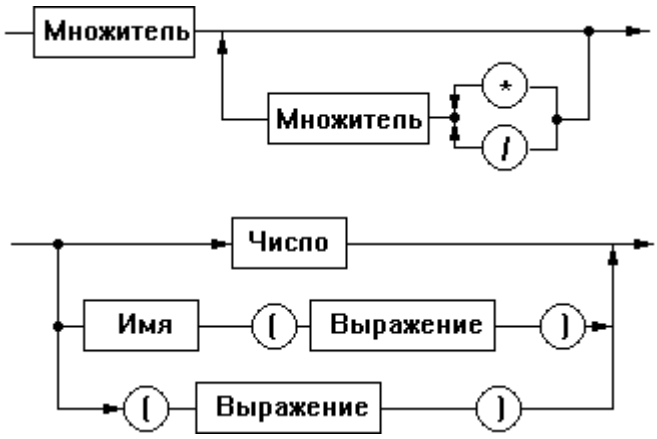
С.Свердлов

ВВЕДЕНИЕ В МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие



Министерство образования Российской Федерации
Вологодский государственный педагогический институт

С.З.Свердлов

ВВЕДЕНИЕ В МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Вологда
1994

ББК
22.183.492я 73
С24

Печатается по решению
редакционно-издательского совета
ВГПИ

Свердлов С.З. **Введение в методы трансляции:** Учебное пособие. – Вологда: Издательство "Русь", 1994. – 80 с.

В пособии рассматриваются алгоритмы, используемые при построении компиляторов и интерпретаторов языков программирования. Показаны возможности применения этих алгоритмов при разработке прикладных программ.
Изложение ведется с использованием языка программирования Паскаль и сопровождается большим количеством примеров.
Пособие предназначено для студентов, специализирующихся по информатике. Может быть полезно программистам-практикам и всем, кто интересуется программированием.

Рецензенты:
С.Ю.Ржеуцкая, доцент кафедры информатики ВоПИ; *С.Е.Микрюков*, доцент кафедры информатики и вычислительной техники ВГПИ; *А.В.Тупицын,* начальник управления по информационному обеспечению администрации Вологодской области.

$$C\,\frac{2404010000(2404000000)-038}{\Gamma\,76(03)-94}$$

ISBN 5-87822-049-0

1. Алгоритмы обработки текста

Разрабатывая программу для компьютера на каком-либо языке программирования, вы, наверное, задумывались о той роли, которую при этом играет компилятор или интерпретатор. Это он выдает иногда сообщения "Syntax error" или, например, "Error 42: Error in expression", означающие наличие в программе синтаксической ошибки.

Каким образом компилятор или интерпретатор анализируют текст программы, обнаруживая там *все* синтаксические ошибки? Очевидно, при этом выполняется весьма сложная работа: ведь структура программы может быть очень непроста. Не нужно забывать и то, что, кроме обнаружения синтаксических ошибок, интерпретатор должен исполнить программу, а компилятор выполнить ее перевод на другой язык (чаще всего в машинный код).

В этом пособии рассматриваются те алгоритмы, которые используются компиляторами и интерпретаторами для анализа текста программы. И хотя не каждому программисту приходится разрабатывать компилятор или интерпретатор языка программирования, но те подходы, которые используются при их построении, применимы к гораздо более широкому классу задач обработки текста. Подчеркнем при этом, что компиляторы и интерпретаторы решают именно задачу обработки текста – ведь программа, подлежащая интерпретации или компиляции, – это текст, последовательность символов.

Предметом нашего рассмотрения будут алгоритмы обработки, применимые к текстам, имеющим сложную (но строго определенную) структуру.

Текстовая форма представления информации является наиболее общей. Действительно, любые данные (текст, графика, звук), хранящиеся в компьютере, представляются в двоичной форме, а следовательно, их можно записать как последовательность символов "0" и "1". Таким образом, рассматриваемый нами класс алгоритмов применим к широкому кругу задач обработки информации различной природы, а использование этих алгоритмов при разработке прикладных программ должно способствовать повышению их качества и надежности.

Изложение будет вестись с использованием языка программирования Паскаль и привлечением при необходимости его расширений, имеющихся в системе программирования Турбо Паскаль. В качестве примера будет написан несложный, но полезный транслятор.

1.1. Пример задачи обработки текста

Рассмотрим в качестве примера задачу, которая была предложена во втором туре I Вологодской областной олимпиады по информатике.

Задача. Составить программу, вводящую в символьной форме два многочлена от x с целыми коэффициентами и выводящую их произведение в символьной форме в порядке убывания степеней. (Суммарная степень полиномов не превышает 255. Длина записи каждого – не более 80 символов).

Вот пример работы такой программы:

```
Перемножение полиномов
----------------------

1-й полином
>3x^2+3x-5
2-й полином
>x^3 - 2x

Произведение:
```

```
3x^5+3x^4-11x^3-6x^2+10x
```

Программа печатает запрос, в ответ на который можно ввести запись первого, а затем второго полинома в привычной, используемой в математике форме. Предполагается, что можно применять знак возведения в степень "^". Обработав полученные исходные данные, программа печатает результат – произведение многочленов в таком же естественном виде. Естественность, привычность записи исходных данных и результатов, а следовательно, удобство в использовании являются важным достоинством такой программы.
Однако реализовать предлагаемый способ ввода данных не очень просто. Действительно, слагаемые полинома могут записываться по-разному: с числовым коэффициентом и без него (`3x^5` и `x^3`); с показателем степени и без показателя (`x^3` и `2x`); в записи могут быть пробелы, порядок следования и количество слагаемых не фиксировано. Программа же должна корректно обработать все эти ситуации.
Возникает и еще ряд вопросов. Как должна реагировать программа, если запись полинома неверна? Какую запись следует считать правильной?
Конечно, наша программа перемножения полиномов не должна аварийно завершаться, если пользователь допустил ошибку в записи исходных данных. Ее реакция на неверный ввод может быть, например, такой:

```
>3x^x - 5
    ^
Синтаксическая ошибка: Ожидается цифра
```

Вообще, любая добротная программа обязана адекватно реагировать на *любые*, в том числе неверные, действия пользователя, которые не должны приводить к ее аварийному завершению.

1.1.1. Разработка программы

Теперь приступим к решению поставленной задачи, используя метод пошаговой детализации. Сформулируем общий план решения:

1. Напечатать заголовок.

2. Ввести текст 1-го полинома.

   Поскольку по условию задачи текст записи каждого полинома не превышает 80 символов, его можно хранить в переменной типа строка (**`string`**).

3. Проанализировать запись 1-го многочлена и преобразовать его в удобную для дальнейших вычислений форму.

   Для выполнения действий с полиномами (в том числе перемножения) удобно хранить их в программе в виде массива коэффициентов, количество которых равно порядку полинома, а индекс каждого коэффициента равен степени соответствующего слагаемого.
   Этот этап является в задаче самым сложным. Именно здесь наша программа должна будет решить задачу трансляции – перевод с языка записи полинома во внутреннюю форму – массив коэффициентов.

4. Ввести текст 2-го полинома.

5. Проанализировать запись 2-го многочлена и преобразовать его в массив коэффициентов.

6. Перемножить полиномы.

   Имея представление обоих полиномов-сомножителей во внутреннем формате и выполняя необходимые вычисления с их коэффициентами, мы получаем коэффициенты полинома-произведения.

7. Напечатать результат.

Запишем начало программы, где определим необходимые константы, типы данных и переменные в соответствии с уже принятыми решениями.

```
program PolyMult;
{ Перемножение полиномов }

const
   Nmax = 255; { Максимальный порядок полинома }
type
   StrType  = string[80];{ Тип входной строки  }
   PolyType =            { Тип полиномов       }
      record
         n  : integer;  { порядок полинома     }
         a  : array [0..Nmax] of integer;
                        { массив коэффициентов }
      end;
var
   P1, P2 : PolyType; { сомножители  }
   Q      : PolyType; { произведение }
   S      : StrType;  { входная строка }
```

Обратите внимание, что полиномы отнесены к типу `PolyType`, который представляет собой запись, содержащую, кроме упоминавшегося массива коэффициентов, величину `n` – фактический порядок полинома.
Теперь, пользуясь предварительно составленным планом, напишем основную программу.

```
begin
   WriteLn('Перемножение полиномов');
   WriteLn('----------------------');
   WriteLn;
   WriteLn('1-й полином');
   Write('>');
   ReadLn(S);
      { Ввод текста 1-го полинома }
   Translate( S, P1 );
    { Трансляция 1-го полинома }
   WriteLn('2-й полином');
   Write('>');
   ReadLn(S);
      { Ввод текста 2-го полинома }
   Translate( S, P2 );
    { Трансляция 2-го полинома }
   MultPoly( P1, P2, Q );
      { Перемножение }
   WriteLn;
   WriteLn('Произведение:');
   WritePoly( Q );
      { Печать результата }
   WriteLn;
end.
```

В программе мы лишь заменили названия пунктов первоначального плана вызовами соответствующих процедур. Описания этих процедур еще не составлены, хотя перечень и смысл их параметров уже определены. Задача разбита на подзадачи, к решению которых мы и переходим.
Начнем с процедуры, которая выполняет перемножение. Ее входными параметрами являются полиномы-сомножители, выходным – полином-произведение. И входные, и выходной параметры являются полиномами, представленными в виде совокупности массива

коэффициентов и величины, задающей порядок полинома т.е. относятся к типу PolyType. Обозначив сомножители X и Y, а результат – Z, запишем заголовок процедуры.

```
procedure MultPoly( var X, Y, Z : PolyType );
```

Основная идея вычисления коэффициентов полинома-произведения состоит в том, что при попарном перемножении слагаемых первого и второго полинома получающееся произведение участвует (в качестве одного из слагаемых) в формировании того члена полинома-результата, степень которого равна сумме степеней сомножителей. То есть при перемножении i-го члена полинома X и j-го члена полинома Y получается величина, которая должна быть добавлена к i+j - му слагаемому Z:

```
Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
```

Такое вычисление нужно выполнить для всех сочетаний i и j, не забыв присвоить нулевые начальные значения коэффициентам Z и побеспокоиться об определении степени полинома Z. Получаем:

```
procedure MultPoly( var X, Y, Z : PolyType );
   { Z = X*Y }
var
   i, j  : integer;
begin
   ClearPoly( Z );   { "Обнуление" Z }
   for i := 0 to X.n do
      for j := 0 to Y.n do
         Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
   { определение степени произведения }
      with Z do begin
         n := Nmax;
         while ( n > 0 ) and ( a[n] = 0 ) do
            n := n-1;
      end;
end;
```

Процедура, ClearPoly, выполняющая "очистку" полинома, выглядит так:

```
procedure ClearPoly( var P : PolyType );
var
   i : integer;
begin
   for i := 0 to Nmax do
      P.a[i] := 0;
   P.n := 0;
end;
```

Теперь займемся печатью полинома. Слагаемые должны выводиться в порядке убывания степеней. Слагаемому может предшествовать знак. Коэффициент при ненулевой степени печатается, если он не ноль и не единица. Буква x выводится для ненулевых степеней, а показатель степени (и знак "^" перед ним), – если он больше единицы. Учитывая это, напишем:

```
procedure WritePoly( P : PolyType );
var
   i : integer;
```

```
begin
   with P do
      for i := n downto 0 do begin
         if ( a[i]>0 ) and ( i<>n ) then
            write( '+' );
         if ( a[i]=-1 ) and ( i>0 ) then
            write( '-' );
         if ( n=0 ) or ( abs(a[i])>1 ) or
            ( i= 0 ) and ( a[i] <> 0 )
         then
            write( a[i] );
         if ( i>0 ) and ( a[i]<>0 ) then
            write( 'x' );
         if ( i>1 ) and ( a[i]<>0 ) then
            write( '^', i )
      end;
end;
```

Осталось реализовать процедуру, которая выполнит преобразование текстовой записи полинома в массив коэффициентов. Ее заголовок должен быть таким:

```
procedure Translate( S :StrType; var P :PolyType );
```

Однако, чтобы запрограммировать такое преобразование[1], обеспечив при этом контроль корректности записи полинома, необходимо обсудить, что такое

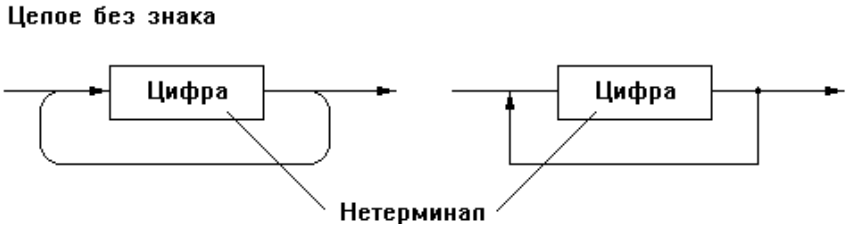
2. Синтаксический анализ

Чтобы правильно выполнить обработку текста, нужно знать или сформулировать правила его записи и истолкования.
Правила записи текста, точнее правила следования символов текста – *синтаксис.*
Синтаксис определяет лишь форму представления текста, но не приписывает ему смысла.
Содержание, смысл, правила истолкования текста – *семантика*.
Множество всех текстов, удовлетворяющих заданному синтаксису, – *язык.* Можно говорить о языке записи многочленов, языке формул, языке записи вещественных чисел, языке Паскаль или Бейсик и т.д.
Синтаксис и семантика связаны. Если рассмотреть определенный язык, то смысл, который придается тексту на этом языке, определяется его формой.
Проверка соответствия текста заданному синтаксису – *синтаксический анализ*.
Программа, выполняющая синтаксический анализ – *синтаксический анализатор*.
Результат работы синтаксического анализатора прост: проверяемый текст либо признается правильным (соответствующим заданному синтаксису), либо обнаруживается несоответствие – *синтаксическая ошибка*. Синтаксический анализатор часто также называют *распознавателем*.
Обрабатывая какой-либо текст (запись полинома, формулы, текст программы на Паскале или Бейсике, команду операционной системы, содержимое поля базы данных, ответ учащегося на вопрос обучающей программы), обычно бывает недостаточно только выяснить, правильно ли записан этот текст. Нужно выполнить его смысловую обработку – преобразовать полином, вычислить по формуле, выполнить команду с учетом всех ее параметров и т.д.
Почему же мы обсуждаем лишь анализ текста? Дело в том, что *обработка текста строится на базе его синтаксического анализа*. В ходе анализа выявляется структура текста, вычленяются отдельные его элементы, которым можно приписать определенный смысл.

