Конечные автоматы. Разбор выражений

На многих олимпиадах по информатике встречаются так называемые “технические” задачи, решение которых на первый взгляд не требует знания никаких алгоритмов и зависит лишь от владения техникой программирования. К этому классу можно отнести задачи на обработку текстов и выделения с целью последующей обработки некоторых конструкций, формальное описание которых нам задано (например, адресов электронной почты). Однако существуют способы решения подобных задач, существенно облегчающие написание программы. Одним из таких способов является построение математической модели для так называемого *конечного автомата*.

Определение конечного автомата

Конечным автоматом называется реализация алгебраической структуры (S, $\sum$, m, s_0, F), где

- S — непустое множество *состояний*;
- $\sum$ — конечное множество входных символов (*алфавит*);
- m — отображение $S \times \sum \rightarrow S$, или функция *переходов*, которая каждой паре (символ, состояние) ставит в соответствие состояние из множества S;
- s_0 — состояние из S, известное как *начальное* (стартовое);
- F — множество заключительных (*допускающих*) состояний, $F \subseteq S$ или просто соответствует окончанию просмотра текста.

Иногда сюда следует добавить для каждой пары (*символ, состояние*) процедуру обработки *символа* (например, его печать). Работа автомата заключается в том, что изначально автомат находится в состоянии s_0 и под действием первого входного символа переходит в следующее состояние и читает следующий символ и т.д. Автомат заканчивает свою работу, если достигнуто одно из состояний множества F, или прочитан символ, не принадлежащий $\sum$, или входные данные исчерпаны.

Если отображение m однозначно, то есть каждой паре (символ, состояние) соответствует определенное состояние, то автомат называют *детерминированным*, в противном случае (одной и той же паре в соответствие ставится сразу несколько состояний, чаще всего в зависимости от предыдущих или последующих символов обрабатываемой входной строки) — недетерминированным. Интересно, что разница между двумя типами автоматов несущественна, так как доказано, что для любого недетерминированного конечного автомата можно построить соответствующий ему детерминированный. Последний легче реализовать, а в терминах первого проще записывать условия для большого числа задач.

На примере следующей задачи рассмотрим построение и программную реализацию конечного автомата.

Задача GoTo.

Ученики, недавно начавшие программировать, употребляют слишком много операторов **GOTO**, что является почти недопустимым для структурированной программы. Помогите преподавателю информатики написать программу, которая будет оценивать степень структурированности отлаженной программы школьника на языке Паскаль, для начала просто подсчитывая количество операторов **GOTO** в ней.

В синтаксически верной программе ключевое слово оператора перехода **GOTO** может стоять или в начале строки или после пробела или после одного из символов — “;”, “:”, “}”, а после него может стоять или пробел или перевод строки или символ “{” (табуляцию в качестве разделителя рассматривать не будем).

Напомним, что кроме обозначения действительно оператора перехода, слово “GOTO” может встречаться в тексте программы в строковых константах, заключенных в апострофы, или в комментариях, причем для простоты будем считать, что комментарий всегда начинается с символа “{”, а заканчивается первым встретившимся после этого символом “}”, при этом символ “{” должен находиться не внутри строковой константы. В этих случаях слово “GOTO” подсчитывать не нужно. Строчные и прописные буквы в Паскале не различимы.

Во входном файле `goto.in` находится текст программы без синтаксических ошибок на языке Паскаль. Размер программы не превосходит 64 Кб. В выходном файле `goto.out` должно оказаться одно число – количество операторов **GOTO** в этой программе.

Пример входного файла:

```
label 1, 2;
var I: byte;
begin
  1: I := I + 1;
  if I > 10 then goto 2 else
    writeln(' goto '); goto 1;
  2: if I < 10 then goto 1{else { goto 2;}
  end.
```

Выходной файл для приведенного примера:

3

Решение. Множество состояний для конечного автомата, с помощью которого легко осуществить решение данной задачи состоит из трех элементов — соответствующих комментарию, строковой программе и собственно программе (в программной реализации автомата они обозначаются константами перечислимого типа `com`, `str`, `prog`. Алфавит в данном случае совпадает со всеми символами кодовой таблицы. Функция переходов для большинства символов из этого алфавита оставляет состояние прежним. Исключение составляют лишь следующие пары:

(“}”, `com`) → `prog`;

(‘'’, `str`) → `prog`;

(“{”,) → `com`;

(‘'’, `prog`) → `str`.

