознавания развивается от начального нетерминала (корень дерева разбора) через вызов процедур для промежуточных нетерминалов (внутренние вершины дерева) к анализу отдельных терминальных символов (листья дерева).

 Это нисходящий разбор.

Каждая распознающая процедура строится по соответствующей синтаксической диаграмме, которая играет роль схемы алгоритма.

Соответствие участков диаграмм и фрагментов распознающих процедур показано в табл. 3.

В таблице участки диаграмм обозначаются D D1, D2,..., Dn. Соответствующие этим участкам фрагменты программы-распознавателя (распознающих процедур) обозначены P(D),P(D1,),P(D2),...,P(Dn).



Принцип работы анализатора, который строится по предлагаемым схемам, состоит в том, что, анализируя очередной символ входной цепочки, распознаватель выбирает путь движения по синтаксической диаграмме, соответствующий этой цепочке.

**Пример: анализатор многочленов**

Пользуясь методом рекурсивного спуска, запрограммируем синтаксический анализатор многочленов, правила записи которых определены диаграммами на рис. 17— 19.

Будем считать, что анализатор должен просто отвечать на вопрос, является ли введенная пользователем строка правильно записанным многочленом.

**Основная программа анализатора**

Вначале необходимо подготовить текст для чтения анализатором. Было бы неправильно делать так, чтобы в основной программе и распознающих процедурах отражались особенности представления входной цепочки, то есть программировать таким образом, чтобы основные части анализатора зависели от того, считываются ли символы из файла, вводятся с терминала, извлекаются из окна редактора текста или получаются как-либо по-другому.

Эта специфика будет скрыта в нескольких процедурах, которые только и будут зависеть от конкретного представления входной цепочки. Предусмотрим, что подготовка входного текста к чтению выполняется процедурой ResetText:

***begin***

 *ResetText;*

Далее вызываем распознающую процедуру начального нетерминала, которым в нашей задаче является «Многочлен»:

 *Polynom;*

Процедура Polynom считывает символы входной цепочки, проверяя, соответствует ли их порядок синтаксису многочленов. Если необходимо, она обращается к другим распознающим процедурам. В случае, когда входная цепочка действительно содержит многочлен, процедура Polynom оставляет текущим (содержащимся в переменной ch) символ, следующий за многочленом. Если при анализе многочлена обнаруживается ошибка, вызывается процедура Error, останавливающая работу всего анализатора.

Для завершения анализа остается проверить, не содержится ли за правильно записанным многочленом во входной цепочке лишних символов, то есть следует ли за многочленом символ «конец текста»:

***if*** *Ch <> EOT t****hen***

 *Error('Ожидается конец текста’)*

***else***

 *WriteLn('Правильно');*

 *WriteLn;*

***end.***

**Константы, переменные и вспомогательные процедуры**

Теперь можно записать начало программы с описаниями констант и переменных, использованными в основном блоке.

Кроме глобальной переменной **ch**, обозначающей текущий символ, предусмотрим глобальную переменную **Pos**, которая будет хранить номер этого символа во входной цепочке.

**program** ParsePolу;

**const**

 EOT = chr(0); { Признак "конец текста" }

**var**

 Ch : **char**; { Очередной символ }

 Pos : **integer**; { Номер символа }

Взаимодействие с входным текстом будут выполнять процедуры *ResetText* — готовит входную цепочку к считыванию распознавателем, и *NextCh* — читает очередной символ входной цепочки, помещая его в переменную **Ch**.

Использование глобальных переменных в процедурах, вообще-то, плохая

практика. Хотя мы еще не начали обращаться к **ch** и **Pos** в процедурах распознавателя, но вот-вот сделаем это. Оправданием служит специфика задачи.

Синтаксический анализатор и компилятор — своеобразные программы, в которых используется не слишком много переменных. А в нашем анализаторе переменных будет и вовсе две — только что определенные **Ch** и **Pos**.

Их передача в процедуры через параметры загромоздила бы программу, в то время как обращение за очередным символом, например к процедуре *Nextch*, все равно выглядело бы всегда одинаково:

NextCh(Ch, Pos).

Предусмотрим чтение исходных данных (записи многочлена) из стандартного входного файла.

Сообщения распознавателя будут выводиться в стандартный выходной файл (т. е. на экран). В этом случае процедура, подготавливающая текст, запишется так:

{ Подготовить текст }

**procedure** ResetText;

**begin**

**WriteLn**('Введите многочлен от х с целыми коэфф-тами');

Pos := 0;

NextCh; { Чтение первого символа }

**end**;

При чтении очередного символа будем игнорировать пробелы, считая их незначащими. Это позволит при записи исходного многочлена применять пробелы, например, для отделения одного слагаемого от другого.

**procedure** NextCh;

{ Читать следующий символ }

**begin**

**repeat**

 Pos := Pos+1;

 **if** not eoln **then**

 Read(Ch)

 **else begin**

 ReadLn;

 Ch := EOT;

 **end;**

**until** Ch <> ' ';

**end;**

Такой пропуск пробелов делает их допустимыми в любом месте записи многочлена. К примеру, такая строка должна теперь считаться правильной:

123х^4 + 567х + 89

Это не соответствует соглашениям современных языков программирования.

Более совершенное решение вопроса о пробелах будет рассмотрено в следующей главе, пока же примем простейший подход, который все же предпочтительней полного запрета пробелов.

Процедура, реагирующая на ошибку, тоже зависит от соглашений по вводу и выводу. Как мы уже условились, она будет выдавать сообщение на экран:

**procedure** Error(Message: string); { Ошибка }

{ Message - сообщение об ошибке }

**begin**

 WriteLn ( ' ^' : Pos);

 WriteLn('Синтаксическая ошибка: ', Message);

 Halt; { Прекращение работы анализатора }

**end;**

Вывод знака ' ^' в позиции Pos позволяет указать стрелкой на символ, вызвавший ошибку. Диалог с анализатором может быть таким:

Введите многочлен от X с целыми коэфф-тами

2х + 1.2

 ^

Синтаксическая ошибка: Ожидается конец текста

Реакция программы в этом примере может показаться непонятной, хотя она совершенно корректна. Получение такого сообщения означает: «Если бы на этом месте запись многочлена закончилась, было бы синтаксически правильно. Но в указанном месте стоит неподходящий символ».

Понятно, что распознаватель не может знать наших желаний и содержательно реагировать на попытку записать вещественное число вместо целого.