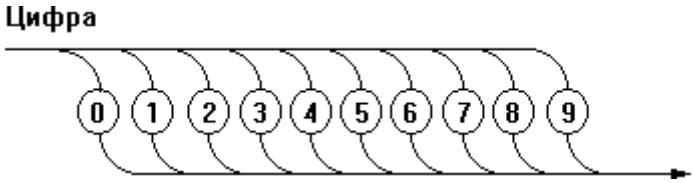
[1] Решение задачи трансляции полинома дано в разделе 3.2, а полный текст программы перемножения полиномов – в приложении 8.3.

2.1. Синтаксические диаграммы

Чтобы решать задачу синтаксического анализа, необходимо иметь способ строгого определения синтаксиса. Описание правил записи текста на естественном языке (например, русском) может оказаться громоздким и неоднозначным. Существует несколько способов формального задания синтаксиса. Среди них наглядностью, простотой и удобством отличается представление правил записи текста с помощью синтаксических диаграмм. В частности, синтаксические диаграммы использованы Н.Виртом при описании языка Паскаль. Рассмотрим использование диаграмм на примере определения правил записи простых элементов языка Паскаль.

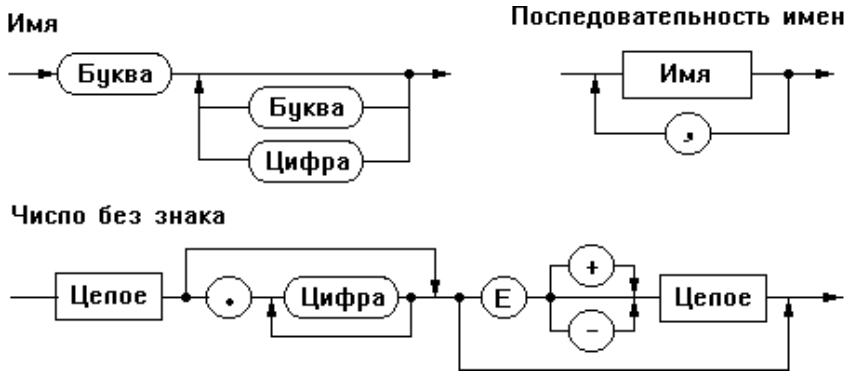


На рисунке показаны два варианта изображения синтаксической диаграммы, задающей правила записи целого числа без знака. В одном случае направление возможного движения по диаграмме показано с помощью закруглений, а в другом – стрелками.
Любому пути от входа диаграммы к выходу соответствует правильный вариант текста. Например, путь от входа к выходу на диаграмме "Целое без знака" с однократным прохождением блока "Цифра" соответствует однозначному числу; пути, включающие несколько оборотов, соответствуют многозначным числам.
Диаграммы можно использовать как справочник по языку, синтаксис которого они определяют. Ответьте с помощью диаграммы "Целое без знака" на вопросы:
1. Может ли целое без знака не содержать ни одной цифры?
2. Можно ли в записи целого без знака использовать римские цифры?
Ответ на первый вопрос безусловно отрицательный: "Нет, не может". Действительно, на диаграмме нет пути, ведущего от входа к выходу, который не проходил бы через блок "Цифра". А вот ответа на второй вопрос наша диаграмма не содержит. Использованное на ней понятие "Цифра" должно быть уточнено, расшифровано.
Обычно это делают так:



Из этой диаграммы ясно, что имеются в виду арабские цифры от 0 до 9. Обратите внимание, что цифры на диаграмме изображены в кружках. Это означает, что каждый из этих символов может быть записан в текстах, синтаксис которых определяется диаграммой. Такие символы, из которых в конечном итоге состоит текст, называют *терминальными.* (*Терминальный* - окончательный, конечный, последний). Для изображения терминальных символов на синтаксических диаграммах могут также использоваться овалы.
В отличие от кружков или овалов прямоугольные блоки изображают понятия, которые должны быть определены через другие понятия и в конечном итоге через терминальные символы. Такие элементы синтаксиса называют *нетерминальными символами* или просто *нетерминалами.*
Очевидно, что каждому нетерминалу (понятию), встретившемуся на диаграмме, определяющей синтаксис некоторого языка (множества текстов), должна соответствовать

диаграмма, которая в качестве своего названия имеет название этого нетерминала и определяет его синтаксис. Вместе с тем, для таких часто используемых понятий, как "цифра" или "буква", было бы неудобно каждый раз приводить соответствующие диаграммы. Поэтому, имея в виду, что цифры используются арабские, а буквы – латинские (большие и маленькие), будем изображать на диаграммах понятия "цифра" и "буква" в овалах, считая их "почти терминальными" символами.
Сказанное проиллюстрируем еще некоторыми примерами синтаксических диаграмм.



Пользуясь диаграммами, ответьте на следующие вопросы (термином "целое" на диаграммах обозначено "целое без знака"):

1. Может ли имя начинаться с цифры?
2. Может ли имя состоять только из букв?
3. Может ли последовательность имен состоять из одного имени?
4. Может ли последовательность имен быть пустой?
5. Может ли число начинаться точкой?
6. Может ли число заканчиваться точкой?
7. С какого символа может начинаться число?
8. Может ли порядок числа содержать более двух цифр?
9. Может ли целая часть числа состоять из 1000 цифр?

Комментариев требуют вопросы 8 и 9. Следует дать утвердительные ответы на оба вопроса, хотя это может противоречить вашему представлению о правилах записи чисел в реальных программах. Дело в том, что ограничения на количество цифр (или другие подобные ограничения) хотя и можно выразить с помощью синтаксических правил и отразить в синтаксических диаграммах, но обычно этого не делают, считая такого рода требования относящимися к семантике языка. Их учет при определении синтаксиса мог бы существенно усложнить диаграммы, в то время как проверка этих требований может быть легко выполнена программой-транслятором как часть семантической обработки.

2.2. Решение задачи синтаксического анализа

Задача. Взяв в качестве примера понятие "Имя", синтаксис которого задан одной из приведенных выше диаграмм, построим синтаксический анализатор, то есть запишем программу, которая проверяет, *является ли текст именем*.
Будем считать, что анализируемый текст заносится в переменную-строку `S`. Примем также, что текст должен завершаться специальным символом "конец текста". Это избавит нас от необходимости проверять выход за пределы строки при выборке очередного символа. Символ "конец текста" выберем таким, чтобы он не мог встретиться среди символов анализируемого понятия[2]. Выберем в качестве признака конца текста символ с порядковым

[2] В теории синтаксического анализа, перевода и компиляции символ "конец текста" обозначают обычно знаком "⊥".

номером 0 – `chr(0)` (На Турбо Паскале можно записать `#0`). Обозначим `i` – номер очередного символа строки `S`. Выполним ввод и подготовку текста к анализу:

```
WriteLn('Введите имя');
ReadLn(S);      { Прочитать строку }
S := S + #0;    { Приписать "конец текста" }
i := 1;         { Встать на первый символ  }
```

Теперь, считывая текст символ за символом, нужно проверить, является ли он правильно записанным именем.
Убедимся, что первой записана буква, и, если это так, перейдем к следующему символу; если нет – значит обнаружена ошибка.

```
if S[i] in ['a'..'z', 'A'..'Z'] then
   i := i + 1 { На следующий символ }
else
   Error;      { Ошибка }
```

Обработка ошибки выполняется процедурой `Error`, которая печатает диагностическое сообщение и *прекращает работу программы*.
Если ошибки не обнаружено, продолжаем обработку, считывая все следующие за первым символом буквы и цифры:

```
while S[i] in ['a'..'z', 'A'..'Z', '0'..'9'] do
   i := i + 1;
```

Этот цикл завершается, как только встретится символ, отличный от буквы и цифры. Если таким символом является "конец текста", то значит текст содержит только правильно записанное имя; если нет – значит после имени в тексте есть другие символы, а, следовательно, текст именем не является. Записываем проверку:

```
if S[i] <> #0 then
   Error
else
      WriteLn('Правильно!');
```

Опираясь на здравый смысл, мы получили простейший синтаксический анализатор простейшего синтаксического понятия "Имя". Используемая в нем процедура, обрабатывающая ошибку, выглядит так:

```
procedure Error;
begin
   WriteLn('Синтаксическая ошибка');
   Halt;
end;
```

Проанализируем структуру получившейся программы. Нетрудно понять, что ее первая часть, отвечающая за подготовку текста к анализу, не зависит от анализируемого понятия, но может зависеть от того, что является источником текста – строка, вводимая с клавиатуры, файл и т.д. Не зависит от синтаксиса анализируемого текста и последняя часть анализатора, которая проверяет, является ли символ признаком "конец текста".
В то же время два центральных фрагмента (**`if S[i] in ...`** и цикл **`while S[i] in ...`**), наоборот, целиком определены синтаксисом анализируемого понятия. Более того, структура этой части анализатора повторяет структуру синтаксической диаграммы

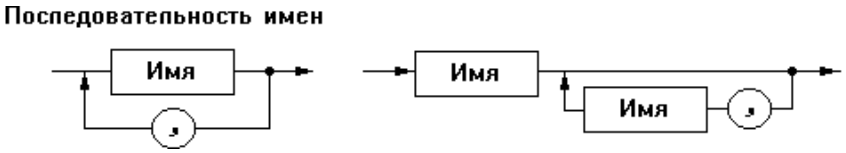
"Имя", представляя собой последовательность двух частей, одна из которых (**if...**) соответствует терминалу "буква", а другая (циклическая) – циклу на диаграмме.

Задача. Проверить, является ли текст последовательностью имен.

Начальная часть программы записывается аналогично предыдущему примеру:

```
WriteLn('Введите последовательность имен');
ReadLn(S);      { Прочитать строку }
S := S + #0;    { Приписать "конец текста" }
i := 1;         { Встать на первый символ  }
```

Далее необходимо обеспечить анализ последовательности имен, синтаксис которой задается диаграммой:

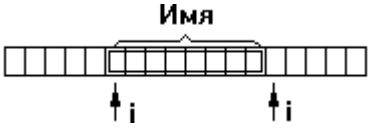


Для программирования анализатора на Паскале удобно преобразовать исходную диаграмму "Последовательность имен" в эквивалентную, представленную на рисунке справа. Цикл на преобразованной диаграмме может не реализовываться ни разу, что соответствует циклу с предусловием в программе анализатора.
На диаграмме "Последовательность имен" имеется нетерминал "Имя", анализатор для которого нами уже разработан. Естественным шагом будет его использование при анализе последовательности имен. Для этого следует оформить анализ имени как отдельную процедуру:

```
procedure Name( S : StrType; var i : integer );
```

Параметрами процедуры Name являются строка, содержащая текст, и номер символа.
Примем такие соглашения о порядке работы анализирующей процедуры:

- процедура вызывается, когда текущим (очередным) является *первый* символ анализируемого понятия;
- в ходе работы процедура считывает все символы анализируемого ею понятия. При обнаружении синтаксической ошибки вызывает процедуру Error, которая прекращает работу анализатора;
- после успешного завершения работы процедуры (без обнаружения ошибки) текущим должен быть символ, *следующий* в тексте за анализируемым процедурой понятием.



На схеме показано положение текущего символа перед началом и по окончании работы анализирующей процедуры Name.

Запишем оставшуюся часть программы анализа последовательности имен и процедуру-анализатор имени:

```
   Name( S, i );
   while S[i] = ',' do begin
      i := i + 1;
      Name( S, i );
   end;
   if S[i] <> #0 then
      Error
   else
      WriteLn('Правильно');

procedure Name( S : StrType; var i : integer );
begin
```

```
   if S[i] in ['a'..'z', 'A'..'Z'] then
      i := i + 1
   else
      Error;
   while S[i] in ['a'..'z','A'..'Z', '0'..'9'] do
      i := i + 1;
end;
```

Так же, как нетерминал "Имя" был использован при определении синтаксиса последовательности имен, так и понятие "Последовательность имен" может быть использовано в более общем контексте. В связи с этим, будет правильным преобразовать анализатор последовательности имен в процедуру:

```
   procedure NameList( S:StrType; var i:integer );
   begin
      Name( S, i );
      while S[i] = ',' do begin
         i := i + 1;
         Name( S, i );
      end;
   end;
```

Теперь основная программа, которая решает нашу задачу, может быть записана так:

```
   begin
      WriteLn('Введите последовательность имен');
      ReadLn(S);         { Прочитать строку }
      S := S + #0;       { Приписать "конец текста"}
      i := 1;            { Встать на первый символ }
      NameList( S, i ); { Проверить список имен }
      if S[i] <> #0 then
         Error
      else
         WriteLn('Правильно');
   end.
```

Нетрудно понять, что другая программа, в задачу которой входит проверка соответствия введенной строки текста заданному синтаксису, может быть построена аналогично. При этом можно заметить, что основная программа не зависит от того, что анализируется, а все особенности синтаксиса отражены в анализирующих процедурах.
В свою очередь структура каждой анализирующей процедуры определяется структурой синтаксической диаграммы, задающей синтаксис анализируемого процедурой понятия. Если на диаграмме есть два соединенных последовательно участка, то в анализирующей процедуре будет цепочка последовательно выполняемых операторов; циклический фрагмент на диаграмме приводит к появлению цикла в программе-анализаторе; точки ветвления на диаграмме порождают проверку условий в программе.
Синтаксическая диаграмма выполняет роль схемы алгоритма при написании синтаксического анализатора.