Начальным является состояние `prog`. В этом же состоянии для символов “G”, “g” следует применить процедуру обработки, проверяющую наличие в данном месте программы оператора **GOTO**. Данную информацию удобно записать в виде следующей таблицы:

Состояние	Символ	Операция	Новое состояние
com	}	Перейти к следующему символу	prog
	Остальные символы	Перейти к следующему символу	com
str	'	Перейти к следующему символу	prog
	Остальные символы	Перейти к следующему символу	str
prog	{	Перейти к следующему символу	com
	'	Перейти к следующему символу	str
	G g	Проверить наличие оператора GOTO Перейти к следующему символу	prog
	Остальные символы	Перейти к следующему символу	prog
	Конец текста	Вывести результат подсчета и завершить работу	

Теперь написать программу, реализующую работу описанного детерминированного автомата, не составит труда.

```
const
  sr: set of char = [' ', ';', ':', '}'];

var
  cgoto: word;{счетчик операторов GOTO}
  s, s1: string;
  i, j: byte;
  state: (com, str, prog);

begin
  assign(input, 'goto.in'); reset(input);
  cgoto := 0;
  state := prog;
```

```
  while not eof do
  begin
    readln(s);
    {s – очередная строка текста программы}
    for i := 1 to length(s) do
      case state of
        com: if s[i] = '}'  then state := prog;
        str: if s[i] = '''' then state := prog;
        prog:
          if s[i] = '''' then state := str else
          if s[i] = '{'  then state := com else
          if (s[i] in ['g', 'G']) and
             ((i = 1) or (s[i - 1] in sr)) then
          begin
            s1 := copy(s, i, 5);
            for j := 1 to length(s1) do
              s1[j] := upcase(s1[j]);
            if (s1 = 'GOTO'){это конец строки}
             or (s1 = 'GOTO ') or (s1 = 'GOTO{')
              then
              inc(cgoto)
          end
      end{case}
  end; {while}
  assign(output, 'goto.out'); rewrite(output);
  writeln(cgoto)
end.
```

Проверка арифметического выражения на корректность

При рассмотрении различных вопросов, касающихся арифметических выражений, обычно оперируют следующими терминами. *Операнд* — число, переменная или выражение в скобках. *Бинарная операция* — операция, выполняющаяся над двумя операндами, например, сложение. *Унарная операция* — операция, выполняющаяся над одним операндом, знак этой операции обычно стоит непосредственно перед операндом, например, унарный минус или функция одной переменной.

Прежде чем проверять корректность того или иного выражения (не обязательно арифметического), его следует формально описать (по-другому говорят — ввести грамматику). Существует несколько способов для подобных описаний, например, синтаксические диаграммы, регулярные выражения, форма Бэкуса — Наура. Воспользуемся последним способом.

```
<выражение>::=<терм>|<терм>+<выражение>|<терм>-<выражение>
<терм>::=<множитель>|<множитель>*<терм>|<множитель>/<терм>
<множитель>::=(<выражение>)|<имя>|<натуральное число>
<имя>::=<буква>|<имя><буква>|<имя><цифра>
<натуральное число>::=<цифра>|<натуральное число><цифра>
<цифра>::=0|1|2|3|4|5|6|7|8|9
<буква>::=_|A|B|…|Z|a|b|…|z|
```

Здесь слева от метазнака “**::=**” стоит определяемое понятие, а справа — его определение. Знаки “|” обозначают логическую операцию ИЛИ в определении, остальные символы входят в определение того или иного понятия. В так введенной грамматике для арифметического выражения отсутствуют унарные операции.

