**Распознающие процедуры**

Для каждого нетерминала грамматики многочленов, то есть для каждой синтаксической диаграммы (рис. 17-19) записываем одну распознающую процедуру.

Первой — процедуру для нетерминала «Многочлен» — начального нетерминала грамматики (рис. 17). Трудность, однако, состоит в том, что диаграмма на рис. 17 неудобна для программирования — она содержит цикл с выходом из середины, в то время как в Паскале такого цикла нет.

Заменим диаграмму эквивалентной, содержащей цикл с предусловием (с выходом в начале).



*Рис.21 Преобразованная диаграмма «Многочлен»*

Теперь не составляет труда записать распознающую процедуру, структура которой в точности повторяет структуру диаграммы: на диаграмме три последовательно соединенных участка — в процедуре три оператора, выполняемых один за другим: **if**, вызов процедуры *Addend*, цикл **while**.

**procedure** Polynom; { Многочлен }

**begin**

 **if** Ch **in** [' +', ' -' ] **then**

 NextCh;

 Addend; { Слагаемое }

 **while** Ch **in** [' +', ' -' ] **do begin**

 NextCh;

 Addend;

 **end;**

**end;**

Следующий нетерминал — «Слагаемое». Однако диаграмма, показанная на рис. 18, не разделяется на типовые фрагменты, что затрудняет программирование распознавателя.

Неструктурированность обусловлена фрагментом, который помечен на рисунке знаком «?». Преобразуем диаграмму в эквивалентную, но состоящую только из совокупности типовых структур (рис. 22).

Для этого изобразим отдельно две ветви: одна соответствует слагаемому, начинающемуся с числа, другая — с буквы х. Фрагмент, выделенный на исходной диаграмме пунктирной рамкой, преобразуем в нетерминал «Степень».



*Рис.22 Синтаксические диаграммы «Слагаемое» и «Степень»*

Программирование распознающих процедур *Addend* (слагаемое) и *Power* (степень) теперь выполняется легко: диаграммы служат схемами алгоритмов.

**procedure** Addend; { Слагаемое }

**begin**

 **if** Ch = ' x' **then begin**

 NextCh;

 Power; { Степень }

**end**

**else begin**

 Number; { Целое }

 **if** Ch = 'x' **then begin**

 NextCh;

 Power;

 **end;**

 **end;**

**end;**

**procedure** Power; { Степень }

**begin**

 **if** Ch = '^ ' **then begin**

 NextCh;

 Number;

 **end;**

**end;**

Полезно обратить внимание на дисциплину вызова процедуры NextCh. Следующий символ считывается, когда опознан текущий.

Последняя распознающая процедура — для «Целого». И в этом случае тоже можно преобразовать исходную диаграмму (см. рис. 19), отделив блок, соответствующий первой цифре (обязательной), от остальных блоков (рис. 23).



*Рис.23 Синтаксическая диаграмма «Целое»*

Предусматривать отдельную распознающую процедуру для «почти терминального символа» «цифра» неразумно. Проще и наглядней непосредственно проверять принадлежность очередного знака множеству цифр.

**procedure** Number; { Целое }

**begin**

 **if** Ch in ['0'..'9'] **then**

 NextCh

 **else**

 Error('Число начинается не с цифры');

 **while** Ch in ['0'..'9'] **do**

 NextCh;

**end;**

Распознаватель готов. Осталось только расположить в программе написанные процедуры в правильном порядке — описание процедуры поместить перед ее вызовом. Поскольку рекурсии в этом примере нет, сделать это легко.

На рассмотренном примере мы продемонстрировали, что синтаксический анализатор методом рекурсивного спуска можно написать «почти так же быстро, как мы вообще можем писать»

**Требование детерминированного распознавания**

Уже в ходе предыдущего рассмотрения можно было заметить, что рекурсивный спуск позволяет построить анализатор не для любой КС-грамматики.

Ограничения возникают при анализе направляющих символов отдельных ветвей синтаксической диаграммы.

Множество направляющих символов first(D) ветви D синтаксической диаграммы образуют терминальные символы, которые могут встретиться первыми при движении по диаграмме вдоль этой ветви.

Движение по диаграмме предполагает, что при достижении прямоугольного блока, изображающего нетерминал, оно продолжается по диаграмме этого нетерминала с последующим возвратом на исходную диаграмму.

Порядок действий анализатора, работающего по методу рекурсивного спуска, и определяется анализом направляющих символов отдельных ветвей синтаксической диаграммы.

Чтобы алгоритм работал без возвратов, а выбор направления движения по диаграмме выполнялся однозначно, должно соблюдаться требование детерминированного распознавания:

*В каждом разветвлении синтаксической диаграммы множества направляющих символов отдельных ветвей не должны попарно пересекаться.*

**

В рассмотренной ранее грамматике многочленов требование детерминированного распознавания всегда соблюдается.

Возьмем, например, диаграмму, показанную на рис. 21. На ней две точки ветвления. Первая — на входе в диаграмму, вторая — на выходе.

В первом разветвлении три ветви. Множества направляющих символов этих ветвей:

 [' +' ] — верхняя ветвь;

[' 0' . . ' 9', 'х']— средняя ветвь;

[' -' ] — нижняя ветвь.

Как видим, эти множества не пересекаются.

В правой точке происходит ветвление на два направления. Одно ведет на выход из диаграммы, множество направляющих символов этой ветви состоит из символа «конец текста»: [⊥]. Другая ветвь соответствует очередному витку цикла, ее множество направляющих символов равно [' +', ' -' ].

Пересечения множеств снова нет.

**LL-грамматики**

*LL(k)-грамматикой называется КС-грамматика, в которой выбор правила в ходе левостороннего вывода однозначно определяется не более чем k очередными символами входной цепочки, считываемой слева направо.*

Название «LL» происходит от двух слов «left» (левый), встречающихся в описании хода распознавания в LL-грамматике: левосторонний вывод при чтении слева.

Самыми удобными для распознавания, конечно же, являются LL(1) грамматики, в которых выбор направления распознавания однозначно определяется очередным входным символом.

Сформулированное требование детерминированного распознавания при рекурсивном спуске есть не что иное, как необходимость того, чтобы используемая грамматика относилась к классу LL(1).

*Рекурсивный спуск — это детерминированный метод нисходящего разбора КС- языков, порождаемых LL(1) -грамматиками.*