2.3. Построение синтаксического анализатора по синтаксическим диаграммам

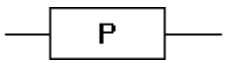
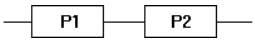
Обобщая использованные при решении предыдущих задач подходы, сформулируем правила построения синтаксического анализатора по синтаксическим диаграммам.
Синтаксический анализатор строится как совокупность анализирующих процедур. Каждой синтаксической диаграмме соответствует одна анализирующая процедура.

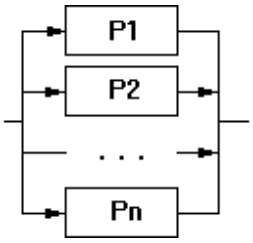
Каждая процедура считывает текст анализируемого ею понятия. В начале работы распознающей процедуры текущим является первый символ анализируемого ею текста. Процедура заканчивает работу, обнаружив синтаксическую ошибку или прочитав все символы анализируемого понятия. По окончании работы процедуры текущим становится символ, следующий за анализируемым понятием.
Алгоритм работы распознающей процедуры определяется структурой соответствующей синтаксической диаграммы.
В таблице показано соответствие фрагментов синтаксических диаграмм и порождаемых ими частей программы-анализатора.
Приняты обозначения: `Ch:char` - очередной символ анализируемого текста; `NextChar` - процедура, которая считывает следующий символ текста, делая его текущим (присваивает новое значение `Ch`). Переменная `Ch` является глобальной для распознающих процедур. P, P1, P2, ..., Pn - нетерминальные символы и соответствующие им распознающие процедуры.
Общий принцип работы анализатора, который строится по предлагаемым схемам, состоит в том, что, анализируя *один очередной символ* текста, распознаватель выбирает путь движения по диаграмме, который соответствует анализируемому тексту.
Чтобы выбор одного из возможных направлений движения по диаграмме был однозначен, должно соблюдаться *условие детерминированного распознавания*:
для каждой точки ветвления на синтаксических диаграммах множества направляющих символов отдельных ветвей не должны пересекаться.
Под *множеством направляющих символов* f нетерминала P понимается множество терминальных символов, с которых может начинаться нетерминал P.
Например, множество направляющих символов для целого без знака есть множество цифр. Множество направляющих символов имени – буквы.

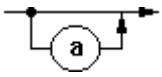
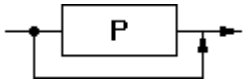
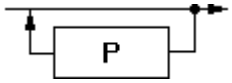
Правила программирования синтаксического анализатора

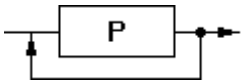
Диаграмма	Анализатор	Примечание

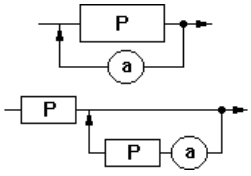
	`if Ch='a' then`
	`  NextChar`
	`else`
	`  Error;`
		Проверка терминального символа

	`P;`	Проверка нетерминала. Вызов процедуры P
	`P1;` `P2;`	Последовательность нетерминалов. Вызов процедур P1 и P2
Ветвления		

	`if Ch in f1 then`
	`  P1`
	`else if Ch in f2`
	`then`
	`  P2`
	`  ...`
	`else if Ch in fn`
	`then`
	`  Pn`
	`else`
	`  Error;`
		Общий случай.
		f1, f2,..., fn - множества направляющих символов каждой ветви
		
		fi*fj = [ ] при
		i **in** 1..n;
		j **in** 1..n;
		i<>j

	`if Ch = 'a' then` `  NextChar;`	Частный случай
	`if Ch in f then` `  P;`	Частный случай f - множество направляющих символов P
Циклы		
	`while Ch in f do` `  P;`	Цикл с предусловием f - множество направляющих символов P

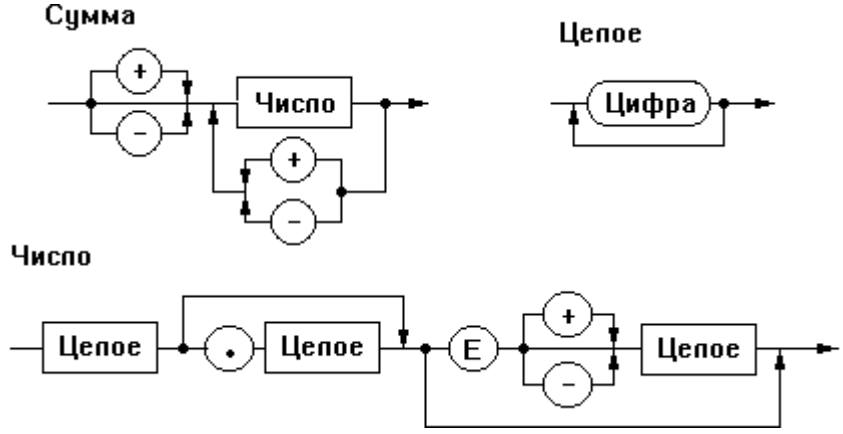
	`repeat`
	`  P;`
	`until not(Ch in f);`
		Цикл с постусловием

	`P;`
	`while Ch = 'a' do`
	`begin`
	`  NextChar;`
	`  P;`
	`end;`
		Обобщенный цикл

Пользуясь приведенными правилами, построим распознаватель (синтаксический анализатор) понятия "Сумма чисел".

Задача. Проверить, является ли текст суммой вещественных чисел.

Определим синтаксис понятия "сумма" с помощью совокупности синтаксических диаграмм.



При разработке распознавателя внесем ряд усовершенствований в используемую технологию. Непосредственное взаимодействие с анализируемым текстом будут осуществлять лишь процедуры `NextChar` (для получения очередного символа) и `ResetText` для подготовки текста к обработке. В этом случае основная часть распознавателя не будет зависеть от того, вводится ли текст с клавиатуры, считывается ли из файла или берется из другого источника. Процедуру, обрабатывающую ошибки – `Error`, снабдим параметром-строкой, которая будет содержать сообщение о характере ошибки. Запишем начало программы и ее основную часть:

```
program Parser;   { Распознаватель }
const
   EOT = #0;      { Признак "конец текста" }
type
   StrType = string[80];
var
   S  : StrType;  { Входная строка   }
   Ch : char;     { Очередной символ }
   i  : integer;  { Номер символа    }
.........
begin
   ResetText;     { Подготовить текст }
   Summa;         { Анализ суммы      }
   if Ch = EOT then
      WriteLn('Правильно')
   else
      Error('Ожидается конец текста');
end.
```

Записываем анализирующие процедуры, действуя по принципу пошаговой детализации:

```
procedure Summa; { Сумма }
begin
   if Ch in ['+','-'] then
      NextChar;
   Number;          { Число }
   while Ch in ['+','-'] do begin
      NextChar; Number;
   end;
end;

procedure Number;   { Число }
begin
   IntNumber;
   if Ch = '.' then begin
      NextChar;
      IntNumber;
   end;
   if Ch in ['e','E'] then begin
      NextChar;
      if Ch in ['+','-'] then
         NextChar;
      IntNumber;
   end;
end;

procedure IntNumber;   { Целое }
begin
   repeat
      if Ch in ['0'..'9'] then
         NextChar
      else
         Error('Ожидается цифра')
   until not ( Ch in ['0'..'9'] );
end;
```

В программе эти процедуры должны располагаться в обратном порядке (`IntNumber; Number; Summa`). Анализатор целого может выглядеть и по-другому:

```
procedure IntNumber;   { Целое }
begin
   if not (Ch in ['0'..'9']) then
      Error('Ожидается цифра');
```

```
   while Ch in ['0'..'9'] do
      NextChar;
end;
```

Теперь записываем процедуры, обеспечивающие взаимодействие анализатора с текстом.

```
procedure ResetText;  { Подготовить текст }
begin
   WriteLn('Введите сумму');
   ReadLn(S);
   i := 0;
   NextChar;
end;

procedure NextChar;   { Следующий символ }
begin
   i := i + 1;
   if i <= Length(S) then begin
      Ch := S[i]
   else
      Ch := EOT;
end;
```

В этих процедурах локализована вся специфика представления текста. Теперь, если источником текста будет, например, текстовый файл 'SUMMA.TXT', достаточно изменить лишь эти процедуры (`f : text` – глобальная переменная):

```
procedure ResetText;   { Подготовить текст }
begin
   Assign( f, 'SUMMA.TXT' );
   Reset(f);
   i := 0; NextChar;
end;

procedure NextChar;    { Следующий символ }
begin
   { Пропуск пустых строк }
      while not eof(f) and eoln(f) do
         ReadLn(f);
   if not eof(f) then begin
      Read( f, Ch ); Write(Ch);
      i := i + 1;
      end
   else begin
      Ch := EOT; WriteLn;
   end;
end;
```

Процедура `NextChar` может выполнять еще одну полезную функцию – пропуск пробелов в тексте. В этом случае пробелы могут использоваться в любом месте текста для улучшения его наглядности. Так, выражение 123.456 + 1.997 - 1E5, содержащее пробелы вокруг знаков плюс и минус, будет считаться правильным.

Для пропуска пробелов достаточно дополнить процедуру чтения очередного символа циклом, который прекращается только, когда символ – не пробел.

```
procedure NextChar;
   { Следующий символ }
begin
   repeat
      i := i + 1;
      if i <= Length(S) then
```

```
            Ch := S[i]
         else
            Ch := EOT;
      until Ch <> ' ';
   end;
```

Процедура, обеспечивающая реакцию на синтаксическую ошибку с выдачей сообщения о характере ошибки и указанием ошибочного символа, выглядит так:

```
procedure Error( Message : StrType );
   { Ошибка }
   { Message - сообщение об ошибке }
begin
   WriteLn;
   WriteLn( '^':i );
   Writeln('Синтаксическая ошибка: ', Message );
   Halt;
end;
```

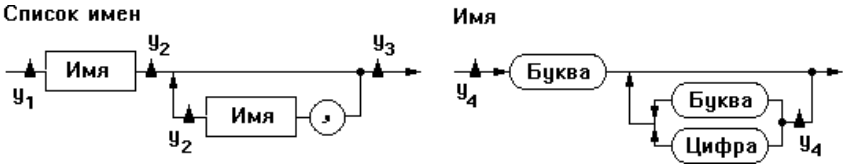
3. Синтаксически управляемая обработка текста

3.1. Включение действий в синтаксис. Семантические процедуры.

В процессе своей работы программа-распознаватель сопоставляет последовательность символов текста с синтаксическими диаграммами. В ходе анализа распознается структура текста, выделяются его составные части – нетерминальные и терминальные символы.
В свою очередь семантическая обработка сводится к выполнению определенного набора действий. Действия по смысловой обработке текста могут быть связаны с его структурой, нетерминальными и терминальными символами. Для этого достаточно обозначить на синтаксических диаграммах места, при прохождении которых в процессе анализа соответствующие действия должны быть выполнены.
Отдельные действия, выполняемые в ходе синтаксического анализа текста с целью его смысловой обработки, назовем *семантическими процедурами*.
Рассмотрим, каким образом может быть организована семантическая обработка на простом примере.

Задача. Текст представляет собой список разделенных запятой имен. Подсчитать количество элементов списка имен и напечатать их, разместив по одному имени в строке.

Синтаксис списка имен определим с помощью синтаксических диаграмм.



Обозначая семантические процедуры y_1, y_2, y_3, y_4, разместим на диаграмме соответствующие им треугольные значки в тех местах, при прохождении которых в ходе анализа должны выполняться эти процедуры.
Имея в виду условие задачи, предусмотрим, что содержание семантических процедур таково (`k` - счетчик количества имен; `Ch` - очередной символ текста):

```
y1: k := 0;
y2: k := k + 1; WriteLn;
y3: WriteLn('Количество имен ', k );
y4: Write( Ch );
```

Теперь запишем анализирующие процедуры для нетерминалов "Список имен" и "Имя", включив в них смысловую обработку. Поскольку действия y_1, ... , y_4 в нашей задаче очень просты, запишем их прямо в соответствующих местах текста анализирующих процедур. В случае, когда обработка более сложна, можно оформить семантические процедуры с помощью **procedure.**

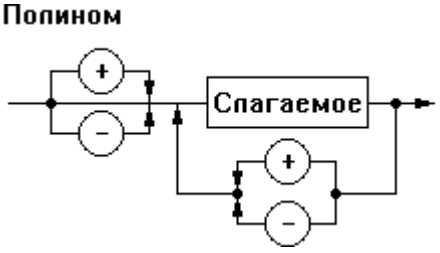
```
procedure NameList;
begin
   k := 0;                          { y1 }
   Name;
   k := k + 1; WriteLn;             { y2 }
   while Ch = ',' do begin
      NextChar; Name;
      k := k + 1; WriteLn;          { y2 }
   end;
   WriteLn('Количество имен ', k ); { y3 }
end;

procedure Name;
begin
   if Ch in ['A'..'Z','a'..'z'] then begin
      Write( Ch );                  { y4 }
      NextChar;
      end
   else
      Error('Ожидается буква');
   while Ch in ['A'..'Z','a'..'z','0'..'9'] do
   begin
      Write( Ch );                  { y4 }
      NextChar;
   end;
end;
```

3.2. Трансляция полинома

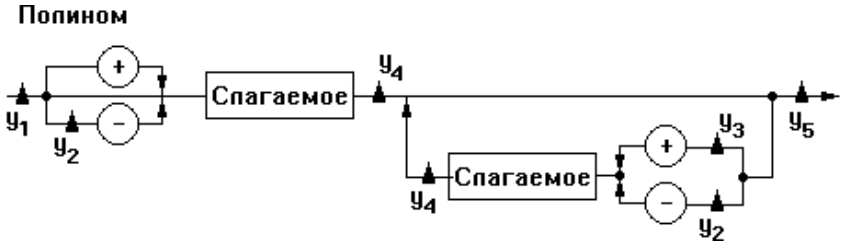
Вернемся к задаче перемножения полиномов, заданных в символьной форме (см.разд.1.1). Для ее решения осталось реализовать процедуру, которая выполнит преобразование (трансляцию) текстовой записи полинома в массив коэффициентов:

```
procedure Translate( S :StrType; var P :PolyType );
   { Транслятор полиномов }
```

Определим с помощью диаграмм синтаксис полинома, представив его как последовательность слагаемых, разделенных знаками плюс или минус. Знак может быть и перед первым слагаемым.