Для проверки соответствия некоторого выражения данной грамматике можно построить конечный автомат, имеющий три допускающих состояния (0, 1 и 2 в программной реализации) плюс состояние обнаружения ошибки. В каждом из трех состояний допустимыми являются лишь некоторые символы. Наличие же на входе другого символа говорит об ошибке. Так стартовое нулевое состояние говорит о том, что выражение может начинаться с цифры, буквы или символа “(”. В первых двух случаях следует пропустить число или имя целиком и перейти в состояние 1, открывающаяся же скобка оставляет автомат в том же состоянии. В первом состоянии допустимыми являются лишь знаки бинарных арифметических операций и закрывающаяся скобка. Знак операции переводит автомат в состояние 0, а закрывающаяся скобка — оставляет его в состоянии 1. Если текущий символ — последний в анализируемой строке, то автомат переводится в состояние 2, допустимым для которого являются буквы, цифры и закрывающаяся скобка. Остается только проверить, что общее число

открывающихся скобок равно числу закрывающихся скобок, а при встрече очередной закрывающейся скобки общее их количество к этому моменту не может превышать количество уже встретившихся открывающихся скобок. Для этого вводится целочисленная переменная — счетчик скобок (в программе k), первоначально равная 0. Если в выражении встретилась открывающаяся скобка (состояние 0), то к счетчику прибавляется единица, если закрывающаяся (состояние 1), то единица вычитается. Данная переменная по ходу вычислений не должна принимать отрицательных значений, а при завершении просмотра (состояние 2) должна быть в точности равна нулю.

Выпишем таблицу, задающую описанный автомат.

Состояние	Символ	Операция	Новое состояние
0	последний символ		2
	(	k := k + 1, перейти к следующему символу	0
	буква или цифра	пропустить имя или число, перейти к следующему символу	1
	остальные символы	ошибка, конец работы	
1	последний символ		2
	)	k := k – 1, проверить, что k ≥ 0, перейти к следующему символу	1
	+, -, *, /	перейти к следующему символу	0
	остальные символы	ошибка, конец работы	
2	), буква или цифра	если k = 0, то выражение корректно, в противном случае – ошибка, конец	
	остальные символы	ошибка, конец работы	

Приведем программу, реализующую работу этого автомата.

```
const
  digits: set of char = ['0'..'9'];
  letters: set of char = ['_', 'A'..'Z', 'a'..'z'];
  op: set of char = ['+', '-', '*', '/'];

var
  s: string;
  i, k: word;
  state: 0..3;

procedure error;
begin
  writeln('Выражение некорректно'); halt
end;

procedure Identifier; {пропустить имя}
begin
  while (i < length(s)) and
        (s[i + 1] in (letters + digits)) do i := i + 1
end;

procedure Number;{пропустить число}
begin
  while (i < length(s)) and
         (s[i + 1] in digits) do i := i + 1
end;

begin {Main}
  readln(s);
  i := 0;
  k := 0;
  state := 0;
  while state <> 3 do
    case state of
      0:
        if i < length(s) then
        begin
          i := i + 1;
          if s[i] = '(' then k := k + 1 else
          if not (s[i] in (letters + digits))
            then error else
```

```
          begin
            if s[i] in letters then Identifier
            else if s[i] in digits then Number;
            state := 1
          end
        end else state := 2;
      1:
        if i < length(s) then
        begin
          i := i + 1;
          if s[i] = ')' then
          begin
            k := k - 1;
            if k < 0 then error
          end
          else
          if s[i] in op then state := 0
          else error
        end else state := 2;
      2:
        if (s[i] in (letters + digits + [')'])) and (k = 0) then
        begin
          writeln('Выражение корректно');
          state := 3
        end
        else error
    end {case}
end.
```

Попробуйте изменить грамматику и построенный конечный автомат так, чтобы в арифметических выражениях допускался унарный минус перед числом, именем или выражением в скобках. Придется ли в этом случае увеличивать число состояний?

Стековый конечный автомат

Изменим грамматику арифметического выражения так, чтобы допустимыми в выражении являлись сразу три вида скобок: круглые, квадратные и фигурные. Для этого к определению множителя следует добавить следующее описание: |[выражение]|{выражение}. К сожалению, подсчет в отдельности сбалансированности скобок каждого вида не гарантирует в данном случае корректности выражения. Например, тогда выражение [1+2+{3-4]*5} будет считаться корректным, что неверно. В такой ситуации все встречающиеся открывающиеся скобки следует заносить в стек. Анализ же корректности закрывающейся скобки будет состоять в ее сравнении с верхним элементом стека (т. е. последней занесенной в него открывающейся скобкой). При соответствии скобок друг другу открывающаяся скобка из стека извлекается, в противном случае — выражение некорректно. При достижении конца входных данных следует проверить, что стек пуст.

Перейдем теперь к рассмотрению различных способов подсчета значений арифметических выражений.

“Палочный” способ разбора арифметических выражений

Данный способ подсчета значения арифметического выражения фактически моделирует действия человека, выполняемые при вычислении арифметических выражений. То есть сначала ищется операция, которую можно выполнить первой, она выполняется и в измененном выражении вновь ищется первая выполнимая операция. Причем в большинстве случаев человек при этом понятие “первой выполнимой операции” не формализует строго, а опирается на многолетний практический опыт работы с арифметическими выражениями, хотя в школьных учебниках по математике для начальной школы эти правила в том или ином виде сформулированы. Для компьютерной же реализации этого метода нужна четкая система правил. Прежде чем ее выписать, поставим в соответствие исходному арифметическому выражению строку, в которой каждый элементарный операнд из выражения (в случае подсчета такими операндами являются только числа) заменим на символ “|” (“палочку”), а знаки операций и скобки оставим неизменными.

Например, выражению `32/(2*4)+10+(5-3-1)` соответствует строка `|/(|*|)+|+(|-|-|)`. Теперь выпишем в терминах “палочек” действия, которые следует выполнять при подсчете значения арифметического выражения.

1.	`(|) → |`	Скобки можно снять
2.	`|*| или |/| → |`	Умножение или деление, встретившееся первым, можно выполнять
3.	`(|±|) → (|) или (|±|± → (|±`	В данном контексте сумму или разность можно вычислять
4.	`|±| → |`	Если 1 – 3 применить нельзя, то эти операции выполнить можно

Применять указанные правила следует так. В “палочной” строке ищется первое вхождение подстроки сначала из левой части правила `1`, если такая не найдется вообще — то из правила `2` и т.д. Для найденной подстроки осуществляется замена согласно правилу, а над соответствующими этой подстроке элементами исходного выражения производятся те же арифметические действия, что и в найденной подстроке (для правила `1` — просто снимаются скобки). После этого то же самое правило пытаются применить еще раз, а если это невозможно, то снова переходят к поиску подстроки из правила `1` (а не из следующего правила, что весьма существенно). Можно доказать, что таким образом к поиску подстроки из левой части правила `4` мы приступим лишь тогда, когда в выражении уже не останется ни скобок, ни операций умножения или деления. Действия заканчиваются, когда у нас останется одна палочка, т.е. исходное выражение будет сведено к одному числу, являющемуся его значением. Покажем, как по этим правилам будет вычисляться значение выражения из примера.

	`|/(|*|)+|+(|-|-|)`	`32/(2*4)+10+(5-3-1)`
по правилу 2	`|/(|)+|+(|-|-|)`	`32/(8)+10+(5-3-1)`
по правилу 1	`|/|+|+(|-|-|)`	`32/8+10+(5-3-1)`
по правилу 2	`|+|+(|-|-|)`	`4+10+(5-3-1)`
по правилу 3	`|+|+(|-|)`	`4+10+(2-1)`
по правилу 3	`|+|+(|)`	`4+10+(1)`
по правилу 1	`|+|+|`	`4+10+1`
по правилу 4	`|+|`	`14+1`
по правилу 4	`|`	`15`

На первый взгляд может показаться, что правила `3` являются лишними. Однако это не так. Например, выражение `2+3*(4-5+6)` без любого из правил 3 будет вычислено неверно.