Для удобства записи анализатора на Паскале преобразуем диаграмму в эквивалентную, но содержащую цикл с предусловием.



На полученной диаграмме расставим обозначения семантических процедур. Предусмотрим семантические вычисления, обеспечивающие в ходе анализа полинома формирование массива его коэффициентов и определение порядка.
Для формирования массива коэффициентов каждое слагаемое полинома представим в виде

$$Sig*b*x^k, \quad \text{где}$$

`Sig` – знак очередного слагаемого (принимает значения +1 и -1);
`b` – абсолютная величина коэффициента;
`k` – показатель степени.

Примем, что процедура анализа и обработки слагаемого в качестве результата будет формировать b и k (переменные `Sig, b` и `k` будут описаны в процедуре `Translate,` а для распознающих процедур они будут глобальными). Формирование значения `Sig` для каждого слагаемого нужно обеспечить в ходе анализа нетерминала "Полином".
Перед началом обработки (семантическая процедура y_1) занесем нули в массив коэффициентов и величину, задающую порядок полинома (эти действия выполняются уже существующей процедурой `ClearPoly`). На случай, когда перед первым слагаемым не будет записан знак, примем `Sig=1`.

```
y1:  ClearPoly(P); Sig := 1;
```

Для формирования знака слагаемого, перед которым поставлен минус или плюс, предусмотрим

```
y2:  Sig := -1;
y3:  Sig := 1;
```

После прочтения очередного слагаемого, когда определены `Sig, b` и `k`, выполним семантическую процедуру y_4, которая обеспечит формирование `k`-го коэффициента в массиве коэффициентов полинома `P`. По окончании обработки полинома определим его степень `n` (y_5) .

```
y4:  a[k] := a[k] + Sig*b;
y5:  n := Nmax;
     while (n>0) and (a[n] = 0) do
        n := n-1;
```

Теперь, пользуясь синтаксической диаграммой "Полином", с учетом предусмотренных семантических процедур запишем распознающую процедуру.

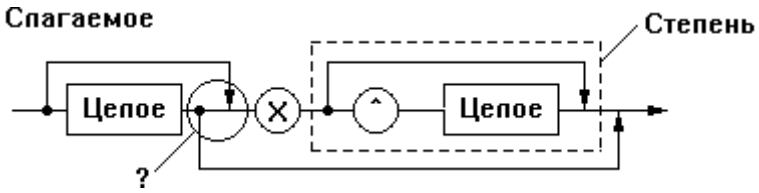
```
procedure Poly; { Полином }
begin
   with P do begin
      ClearPoly( P );                { y1 }
      Sig := 1;                      {    }
      if Ch in ['+','-'] then begin
```

```
         if Ch = '-' then
            Sig := -1;              { y2 }
         NextChar;
      end;
      Addend;                  { Слагаемое }
      a[k] := a[k] + Sig*b;          { y4 }
      while Ch in ['+','-'] do begin
         if Ch = '+' then
            Sig := 1                { y3 }
         else
            Sig := -1;              { y2 }
         NextChar;
         Addend;               { Слагаемое }
         a[k] := a[k] + Sig*b;      { y4 }
      end;
      n := Nmax;                     { y5 }
      while (n>0) and (a[n] = 0) do {     }
         n := n-1;                   {     }
   end;
end;
```

Процедура `Poly` и другие процедуры транслятора полиномов будут расположены внутри процедуры `Translate`, общая схема которой такова:

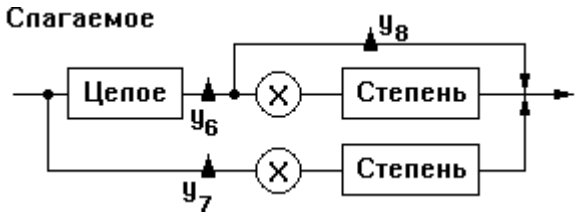
```
procedure Translate( S :StrType; var P :PolyType );
   { Транслятор полиномов }

const
   EOT = #0;
var
   i     : integer;  { номер символа         }
   Ch    : char;     { очередной символ      }
   Sig   : integer;  { знак слагаемого       }
   b     : integer;  { модуль коэффициента   }
   k     : integer;  { степень слагаемого    }
   Int   : integer;  { значение целого       }

   procedure Error( Message : StrType ); { Ошибка }
     ...

   procedure NextChar; { Читать следующий символ }
     ...

   { Распознающие процедуры }

     ...

begin
   i := 0;
   NextChar;
   Poly;
   if Ch <> EOT then
      Error('Ожидается конец текста');
end; { Translate }
```

Перейдем к разработке распознавателя слагаемого (процедура Addend). Построим синтаксическую диаграмму, предусмотрев все разрешенные варианты записи слагаемого.



Получившаяся диаграмма правильно отражает синтаксис слагаемого, но структура ее такова, что она не разделяется на типовые фрагменты, а это затруднит программирование анализатора. Неструктурированность диаграммы обусловлена фрагментом, который на рисунке обозначен знаком "?".

Преобразуем диаграмму в эквивалентную, но состоящую только из совокупности типовых структур. Для этого изобразим отдельно две ветви: одна соответствует слагаемому, начинающемуся с числа; другая – с буквы X. Фрагмент, выделенный на исходной диаграмме пунктирной рамкой, преобразуем в нетерминал "Степень".



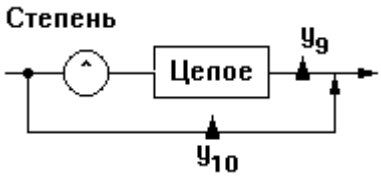
Семантическая обработка, которую нужно выполнить при анализе слагаемого, должна обеспечить вычисление коэффициента слагаемого – `b` и показателя степени `k`. В случае, когда в слагаемом присутствует X, определение `k` выполнит распознаватель нетерминала "Степень", а если X нет, то `k=0`. В соответствии с этим предусмотрим семантические процедуры, выполняемые в ходе анализа слагаемого:

```
y6:  b := Int;  { Int - значение целого }
y7:  b := 1;
y8:  k := 0;

procedure Addend;       { Слагаемое }
begin
   if Ch in ['0'..'9'] then begin
      IntNumber;
      b := Int;                { y6 }
      if Ch = 'X' then begin
         NextChar;
         Power;
         end
      else
         k := 0;               { y8 }
      end
   else if Ch = 'X' then begin
      b := 1;                  { y7 }
      NextChar;
      Power;
      end
   else
      Error('Ожидается число или ''X''');
end;
```

Реализация распознавателя понятия "Степень" не вызывает затруднений и выполняется по соответствующей синтаксической диаграмме, на которой предусмотрены семантические процедуры y_9 и y_{10}. Чтобы защитить программу от ошибки при обращении к массиву коэффициентов, в процедуре y_9 предусмотрен контроль величины показателя степени. Это

пример ситуации, когда ограничения, которые трудно выразить синтаксически, контролируются на этапе семантической обработки.

```
y9 :  if Int <= Nmax then
         k := Int
      else begin
         k := Nmax;
         Error('Слишком большая степень');
      end;

y10:  k := 1;
```

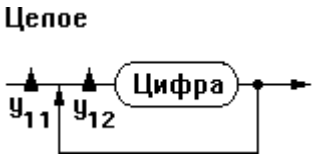


Обратите внимание, что в случае, когда семантическая процедура (y_{10}) расположена на ветви диаграммы, где не было терминальных и нетерминальных блоков, она сама играет роль блока, что несколько меняет структуру программы-распознавателя (в нашем случае появляется ветвь **`else`**).

```
procedure Power;          { Степень }
begin
   if Ch = '^' then begin
      NextChar;
      IntNumber;
      if Int <= Nmax then     { y9 }
         k := Int
      else begin
         k := Nmax;
         Error('Слишком большая степень');
      end;
      end
   else
      k := 1;                 { y10 }
end;
```

Распознаватель целого числа должен в ходе синтаксического анализа сформировать значение этого числа (переменная `Int`). Поскольку для представления чисел в нашей программе используется тип integer, нужно побеспокоиться о контроле возможного переполнения.

```
procedure IntNumber;              { Целое число }
var
   Digit : integer;
begin
   Int := 0;                                 { y11 }
   if not ( Ch in ['0'..'9'] ) then
      Error('Число начинается не с цифры');
   while Ch in ['0'..'9'] do begin
      Digit := ord(Ch) - ord('0');           { y12 }
      if  ( Maxint - Digit ) div 10 >= Int then
         Int := 10*Int + Digit               {     }
      else                                   {     }
         Error('Слишком большое число');     {     }
      NextChar;
   end;
end;
```



Транслятор полиномов готов, а с ним завершена и программа перемножения полиномов, с решения которой мы начали обсуждение синтаксически управляемых алгоритмов обработки текста. Полный текст этой программы приводится в приложении.

4. Обработка синтаксических ошибок

Во всех написанных нами программах обработка ошибок выполнялась с помощью процедуры `Error`, которая после выдачи сообщения *прекращала работу всей программы.* Такой способ реакции на ошибку упрощает синтаксический анализатор, но годится далеко не всегда – ведь завершать работу программы с выдачей диагностического сообщения – это ненамного лучше и удобней для пользователя, чем просто аварийное завершение программы. В связи с этим необходимо выработать такой способ реакции синтаксического анализатора на ошибку, чтобы работа программы не прерывалась, хотя процесс синтаксического анализа и обработки текста может быть прекращен.

Первый вариант усовершенствования синтаксического анализатора состоит в том, чтобы после вызова каждой распознающей процедуры проверять успешность ее завершения и, если при анализе нетерминала обнаружена ошибка, пропускать оставшиеся части вызывающей процедуры. В этом случае процедура Error уже не прерывает программу с помощью `Halt`, а формирует признак ошибки, который может быть проверен в распознающих процедурах. Такой подход, однако, сильно загромоздит программы анализатора, сделает их неудобочитаемыми.

Используем другое решение. Проблема при обработке ошибок состоит лишь в том, чтобы после обнаружения ошибки завершить *все процедуры*, цепочка вызова которых привела к процедуре, обнаружившей ошибку. Это можно сделать, если при обнаружении ошибки процедура `Error` установит признак ошибки, а текущему символу присвоит значение "конец текста". Поскольку этот символ не может встретиться в анализируемом тексте, он будет отвергнут всеми частями анализатора, который завершит свою работу без прерывания программы в целом. В качестве признака ошибки разумно использовать номер ошибочного символа (обозначим `ErrPos`), ненулевое значение которого соответствует наличию ошибки. Перед началом работы анализатора нужно выполнить `ErrPos := 0`. Исправленные процедуры `Error` и `NextChar` теперь выглядят так:

```
procedure Error( Message : StrType ); { Ошибка }
begin
   if ErrPos = 0 then begin
      ...
      ErrPos := i;
      Ch := EOT;
   end;
end;

procedure NextChar; { Читать следующий символ }
begin
   if ErrPos <> 0 then
      Ch := EOT
   else
      ...
end;
```

5. Интерпретация арифметических выражений

Одной из самых интересных и важных задач, которые могут быть решены с помощью обсуждаемых нами методов, является задача вычисления арифметического выражения по его текстовой записи. Мы обсудим и решим ее в такой постановке:

Задача. Разработать процедуру, которая по тексту выражения, записанного по правилам языка Паскаль и переданного процедуре в качестве входного параметра-строки, вычисляет значение этого выражения или сообщает об ошибке в его записи. Выражение может содержать вещественные числа, знаки операций +, –, *, /, стандартные функции и круглые скобки.

Назовем разрабатываемую процедуру `Calc` (от Calculator – вычислитель). Параметрами процедуры будут:

- `S` – входная строка;
- `V` – результат вычисления выражения;
- `ErrPos` (Error Position – позиция ошибки) – выходной параметр целого типа, который будет служить индикатором синтаксической ошибки. При отсутствии ошибки возвращается `ErrPos=0`, при наличии – значение `ErrPos` должно быть равно номеру ошибочного символа;
- `ErrMess` (Error Message – сообщение об ошибке) – строка, содержащая текст сообщения об ошибке, если `ErrPos<>0`.

```
procedure Calc(     S      : string;
                var V      : real;
                var ErrPos : integer;
                var ErrMess: string
               );
```

Вот пример программы, которая использует процедуру Calc и позволяет выполнять вычисления по формулам:

```
program Calculator;
{ Формульный калькулятор }
var
   x      : real;      { Значение выражения    }
   S      : string;    { Текст выражения       }
   ErrPos: integer;    { Позиция ошибки        }
   Mess   : string;    { Сообщение об ошибке   }

   procedure Calc(     S      : string;
                   var V      : real;
                   var ErrPos : integer;
                   var ErrMess: string
                  );
   ........
   end; { Calc }

begin     { Основная программа }
   WriteLn;
   WriteLn('ФОРМУЛЬНЫЙ КАЛЬКУЛЯТОР');
   repeat
      Write('?');
      Readln(S);
      Calc( S, x, ErrPos, Mess );
      if ErrPos > 0 then begin
         WriteLn('^':ErrPos+1);
         WriteLn( Mess );
         end
      else if abs(x) > 0.0001 then
         WriteLn( x:15:6 )
      else
         WriteLn( x );
   until S = '';
end.
```

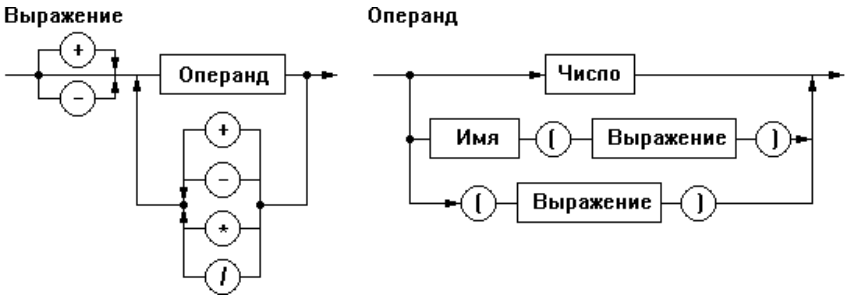
Ниже приводится фрагмент диалога с этой программой (курсивом выделены строки, введенные пользователем).

```
ФОРМУЛЬНЫЙ КАЛЬКУЛЯТОР
?12.345e6
12345000.000000
?2 + 2
       4.000000
?sin(2*arctan(1)/3) - 1/2
-9.0949470177E-13
?log(2)
    ^
Неправильное имя функции
?(((2+3)-7/3)))
              ^
Ожидается конец выражения
?sqrt( sqr(2.123) - 4*5*2.5 )
                            ^
Корень из отрицательного числа
?
 ^
Ожидается число, функция или '('
```

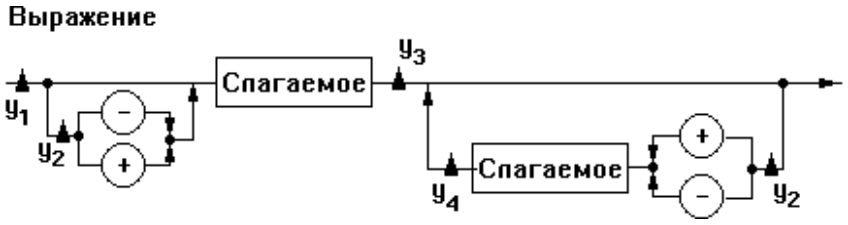
Диалог заканчивается вводом пустой строки, к которой и относится последнее сообщение.

5.1. Синтаксис арифметических выражений

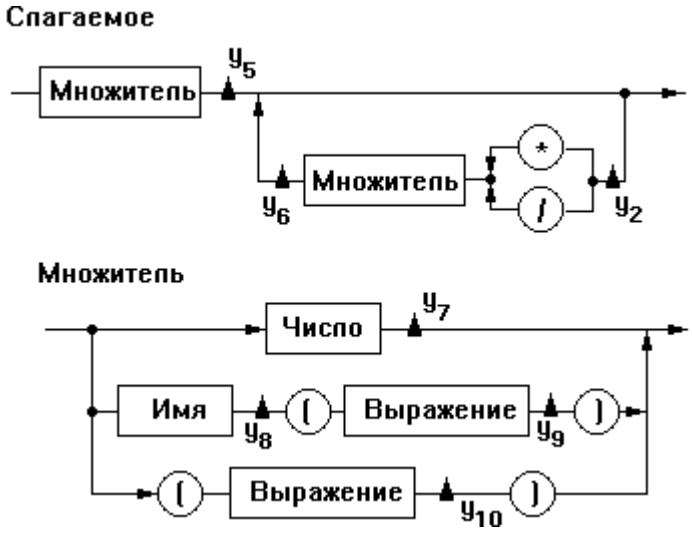
Определим синтаксис арифметического выражения с помощью диаграмм. Выражение состоит из операндов, которые разделены знаками +, -, *, /. Перед первым операндом могут быть записаны знаки + или -. Под операндами подразумеваются числа, функции и выражения в круглых скобках. Заметим сразу же, что такое определение рекурсивно: выражение состоит из операндов, которые в свою очередь могут содержать выражения (в скобках). В соответствии со сказанным изобразим диаграммы.



Приведенные диаграммы позволяют построить синтаксический анализатор, но использовать этот анализатор для вычисления выражения оказывается затруднительно. Дело в том, что при таком определении синтаксиса все знаки операций оказываются равноправны. При работе такого анализатора с встроенными в него семантическими процедурами операнды будут вычисляться и операции выполняться (при отсутствии скобок) слева направо, что не соответствует обычному порядку вычисления выражений, когда в первую очередь выполняются умножение и деление, а затем – сложение и вычитание.



В связи с этим синтаксис выражений следует определить по-другому, используя вспомогательные понятия-нетерминалы "Слагаемое" и "Множитель".



Такое задание синтаксиса позволит правильно вычислять выражения с учетом приоритета операций. Поясним это на примере вычисления выражения 1 + 2*3. Выполнение операции сложения происходит после распознавания слагаемых, а операции умножения – после распознавания множителей (семантические процедуры y_4 и y_6). При интерпретации выражения вначале будет обработано первое слагаемое (1), затем после распознавания (и запоминания) знака + произойдет повторный вызов процедуры "Слагаемое", в ходе работы которой будет вычислена величина этого слагаемого, состоящего из двух множителей (2*3), и в последнюю очередь семантической процедурой y_4 будет выполнена операция сложения.

5.2. Разработка интерпретатора

Сформулируем содержание семантических процедур, расположение которых показано на диаграммах. Ключевой принцип, которого будем придерживаться при интерпретации выражения, таков: результатом обработки каждого нетерминала – множителя, слагаемого, выражения должно быть значение этого нетерминала. В соответствии с этим снабдим распознающие процедуры "Выражение" (Expression), "Слагаемое" (`Addend`) и "Множитель" (`Multiplier`) выходным параметром-переменной (**`var`** `V: real`), которому в результате выполнения процедур должно быть присвоено значение соответствующего подвыражения. Обозначим также: `Op:char` – знак операции; `X : real` – вспомогательная переменная. Теперь семантические процедуры, имеющиеся на диаграмме "Выражение", реализуются так:

```
y1:  Op := '+';
y2:  Op := Ch;
y3:  V := значение слагаемого (первого);
       if Op = '-' then V := -V;
y4:  X := значение слагаемого;
```

```
     if Op = '+' then V := V + X
     else V := V - X;
```

Запишем распознающую процедуру-интерпретатор выражения.

```
procedure Expression( var V: real );{ Выражение }
var
   Op    : char;     { Знак операции      }
   X     : real;     { Значение операнда  }
begin
   Op := '+';
   if Ch in ['+','-'] then begin
      Op := Ch;
      NextChar;
   end;
   Addend(V);
   if Op = '-' then
      V := -V;
   while Ch in ['+','-'] do begin
      Op := Ch;
      NextChar;
      Addend(X);
      if Op = '+' then
         V := V + X
      else
         V := V - X;
   end;
end;
```

Величины `V, Op` и `X` должны быть непременно локальными для `Expression`. Дело в том, что в процессе работы анализатора при наличии в выражении скобок произойдет косвенный рекурсивный вызов `Expression`. В этом случае правильная работа интерпретатора будет обеспечена только в случае, если каждому вхождению в процедуру `Expression` будут соответствовать свои экземпляры `V, Op` и `X`. Рассмотрим ход вычисления выражения 1 - (2+3), представив, что `Op` – глобальная переменная. После прочтения первого слагаемого `Op` будет присвоено значение '–'. Но выполняться вычитание должно только *после* обработки выражения в скобках, в ходе которого `Op` примет значение '+'. Вместо исходного выражения будет вычислено 1+(2+3). Мы видим, что в случае, когда существует лишь один экземпляр `Op`, вычисления происходят неверно.
Действия, которые нужно выполнить при обработке слагаемого, подобны тем, что предусмотрены при анализе выражения, и задаются так:

```
y2:  Op := Ch;
y5:  V := значение множителя(первого);
y6:  X := значение множителя;
     if Op = '*' then
        V := V*X
     else if X <> 0 then
        V := V/X
     else
        Error('Деление на ноль');
```

Процедура-интерпретатор слагаемого записывается по соответствующей диаграмме с учетом семантических процедур.

```
procedure Addend( var V : real );
   { Слагаемое }
var
   Op    : char;     { Знак операции      }
   X     : real;     { Значение операнда  }
begin
   Multiplier( V );
   while Ch in ['*','/'] do begin
      Op := Ch;
      NextChar;
      Multiplier(X);
      if Op = '*' then
         V := V*X
      else if X <> 0 then
         V := V/X
      else
         Error('Деление на ноль');
   end;
end;
```

Некоторые особенности имеют распознавание и интерпретация множителя, в роли которого в нашем случае выступают первичные арифметические выражения – число, функция и выражение в скобках. Семантические вычисления, связанные с получением значения числа и выражения в скобках, несложны.

$У_7$: V := *значение числа;*

$У_{10}$: V := *значение выражения в скобках;*

Для распознавания и вычисления стандартных функций определим в программе перечислимый тип "функции" (Functions), построенный из идентификаторов всех функций, а также массив имен стандартных функций[3] (FuncNames). Определим также константы (типа Functions), соответствующие первой и последней функциям в этих списках. Обратите внимание, что в конец перечня включена фиктивная функция DUMMY (ее имя – пустая строка), наличие которой упростит обработку списка.

```
type
   Functions = ( ABSOLUTE, SQUARE, TRUNCATE,
                 ROUNDING, SINUS, COSINUS, ARC,
                 LOG, EXPONENT, SQUAREROOT, DUMMY );
const
   FirstFunc = ABSOLUTE;     { первая функция     }
   LastFunc  = SQUAREROOT;   { последняя функция  }

   FuncNames : array[Functions] of string[6] =
      ( 'ABS', 'SQR', 'TRUNC', 'ROUND', 'SIN',
        'COS', 'ARCTAN', 'LN', 'EXP', 'SQRT', ''
      );
```

Интерпретация функции выполняется в два этапа. Вначале путем сопоставления имени со списком имен стандартных функций[4] определяется идентификатор функции, а если имя в списке не найдено, – выдается сообщение об ошибке. Эту работу должен выполнить

---

[3] Здесь используются расширенные возможности языка, предусмотренные в Турбо Паскале.

[4] Для упрощения сравнения имен предусмотрим, что процедура NextChar все строчные буквы будет преобразовывать в соответствующие заглавные.

распознаватель имени. Семантическая процедура y_8 запоминает в локальной переменной F идентификатор функции, выданный в качестве результата распознавателем имени.

```
y8 : F := идентификатор функции;
```

Вычисление функции происходит после распознавания и вычисления ее аргумента:

```
y9 :V := значение выражения-аргумента;
     V := Func(F,V);
```

Здесь Func – функция, которая по идентификатору F и значению аргумента выдает значение соответствующей стандартной функции (или вызывает Error, если значение аргумента некорректно).

```
procedure Multiplier( var V : real );
   { Множитель }
var
   F : Functions;
begin
   if Ch in ['0'..'9'] then
      Number(V)
   else if Ch in ['A'..'Z'] then begin
      Name( F );
      if Ch <> '(' then
         Error('Ожидается ''('')
      else begin
         NextChar;
         Expression(V);
         V := Func( F, V );
         if Ch = ')' then
            NextChar
         else
            Error('Ожидается '')''')
         end
      end
   else if Ch = '(' then begin
      NextChar;
      Expression(V);
      if Ch <> ')' then
         Error('Ожидается '')''')
      else
         NextChar;
      end
   else
      Error('Ожидается число, функция или ''('');
end;
```

Процедура-распознаватель имени Name, считывая символы имени, формирует из них строку NameStr, которая затем сравнивается с именами стандартных функций.

```
   procedure Name( var F: Functions ); {Имя функции}
   var
      NameStr : string;
   begin
      if Ch in ['A'..'Z'] then begin
         NameStr := Ch;
         NextChar;
         end
      else
```

```
      Error('Ожидается буква');
   while Ch in ['A'..'Z','0'..'9'] do begin
      NameStr := NameStr + Ch;
      NextChar;
   end;

   { Определение функции по имени }

      F := FirstFunc;
      while ( F <= LastFunc ) and
            ( NameStr <> FuncNames[F] )
      do
         F := succ(F);
      if F=DUMMY then
         Error('Неправильное имя функции');
end;
```

Вычисление стандартной функции по ее идентификатору и значению аргумента реализуется несложно.

```
function Func( F : Functions; x : real ) : real;
   { Значение F(x) }
begin
   Func := x;
   case F of
   ABSOLUTE    : Func := abs(x);
   SQUARE      : Func := sqr(x);
   TRUNCATE    : Func := trunc(x);
   ROUNDING    : Func := round(x);
   SINUS       : Func := sin(x);
   COSINUS     : Func := cos(x);
   ARC         : Func := arctan(x);
   LOG:
      if x > 0 then
        Func := ln(x)
      else
        Error('Логарифм неположительного числа');
   EXPONENT    : Func := exp(x);
   SQUAREROOT  :
      if x >= 0 then
         Func := sqrt(x)
      else
         Error('Корень из отрицательного числа');
   end;
end;
```

Теперь осталось лишь обеспечить получение значения числа по его записи. Синтаксические диаграммы понятия "Число" приводились выше в разделах "Синтаксические диаграммы" и "Построение синтаксического анализатора по синтаксическим диаграммам". В последнем разделе был разработан и синтаксический анализатор. Для использования анализатора числа в интерпретаторе выражений достаточно включить в него семантические процедуры. Эти процедуры должны обеспечить вычисление целой и дробной частей мантиссы и вычисление порядка числа. Затем необходимо по полученным значениям мантиссы и порядка сформировать само число. Хотя эти вычисления несложны, но при их выполнении должен быть обеспечен точный учет ограничений в представлении вещественных чисел на конкретном компьютере, что приводит к тому, что необходимые процедуры становятся довольно громоздки. Корректное решение задачи получения вещественного числа по его записи можно найти в книге К.Йенсен и Н.Вирта[1].

Мы используем упрощенный подход, основанный на использовании имеющейся в Турбо Паскале стандартной процедуры `Val`, которая преобразует строку, содержащую запись

числа, в вещественное значение. Задача же нашей программы в этом случае состоит лишь в формировании входной строки для Val. Взяв за основу имеющийся анализатор числа, добавим в него такие действия: обозначим формируемую строку NumStr; присвоим ей вначале пустое значение; перед каждым чтением следующего символа (вызовом NextChar) будем приписывать очередной символ к формируемой строке: NumStr := NumStr + Ch; по окончании формирования строки преобразуем строку в число V с помощью процедуры Val, проанализировав возвращаемое ею значение номера ошибочного символа[5]. Используемую распознавателем числа процедуру IntNumber (целое число) поместим внутрь процедуры Number, а семантическая обработка при выполнении IntNumber будет сводиться в приписывании каждой распознанной цифры к строке NumStr.

```
procedure Number( var V : real );
   { Число }
var
   NumStr : string;  { Текст числа              }
   Err    : integer; { Номер неверного символа  }

procedure IntNumber;
   { Целое }
begin
   if not ( Ch in ['0'..'9'] ) then
      Error('Число начинается не с цифры');
   while Ch in ['0'..'9'] do begin
      NumStr := NumStr + Ch;
      NextChar;
   end;
end;

begin
   NumStr := '';
   IntNumber;
   if Ch = '.' then begin
      NumStr := NumStr + Ch;
      NextChar;
      IntNumber;
   end;
   if Ch = 'E' then begin
      NumStr := NumStr + Ch;
      NextChar;
      if Ch in ['+','-'] then begin
         NumStr := NumStr + Ch;
         NextChar;
      end;
      IntNumber;
   end;

   {Вычисление вещественного числа по его записи}
   {Val - стандартная процедура Турбо Паскаля   }

     Val( NumStr, V, Err );
     if Err <> 0 then
        Error('Ошибка в числе');

end; { Число }
```

[5] К сожалению, встроенная процедура Val в Турбо Паскале (версия 6.0) работает не всегда корректно. Так, при включенной эмуляции сопроцессора она, например, не в состоянии правильно обработать строку, состоящую из 40 цифр.

Интерпретатор арифметических выражений почти готов. Осталось лишь добавить процедуры `NextChar`, `Error` и раздел операторов процедуры `Calc`. Полный текст получающейся после этого программы приведен в приложении. При знакомстве с этим текстом обратите внимание на порядок расположения распознающих процедур и опережающее описание процедуры `Expression`. Чем вызвана необходимость такого описания?
Разработанный интерпретатор можно очень просто использовать в различных прикладных программах. В любом месте, где выполняется ввод числовых значений, можно использовать процедуру `Calc`, что предоставит пользователю возможность везде, где разрешается ввести число, вводить и выражение.

6. Рекурсивный спуск

В ходе предыдущего рассмотрения мы постепенно сформулировали и использовали достаточно общий подход к решению определенного класса задач обработки текста.
Примененный нами алгоритм синтаксического анализа носит название *рекурсивный спуск.* Такое название обусловлено тем, что в ходе работы анализатора, представляющего собой совокупность распознающих процедур, первой вступает в действие процедура, соответствующая анализируемому понятию в целом (начальному нетерминалу). Затем по мере необходимости вызываются распознаватели нижнего уровня, выполняющие анализ отдельных элементов анализируемого понятия. Процесс развивается от общего к частному, сверху вниз. Поэтому в названии алгоритма есть слово спуск. Как видно из примера интерпретатора формул, вызовы процедур могут быть рекурсивными, что в свою очередь определяется рекурсивной природой анализируемых понятий. (Рекурсия возникает при наличии "вложенности" понятий – внутри выражения может быть выражение, оператор языка программирования может содержать другие операторы и т.д.). Следовательно – *рекурсивный* спуск.
Метод рекурсивного спуска широко используется в практике разработки компиляторов и интерпретаторов языков программирования, что обусловлено его простотой и эффективностью. Рекурсивный спуск хорошо сочетается с общепризнанным подходом к разработке программ – методом пошаговой детализации, который мы использовали в данном пособии.
Вместе с тем, не следует думать, что рекурсивный спуск – это единственный метод синтаксического анализа. Существует большое количество других алгоритмов. Рекурсивный спуск – это лишь один из класса алгоритмов нисходящего разбора.
Может возникнуть вопрос и о пределах применимости метода рекурсивного спуска. Для распознавания какого класса языков он применим? Из предыдущего рассмотрения ясно, что речь идет лишь о *формальных языках*, то есть таких, правила записи текстов на которых могут быть формализованы, например, в виде синтаксических диаграмм. Не для каждого формального языка можно построить анализатор методом рекурсивного спуска. Синтаксис языка должен удовлетворять определенным ограничениям, о некоторых из которых говорилось выше (требование детерминированного распознавания).
Вопросы классификации формальных языков, способов описания и преобразования их синтаксиса, применимости различных алгоритмов распознавания выходят за рамки настоящего пособия. С этими вопросами можно ознакомиться по предлагаемой литературе.

7. Литература

Программирование на языке Паскаль

1. Й е н с е н К., В и р т Н. Паскаль. Руководство для пользователя и описание языка / Пер. с англ., предисл. и послесл. Д.Б.Подшивалова. - М.: Финансы и статистика, 1982. - 151с.: ил.

   Авторское описание языка Паскаль. Книга содержит полное описание синтаксиса языка с помощью синтаксических диаграмм и формул Бэкуса – Наура (БНФ). В качестве примера рекурсивного использования процедур рассматривается задача трансляции арифметического выражения в обратную польскую запись. Приводится программа преобразования записи числа в вещественное значение, учитывающая машинно-зависимые ограничения. Лучший учебник по языку Паскаль.

2. Й е н с е н К., В и р т Н. Паскаль: руководство для пользователя / Пер. с англ. и предисл. Д.Б.Подшивалова. - М.: Финансы и статистика, 1989. - 255с.: ил.

   Перевод третьего английского издания предыдущей книги. Описание синтаксиса Паскаля дано с помощью диаграмм и расширенной формы Бэкуса – Наура (РБНФ). Приводится краткое описание РБНФ.

3. Г р о г о н о П. Программирование на языке Паскаль: Пер с англ.-М.: Мир, 1982. - 384 с., ил.

   Хороший учебник по Паскалю. В качестве примера рассматривается программа-калькулятор, использующая метод рекурсивного спуска для вычисления выражений.

4. Д а г е н е В.А. и др. 100 задач по программированию: Кн. для учащихся: Пер. с лит. / В.А.Дагене, Г.К.Григас, К.Ф.Аугутис. – М.: Просвещение, 1993. – 255 с., ил.

   Книга содержит хорошо подобранные задачи по программированию с решениями на Паскале. Задачи 99 и 98 посвящены синтаксическому и лексическому анализу.

5. Ф а р о н о в В.В. Программирование на персональных ЭВМ в среде Турбо-Паскаль.- М.: Высшая школа, 1991. - 160с.

   Книга содержит необходимые сведения по программированию в системе Турбо Паскаль, включая те отличия языка системы от стандартного Паскаля, которые использовались в настоящем пособии. Существуют и более поздние ее издания.

Общие вопросы программирования

6. В и р т Н. Алгоритмы + структуры данных = программы: Пер. с англ. - М.: Мир, 1985. - 405 с.

   Рассматривается широкий круг общих вопросов программирования, методов организации данных, алгоритмов поиска и сортировки. Отдельная глава посвящена разработке трансляторов методом рекурсивного спуска. Программы записываются на Паскале.

   .

7. В ь ю к о в а Н.И., Г а л а т е н к о В.А., Х о д у л е в А.В. Систематический подход к программированию / Под ред. Ю.М.Баяковского - М.: Наука. Гл.ред. физ.-мат. лит. 1988. - 208с. - (Библиотечка программиста).

   Рассматривается программирование на Паскале с упором на систематический подход, основанный на пошаговой детализации. В качестве примера обсуждается задача трансляции арифметических выражений.

8. З е л к о в и ц М.,Ш о у А., Г э н н о н Дж. Принципы разработки программного обеспечения: Пер. с англ. - М.: Мир, 1982 - 368с. ил.

   Обсуждается методология разработки больших систем программного обеспечения: этапы, управление разработкой, структурное программирование, тестирование. Отдельная глава посвящена разработке компиляторов. Рассматриваются все этапы создания компилятора с упрощенного языка программирования. Программы записываются на языке ПЛ/1.

Разработка трансляторов

9. Г р и с Д. Конструирование компиляторов для цифровых вычислительных машин. Пер. с англ. - М.: Мир, 1975 - 544с.

   Классическая монография крупнейшего американского специалиста в области компиляторов. Рассматривается весь комплекс вопросов, относящихся к построению трансляторов.

10. Х а н т е р Р. Проектирование и конструирование компиляторов / Пер. с англ. - М.: Финансы и статистика, 1984. - 232с., ил.

Очень хорошая книга, в которой рассматриваются современные методы разработки трансляторов. Обсуждаются все этапы построения транслятора, вопросы теории формальных языков, различные алгоритмы распознавания. Изложение ведется с использованием языка Алгол-68.

11. К а р п о в Ю.Г. Основы построения компиляторов. Учебное пособие. - Л., изд. ЛПИ, 1982. - 79с.

Чрезвычайно ценное пособие, содержащее лаконичное и содержательное изложение широкого круга вопросов, относящихся к трансляции – от введения в теорию формальных грамматик до практических рекомендаций по разработке интерпретатора и компилятора языка программирования.

12. Р е й у о р д - С м и т В. Д ж. Теория формальных языков. Вводный курс: Пер. с англ. - М.: Радио и связь, 1988. - 128с.: ил.

В весьма удачной и вполне доступной форме излагаются теоретические основы построения трансляторов – теория формальных языков и грамматик.

13. Б е р р и Р., М и к и н з Б. Язык Си: введение для программистов / Пер. с англ. и предисл. Д.Б.Подшивалова. - М.: Финансы и статистика, 1988. - 191с.: ил.

Книга интересна тем, что в ней приведен полный текст на языке Си компилятора с подмножества языка Си. Книга и содержащиеся в ней программы написаны в хорошем стиле. Представляют интерес сведения о программе оценки стиля программ. Приводятся синтаксические диаграммы языка Си.

8. Приложения

8.1. Темы для самостоятельной работы

Ниже приводится перечень задач, которые могут быть решены с использованием обсуждавшихся в настоящем пособии методов. Для решения некоторых задач достаточно сведений, изложенных в пособии, другие требуют более глубокого знакомства с вопросами построения трансляторов.

1. Доработайте интерпретатор выражений, чтобы его удобнее было использовать в разнообразных прикладных программах. Расширьте список стандартных функций. Предусмотрите названия функций, совпадающие с общепринятыми. Предусмотрите операцию возведения в степень.
2. Разработайте интерпретатор функций одной переменной, который по выражению, содержащему имя переменной и заданному значению этой переменной, может вычислить значение выражения.
3. Разработайте компилятор формул, который преобразует текст выражения, содержащего имя переменной, в некоторое внутреннее представление, которое в дальнейшем может эффективно использоваться для вычисления выражения при различных значениях переменной. В качестве внутреннего представления можно выбрать, например, обратную польскую запись [1, 6, 8, 9, 10, 11]. Разработайте интерпретатор внутреннего представления, позволяющий вычислять выражение.
4. Пользуясь компилятором и интерпретатором из предыдущей задачи, разработайте программу для построения графиков функций одной переменной. Выражение для функции вводится в режиме диалога.
5. Разработайте компилятор, который преобразует выражение, состоящее из целых чисел, имен целочисленных переменных, знаков операций и скобок, в текст программы на языке ассемблера.
6. Разработайте интерпретатор и компилятор для функций двух переменных.
7. Разработайте программу построения на экране поверхностей для функции двух переменных. Выражение функции вводится в режиме диалога.

8. Даны два выражения, содержащие числа, функции, скобки, имена определенных переменных. Напишите модуль для контролирующей программы, позволяющий установить эквивалентность этих выражений. Предполагается, что одно из выражений является ответом учащегося на вопрос контролирующей программы, а другое – правильным эталонным ответом.
9. Напишите компилятор, преобразующий запись арифметического выражения в последовательность нажатий клавиш вашего любимого калькулятора.
10. Напишите программу, выполняющую символьное дифференцирование. По записи выражения, содержащего имена переменных, программа должна формировать запись производной исходного выражения по указанной переменной.
11. Синтаксический анализатор РБНФ. Эквивалентной по отношению к синтаксическим диаграммам формой задания синтаксиса является Расширенная Бэкуса – Наура Форма (РБНФ) [2]. Преимущество описания синтаксиса с помощью РБНФ состоит в том, что это текст, который удобно обрабатывать с помощью программы. Определите синтаксис РБНФ с помощью синтаксических диаграмм и с помощью самой РБНФ. Разработайте синтаксический анализатор РБНФ. Проверьте его работу на примере определения синтаксиса РБНФ с помощью РБНФ.
12. Универсальный анализатор регулярных языков. Будем называть регулярным выражение на РБНФ, не содержащее нетерминалов. Напишите программу, которая по тексту регулярного РБНФ-выражения и входному тексту проверяет соответствие входного текста синтаксису, заданному выражением. Полученное решение может использоваться, например, при создании контролирующих программ.
13. Генератор синтаксических анализаторов. Синтаксис некоторого понятия задан с помощью РБНФ. Напишите программу, которая, обрабатывая РБНФ, создает текст (на Паскале) программы-анализатора для этого понятия. (Упрощенный вариант: синтаксис задается регулярным РБНФ-выражением.)
14. Универсальный синтаксический анализатор. Синтаксис некоторого понятия задан с помощью РБНФ. Разработайте программу, которая по тексту РБНФ и входному тексту проверяет правильность входного текста.
15. Разработайте синтаксический анализатор для школьного алгоритмического языка. При решении этой задачи следует определить синтаксис языка на двух уровнях. На нижнем уровне язык определяется как последовательность лексем. Лексемы – это числа, служебные слова, имена, знаки. При определении собственно синтаксиса лексемы считаются терминальными символами. Для получения очередного терминального символа-лексемы синтаксический анализатор обращается к *лексическому анализатору – сканеру* [4, 8, 9, 10, 11].
16. Напишите компилятор, преобразующий алгоритмы, записанные на школьном алгоритмическом языке, в программы для программируемого калькулятора. При решении этой задачи имеет смысл сформировать подмножество алгоритмического языка, оставив лишь то, что соответствует возможностям калькулятора.
17. Разработайте интерпретатор школьного алгоритмического языка.
18. Напишите компилятор, преобразующий алгоритмы, записанные на школьном алгоритмическим языке, в программы на языке Паскаль.
19. Напишите компилятор, преобразующий алгоритмы, записанные на школьном алгоритмическим языке, в программы на языке Форт.
20. Разработайте интерпретатор языка Лого.
21. Разработайте синтаксический анализатор языка Паскаль. На базе такого анализатора могут быть построены различные инструментальные средства, которые будут полезны при разработке программ и обучении программированию на Паскале. Некоторые из них упоминаются ниже.
22. Разработайте программу, которая, обрабатывая текст программы на Паскале, модифицирует его, размещая с помощью отступов правильным образом в соответствии со

структурой программы. Предусмотрите возможность использования различных стилей размещения текста программы. Испытайте программу на ее собственном тексте.

23. Напишите программу для оценки стиля программ на Паскале. За основу можно взять оценки и подходы, предложенные в [13] для языка Си.

24. Напишите программу, которая строит дерево вызовов процедур для программы на Паскале, состоящей из нескольких файлов-модулей.

25. Напишите программу, которая позволяет получить таблицу имен для программы на Паскале, состоящей из нескольких модулей. Таблица должна содержать сведения о том, где (в каком модуле, процедуре, строке) определена (описана) каждая константа, тип, переменная, процедура, функция и где она используется. Следует предусмотреть формирование таблицы с разной степенью детализации.

26. Разработайте программу (программы), которая выполняет описанные в темах 22-25 действия для программ на языке Бейсик (Фортран, Си, ...).

27. Известно, что некоторые интерпретаторы Бейсика выполняют программу несколько быстрее, а программа занимает меньше памяти, если в ней меньше строк. Напишите программу, которая "сжимает" программу на Бейсике, удаляя комментарии и максимально объединяя строки.

8.2. Индивидуальные задания по синтаксическому анализу

Выполнение индивидуальных заданий предполагает разработку синтаксического анализатора предложенного в задании понятия. Программа-анализатор должна выполнять только проверку соответствия входного текста заданному синтаксису. В качестве результата ее работы выдается или сообщение об ошибке с указанием места обнаружения ошибки (ошибочного символа), или сообщается, что текст записан верно. Какой-либо семантической обработки не предусматривается.

Программа должна быть написана с соблюдением требований структурного программирования (разбиение на модули-подпрограммы, отступы для выделения структуры программы) и снабжена достаточным количеством комментариев.

Отчет по заданию готовится на компьютере.

8.2.1. Порядок выполнения индивидуального задания.

1. Постройте синтаксические диаграммы в соответствии с правилами записи анализируемого понятия.
2. Согласуйте построенные диаграммы с преподавателем.
3. Запишите программу-анализатор и проведите ее отладку.
4. Предъявите работающую программу преподавателю для испытания.
5. Подготовьте и напечатайте отчет по заданию.

8.2.2. Содержание отчета.

1. Титульный лист.
2. Формулировка задания.
3. Набор диаграмм для анализируемого понятия.
4. Текст программы анализатора на языке программирования.

8.2.3. Варианты заданий

Построить синтаксический анализатор понятия:

1. Арифметическое выражение. Элементами выражения являются имена переменных, вещественные числа, функции (названия функций произвольные), знаки арифметических операций, круглые скобки.
2. Сумма – последовательность натуральных чисел и имен, разделенных знаками плюс и минус. Возможен и знак перед первым слагаемым.
3. Сумма вещественных чисел в форме с фиксированной точкой.
4. Квадратное уравнение с целыми коэффициентами.
5. Линейное алгебраическое уравнение с N неизвестными (X_k, $k=1,2, \ldots N$) и постоянными целыми коэффициентами.
6. Сумма обыкновенных дробей.
7. Комплексное число (с целочисленными значениями действительной и мнимой частей).

8. Алгебраическая сумма – последовательность имен, разделенных знаками "+" и "-" с возможными знаками перед первым членом.
9. Линейное однородное дифференциальное уравнение с постоянными целочисленными коэффициентами.
10. Произведение вида $(X-A_1)(X-A_2)(X-A_3) \ldots (X-A_n)$, где A_i, ($i=1..n$) – целые числа.
11. Бесскобочное арифметическое выражение с целочисленными операндами (без функций).
12. Линейное уравнение с n неизвестными и постоянными целочисленными коэффициентами. Имена переменных произвольные.
13. Мультипликативная функция n переменных:

$$\Phi = A_0 \prod_{i=1}^{n} X_i ,$$

где A_0 – целая константа;
X_i – имена.

14. Отношение:

Операнд ⊗ *Операнд*,

где *Операнд* – целое или имя;
⊗ – знак отношения (>, <, =, <>, >=, <=).

15. Условие – отношения (см. выше), объединенные операциями "**и**" и "**или**" . Операнды – однобуквенные имена.
16. Сумма произведений – последовательность слагаемых, представляющих собой произведение нескольких операндов. Операнды – имена.
17. Последовательность записанных через запятую элементов двумерных массивов. Индексы – натуральные числа.
18. Серия команд присваивания, разделенных ";". В каждой команде слева записано имя, справа – натуральное число. Знак присваивания ":=".
19. Арифметическое выражение с натуральными операндами, знаками операций +, -, *, : и со скобками (без вложенности).
20. Сумма функций – последовательность функций с произвольными именами, – разделенных знаками "+" и "-". Аргументы функций – имена или натуральные числа в скобках.
21. Бесскобочное арифметическое выражение с функциями. Все операнды – однобуквенные имена. Аргументы функций записываются в скобках.
22. Оператор Write (элементы списка – имена, строки; параметры форматов - целые числа).
23. Оператор Read с элементами списка ввода – именами простых переменных и элементами линейных массивов. Индексы массивов – целые числа.
24. Переменная языка Паскаль (включая элементы массива, компоненты записей и динамические переменные. Индексы – целые числа).
25. Простой тип языка Паскаль (имя, перечислимый тип, ограниченный тип. Константы ограниченного типа – целые числа или символы).
26. Оператор **case** языка Паскаль. В роли выбирающего выражения – имя переменной, метки варианта – целые или символы, операторы – присваивания с целым правой части.
27. Оператор **with** языка Паскаль. В роли оператора после слова **do** – единственный оператор присваивания, в котором слева имя, справа – целое.
28. Раздел констант языка Паскаль. Предполагаются константы только целого и символьного типа.
29. Раздел меток языка Паскаль.
30. Список параметров процедур и функций языка Паскаль.
31. Числовой ряд вида *1 + 1/n1 + 1/n2 + ...*, где *n1, n2, ...* – натуральные числа.
32. Числовой ряд вида *1 + 1/(n1*m1) + 1/(n2*m2)+ ...*, где *n1, n2, ..., m1, m2, ...* - натуральные числа.
33. Числовой ряд вида *1 + 1/n1^2 + 1/n2^2 + ...,* где *n1, n2, ...* - натуральные числа.
34. Многочлен от x с рациональными коэффициентами.
35. Дробно-линейная функция от *x: (a1*x+b1)/(a2*x+b2),* где *a1, b1, a2, b2* – целые числа (при равенстве 0 могут не записываться).
36. Иррациональная сумма вида *x^(n1/m1)+x^(n2/m2)-...+...*, где *n1, n2, ..., m1, m2, ...* - натуральные числа.
37. Спецификация файла по правилам операционной системы.[6]
38. Команда операционной системы Copy.
39. Команда операционной системы Date.

---

[6] Сложность этого и нескольких последующих заданий зависит от операционной системы, применительно к которой они выполняются. Более простой синтаксис имеют спецификации файлов и команды в операционных системах CP/M, MSX DOS; несколько сложнее – в MS DOS.

40. Команда операционной системы Del
41. Команда операционной системы Dir
42. Последовательность векторов: *(x1,y1,z1), (x2,y2,z2)*, ..., где *x,y,z* - целочисленные координаты векторов.
43. Многочлен от *x: (x-x1)^k1*(x-x2)^k2*....,* где *x1, x2, ..* - целые числа; *k1, k2, ...* - неотрицательные целые.
44. Многочлен от *x*, записанный по схеме Горнера:

$$a_0 + x(a_1 + x(a_2 + x(\dots))),$$

где a_0, a_1, ... – целые числа.

45. Дробно-рациональная функция: *(x-a1)(x-a2).../((x-b1)(x-b2)...)*, где *a1, a2, ..., b1, b2, ...* - целые числа.
46. Уравнение плоскости вида *Ax+By+Cz+D=0* с целыми коэффициентами.
47. Уравнение плоскости в отрезках: *x/a+y/b+z/c=1*, где *a, b, c* - ненулевые целые числа.
48. Линейная функция двух переменных (*x* и *y*) с вещественными коэффициентами.
49. Множество-константа языка Паскаль с базовым типом char;
50. Множество-константа языка Паскаль с базовым типом integer.

8.2.4. Пример выполнения индивидуального задания

Вологодский государственный педагогический институт
Кафедра информатики

Индивидуальное задание по синтаксическому анализу текста

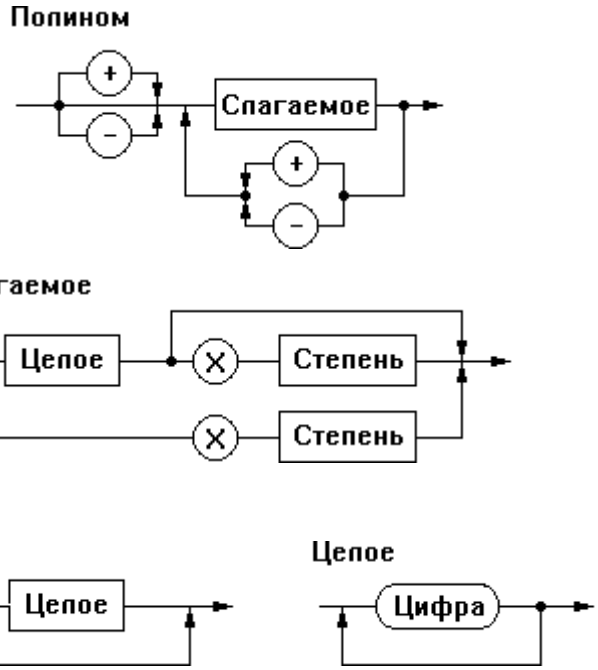
Студент
Курс
Группа

Преподаватель

Вологда
1994

Задание. Построить синтаксический анализатор понятия: многочлен от X с целочисленными коэффициентами.

Синтаксические диаграммы



Программа на Паскале

```
program PolynomTest;
const
   EOT    = #0;       { Признак "конец текста"   }
type
   StrType = string[80];   { Тип входной строки }
var
   Str    : StrType;        { Входная строка     }
   Ch     : char;           { Очередной символ   }
   i      : integer;        { Номер символа      }
   ErrPos: integer;         { Позиция ошибки     }

procedure Error( Message : StrType ); { Ошибка  }
   { Message - сообщение }
begin
   if ErrPos = 0 then begin
      WriteLn( '^' : i+1 );
      WriteLn('Синтаксическая ошибка: ',Message );
      ErrPos := i;
      Ch := EOT;
   end;
end;

procedure NextChar;   { Читать следующий символ }
begin
   if ErrPos <> 0 then
      Ch := EOT
   else
      repeat
         i := i + 1;
         if i<=length(Str) then
            Ch := UpCase(Str[i])
```

```
         else
            Ch := EOT;
      until Ch <> ' ';
end;

procedure Number; { Целое число }
begin
   if not ( Ch in ['0'..'9'] ) then
      Error('Число начинается не с цифры')
   else
      NextChar;
   while Ch in ['0'..'9'] do
      NextChar;
end;

procedure Power; { Степень }
begin
   if Ch = '^' then begin
      NextChar; Number;
   end;
end;
procedure Addend; { Слагаемое }
begin
   if Ch = 'X' then begin
      NextChar;
      Power;
      end
   else begin
      Number;
      if Ch = 'X' then begin
         NextChar;
         Power;
      end;
   end;
end;

procedure Polynom; { Полином, многочлен }
begin
   if Ch in ['+','-'] then NextChar;
   Addend;
   while Ch in ['+','-'] do begin
      NextChar; Addend;
   end;
end;

procedure ResetText;
begin
   WriteLn(
  'Введите многочлен от X с целыми коэффициентами');
   Write('>'); ReadLn( Str );
   i := 0;
   NextChar;
end;

begin { Основная программа }
   repeat
      ErrPos := 0;
      ResetText;
      Polynom;
      if Ch <> EOT then
         Error('Ожидается конец текста');
      WriteLn;
   until FALSE;
end.
```

8.3. Программа перемножения полиномов

Приведенная ниже программа выполняет перемножение двух многочленов от *x* с целыми коэффициентами, записанных в символьной форме, и выводит произведение в символьной форме в порядке убывания степеней. (Суммарная степень полиномов не превышает 255. Длина записи каждого – не более 80 символов).

```
program PolyMult;
{ Перемножение полиномов  }
{ (c) С.Свердлов, 1989,94 }
const
   Nmax = 255; { Максимальный порядок полинома }
type
   StrType  = string[80];
   PolyType =
      record
         n  : integer;  { порядок полинома }
         a  : array [0..Nmax] of integer;
      end;
var
   P1, P2, Q: PolyType;{ сомножители и произведение}
   S        : StrType; { входная строка }
   ErrPos   : integer;

procedure ClearPoly( var P : PolyType ); { Очистка }
var
   i : integer;
begin
   for i := 0 to Nmax do
      P.a[i] := 0;
   P.n := 0;
end;

procedure MultPoly( var X, Y, Z : PolyType );
   { Z = X*Y }
var
   i, j  : integer;
begin
   ClearPoly( Z );   { "Обнуление" Z }
   for i := 0 to X.n do
      for j := 0 to Y.n do
         Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
   with Z do begin
      n := Nmax;
      while ( n > 0 ) and ( a[n] = 0 ) do
         n := n-1;
   end;
end;

procedure WritePoly( P : PolyType );
   { Вывод полинома }
var
   i : integer;
begin
   with P do
      for i := n downto 0 do begin
         if ( a[i]>0 ) and ( i<>n ) then
            write( '+' );
         if ( a[i]=-1 ) and ( i>0 ) then
            write( '-' );
         if ( abs(a[i])>1 ) or
            ( i= 0 ) and ( a[i] <> 0 ) or
            ( n=0 )
```

```
         then
            write( a[i] );
         if ( i>0 ) and ( a[i]<>0 ) then
            write( 'x' );
         if ( i>1 ) and ( a[i]<>0 ) then
            write( '^', i )
      end;
end;

procedure Translate(     S : StrType;
                     var P : PolyType;
                     var ErrPos: integer );
const
   EOT = #0;
var
   i     : integer;  { номер символа          }
   Ch    : char;     { очередной символ       }
   Sig   : integer;  { знак слагаемого        }
   b     : integer;  { модуль коэффициента    }
   k     : integer;  { степень слагаемого     }
   Int   : integer;  { значение целого        }

procedure Error( Message : StrType ); { Ошибка }
   { Message - сообщение }
begin
   if ErrPos = 0 then begin
      WriteLn( '^' : i+1 );
      WriteLn('Синтаксическая ошибка: ', Message);
      ErrPos := i;
      Ch := EOT;
   end;
end;


procedure NextChar;
{ Читать следующий символ }
begin
   if ErrPos <> 0 then
      Ch := EOT
   else
      repeat
         i := i + 1;
         if i<=length(S) then
            Ch := UpCase(S[i])
         else
            Ch := EOT;
      until Ch <> ' ';
end;

procedure IntNumber;          { Целое число }
var
   Digit : integer;
begin                                      { y10 }
   Int := 0;
   if not ( Ch in ['0'..'9'] ) then
      Error('Число начинается не с цифры');
   while Ch in ['0'..'9'] do begin
      Digit := ord(Ch) - ord('0');         { y11 }
      if  ( Maxint - Digit ) div 10 >= Int then
         Int := 10*Int + Digit             {     }
      else                                 {     }
         Error('Слишком большое число');   {     }
      NextChar;
   end;
```

```
end;

procedure Power;          { Степень }
begin
   if Ch = '^' then begin
      NextChar;
      IntNumber;
      if Int <= Nmax then
         k := Int             { y8 }
      else begin
         Error('Слишком большая степень');
         k := 0;
      end
      end
   else
      k := 1;                 { y9 }
end;

procedure Addend;       { Слагаемое }
begin
   if Ch in ['0'..'9'] then begin
      IntNumber;
      b := Int;               { y6 }
      if Ch = 'X' then begin
         NextChar;
         Power;
         end
      else
         k := 0;              { y8 }
      end
   else if Ch = 'X' then begin
      b := 1;                 { y7 }
      NextChar;
      Power;
      end
   else
      Error('Ожидается число или ''X''');
end;

procedure Poly; { Полином }
begin
   with P do begin
      ClearPoly( P );               { y1 }
      Sig := 1;                     {    }
      if Ch in ['+','-'] then begin
         if Ch = '-' then
            Sig := -1;              { y2 }
         NextChar;
      end;
      Addend;
      P.a[k] := P.a[k] + Sig*b;     { y4 }
      while Ch in ['+','-'] do begin
         if Ch = '+' then
            Sig := 1                { y3 }
         else
            Sig := -1;              { y2 }
         NextChar; Addend;
         P.a[k] := P.a[k] + Sig*b;  { y4 }
      end;
      n := Nmax;                    { y5 }
      while (n>0) and (a[n] = 0) do {    }
         n := n-1;                  {    }
   end;
end;
```

```
begin
   ErrPos := 0; i := 0; NextChar;
   Poly;
   if Ch <> EOT then
      Error('Ожидается конец текста');
end; { Translate }

begin
   WriteLn('Перемножение полиномов');
   WriteLn('----------------------');
   WriteLn;
   repeat
      WriteLn('1-й полином');
      Write('>');
      ReadLn(S);
      Translate( S, P1, ErrPos );
   until ErrPos = 0;
   repeat
      WriteLn('2-й полином');
      Write('>');
      ReadLn(S);
      Translate( S, P2, ErrPos );
   until ErrPos = 0;
   WriteLn('Произведение');
   MultPoly( P1, P2, Q );
   WritePoly( Q );
   WriteLn;
end.
```

8.4. Программа – формульный калькулятор

Программа, полный текст которой приведен ниже, позволяет вычислять значения арифметических выражений, вводимых с клавиатуры. Выражения могут содержать числа, знаки операций +,–,*,/, круглые скобки и стандартные функции языка Паскаль.

```
program Calculator;
{ (C) С.Свердлов, 1994 }

var
   x     : real;     { Значение выражения    }
   S     : string;   { Текст выражения       }
   ErrPos: integer;  { Позиция ошибки        }
   Mess  : string;   { Сообщение об ошибке   }

{ Формульный калькулятор }
procedure Calc( S : string; { входная строка     }
    var V : real;           { значение выражения }
    var ErrPos  : integer;  { позиция ошибки     }
    var ErrMess : string    { сообщение об ошибке}
    );
const
   EOT = #0;   { Признак "конец текста" }

type
   Functions = ( ABSOLUTE, SQUARE, TRUNCATE,
                 ROUNDING, SINUS, COSINUS, ARC,
                 LOG, EXPONENT, SQUAREROOT,
                 DUMMY
               );
const
   FirstFunc = ABSOLUTE;   { первая функция     }
   LastFunc  = SQUAREROOT; { последняя функция  }
```

```
   FuncNames : array[Functions] of string[6] =
                  (  'ABS', 'SQR', 'TRUNC', 'ROUND',
                     'SIN', 'COS', 'ARCTAN', 'LN',
                     'EXP', 'SQRT', ''
                  );
var
   i  : integer;              { номер символа      }
   Ch : char;                 { очередной символ   }

procedure Error( Message : string ); { Ошибка }
   { Message - сообщение   }
begin
   if ErrPos = 0 then begin
      ErrMess := Message;
      ErrPos := i; Ch := EOT;
   end;
end;

procedure NextChar; { Читать следующий символ }
begin
   if ErrPos <> 0 then
      Ch := EOT
   else
      repeat
         i := i + 1;
         if i<=length(S) then
            Ch := UpCase(S[i])
         else
            Ch := EOT;
      until Ch <> ' ';
end;

procedure Number( var V : real ); { Число }
var
   NumStr   : string;   { Текст числа              }
   Err      : integer;  { номер неверного символа  }

procedure IntNumber; { Целое }
begin
   if not ( Ch in ['0'..'9'] ) then
      Error('Число начинается не с цифры');
   while Ch in ['0'..'9'] do begin
      NumStr := NumStr + Ch;
      NextChar;
   end;
end;
begin
   NumStr := '';
   IntNumber;
   if Ch = '.' then begin
      NumStr := NumStr + Ch;
      NextChar;
      IntNumber;
   end;
   if Ch = 'E' then begin
      NumStr := NumStr + Ch;
      NextChar;
      if Ch in ['+','-'] then begin
         NumStr := NumStr + Ch;
         NextChar;
      end;
      IntNumber;
   end;
```

```
   { Вычисление вещественного числа по его записи  }
   { Val - стандартная процедура Турбо Паскаля     }

     Val( NumStr, V, Err );
     if Err <> 0 then
        Error('Ошибка в числе');

end; { Число }

procedure Name( var F : Functions ); { Имя функции }
var
   NameStr : string;
begin
   if Ch in ['A'..'Z'] then begin
      NameStr := Ch;
      NextChar;
      end
   else
      Error('Ожидается буква');
   while Ch in ['A'..'Z','0'..'9'] do begin
      NameStr := NameStr + Ch;
      NextChar;
   end;

   { Определение функции по имени }

      F := FirstFunc;
      while ( F <= LastFunc ) and
            ( NameStr <> FuncNames[F] )
      do
         F := succ(F);
      if F=DUMMY then
         Error('Неправильное имя функции');
end;

function Func( F : Functions; x : real ) : real;
   { Значение F(x) }
begin
   Func := x;
   case F of
   ABSOLUTE    : Func := abs(x);
   SQUARE      : Func := sqr(x);
   TRUNCATE    : Func := trunc(x);
   ROUNDING    : Func := round(x);
   SINUS       : Func := sin(x);
   COSINUS     : Func := cos(x);
   ARC         : Func := arctan(x);
   LOG         :
      if x > 0 then
         Func := ln(x)
      else
         Error('Логарифм неположительного числа');
   EXPONENT    : Func := exp(x);
   SQUAREROOT  :
      if x >= 0 then
         Func := sqrt(x)
      else
         Error('Корень из отрицательного числа');
   end;
end;


procedure Expression( var V : real ); forward;
```

```
   { Выражение (опережающее описание) }

procedure Multiplier( var V : real );
   { Множитель }
var
   F : Functions;
begin
   if Ch in ['0'..'9'] then
      Number(V)
   else if Ch in ['A'..'Z'] then begin
      Name( F );
      if Ch <> '(' then
         Error('Ожидается ''('')
      else begin
         NextChar;
         Expression(V);
         V := Func( F, V );
         if Ch = ')' then
            NextChar
         else
            Error('Ожидается '')''')
         end
      end
   else if Ch = '(' then begin
      NextChar;
      Expression(V);
      if Ch <> ')' then
         Error('Ожидается '')''')
      else
         NextChar;
      end
   else
      Error('Ожидается число, функция или ''('');
end;

procedure Addend( var V : real );
   { Слагаемое }
var
   Op    : char;     { Знак операции      }
   X     : real;     { Значение операнда  }
begin
   Multiplier( V );
   while Ch in ['*','/'] do begin
      Op := Ch;
      NextChar;
      Multiplier(X);
      if Op = '*' then
         V := V*X
      else if X <> 0 then
         V := V/X
      else
         Error('Деление на ноль');
   end;
end;

procedure Expression( var V : real );
   { Выражение }
var
   Op    : char;     { Знак операции      }
   X     : real;     { Значение операнда  }
begin
   Op := '+';
   if Ch in ['+','-'] then begin
      Op := Ch; NextChar;
```

```
   end;
   Addend(V);
   if Op = '-' then V := -V;
   while Ch in ['+','-'] do begin
      Op := Ch; NextChar;
      Addend(X);
      if Op = '+' then
         V := V + X
      else
         V := V - X;
   end;
end;

begin
   ErrPos := 0;
   i := 0; NextChar;
   Expression( V );
   if Ch <> EOT then
      Error('Ожидается конец выражения');
end; { Calc }

begin    { Основная программа }
   WriteLn;
   WriteLn('ФОРМУЛЬНЫЙ КАЛЬКУЛЯТОР');
   repeat
      Write('?');
      Readln(S);
      Calc( S, x, ErrPos, Mess );
      if ErrPos > 0 then begin
         WriteLn('^':ErrPos+1);
         WriteLn( Mess );
         end
      else if abs(x) > 0.0001 then
         WriteLn( x:15:6 )
      else
         WriteLn( x );
   until S = '';
end.
```

Содержание

1. АЛГОРИТМЫ ОБРАБОТКИ ТЕКСТА 4

1.1. Пример задачи обработки текста 4

1.1.1. Разработка программы 5

2. СИНТАКСИЧЕСКИЙ АНАЛИЗ 8

2.1. Синтаксические диаграммы 9

2.2. Решение задачи синтаксического анализа 10

2.3. Построение синтаксического анализатора по синтаксическим диаграммам 13

3. СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ОБРАБОТКА ТЕКСТА 18

3.1. Включение действий в синтаксис. Семантические процедуры. 18

3.2. Трансляция полинома 19

4. ОБРАБОТКА СИНТАКСИЧЕСКИХ ОШИБОК 24

5. ИНТЕРПРЕТАЦИЯ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ 24

5.1. Синтаксис арифметических выражений 26

5.2. Разработка интерпретатора 27

6. РЕКУРСИВНЫЙ СПУСК 33

7. ЛИТЕРАТУРА 34

8. ПРИЛОЖЕНИЯ 35

8.1. Темы для самостоятельной работы 35

8.2. Индивидуальные задания по синтаксическому анализу 37

8.2.1. Порядок выполнения индивидуального задания. 37

8.2.2. Содержание отчета. 37

8.2.3. Варианты заданий 37

8.2.4. Пример выполнения индивидуального задания 40

8.3. Программа перемножения полиномов 43

8.4. Программа – формульный калькулятор 46

СОДЕРЖАНИЕ 51