Программу, реализующую описанный алгоритм подсчета написать нетрудно. “Палочное” выражение легко представить с помощью переменной типа **`string`** (тогда поиск любой подстроки можно выполнять с помощью стандартной функции `Pos`), а исходное выражение — с помощью массива записей, одно поле которых символьное, а второе — числовое: если *i*-й элемент выражения символ скобки или арифметической операции, то у *i*-го элемента массива соответствующим символом будет заполнено только символьное поле, если же элемент — число, то в символьном поле можно поставить все ту же палочку, а само число поместить в числовое поле. При таком способе представления исходного арифметического выражения *i*-й символ строки будет соответствовать *i*-у элементу массива. При дальнейших преобразованиях это соответствие нужно сохранять (из строки удалять лишние символы и сдвигать массив влево на место удаляемых элементов).

Однако такой способ подсчета арифметических выражений весьма неэффективен и, например, в компиляторах не применяется. Ведь для выполнения очередной арифметической операции нам придется просмотреть в худшем случае всю строку из палочек, причем возможно не один раз (ведь даже нахождение операции умножения из правила `2` еще не означает, что ее можно выполнять, т.к. операция деления может встретиться раньше). Поэтому перейдем к рассмотрению более эффективных способов разбора, правда не столь очевидных.

Подсчет арифметических выражений с помощью постфиксной нотации

Любое арифметическое выражение можно переписать в виде бесскобочного постфиксного выражения, в котором знак операции стоит после своих операндов, а обозначение функции — после всех ее аргументов. Такую запись еще называют *обратной польской нотацией* (*записью*). Преимущество этой записи заключается в том, что все встречающиеся в ней операции выполняются последовательно слева направо. Разделим алгоритм подсчета арифметического выражения на две части — перевод в обратную польскую запись и подсчет по ней значения выражения.

Перевод в обратную польскую запись.

При выполнении следующих действий над исходной строкой с арифметическим выражением мы будем использовать вспомогательную структуру данных — стек и строку для записи результата. Просматриваем исходную строку с выражением один раз слева направо.

1. Если очередной элемент — число или имя переменной, то он сразу переносится в результирующую строку.
2. Открывающаяся скобка всегда помещается в стек.
3. Если встретилась закрывающаяся скобка, то из стека извлекаются все знаки операций до первой открывающейся скобки и в порядке извлечения переносятся в результирующую строку, а открывающаяся скобка в вершине стека уничтожается.
4. Если встречается знак операции, то
   a) если в вершине стека знак операции с более низким приоритетом или открывающаяся скобка или ничего нет, то встретившийся знак заносится в стек;
   b) в противном случае знаки операций из стека, пока приоритет их больше или равен приоритету данного знака, извлекаются из стека и заносятся в результирующую строку, после чего рассматриваемый знак записывается в стек.
5. Если исходная строка исчерпана, то оставшиеся в стеке знаки операций по порядку переносятся в результирующую строку.

Например, постфиксной записью для `9/(5+2*3-8)` будет `9523*+8-/`.

Несложно заметить, что эти действия легко описать и реализовать с помощью стекового конечного автомата. Интересно, что для данного алгоритма преобразования выражения неважно, какие именно операции в нем могут встречаться. Нужно только знать приоритеты всех допустимых операций. Укажем приоритеты наиболее типичных операций в арифметических выражениях (от высшего к низшему):

1) унарный минус, функция;
2) умножение, деление, в том числе целочисленные;
3) сложение, вычитание.

При преобразовании выражения в постфиксную нотацию унарный минус следует обозначать специальным знаком, например символом подчеркивания, в противном случае вычисления по такой записи могут быть произведены неверно.

Рассмотрим теперь как по построенной постфиксной записи подсчитать значение выражения.

Подсчет значения выражения, записанного в польской нотации.

Строка с выражением в “обратной польской записи” просматривается один раз слева направо. В качестве вспомогательной структуры данных опять используется стек.

1. Если встретившийся элемент — операнд, то он помещается в стек.
2. Если очередной элемент — знак бинарной операции, то из стека извлекаются два верхних элемента и над ними выполняется операция, результат которой заносится в стек, унарная операция выполняется над верхним элементом.

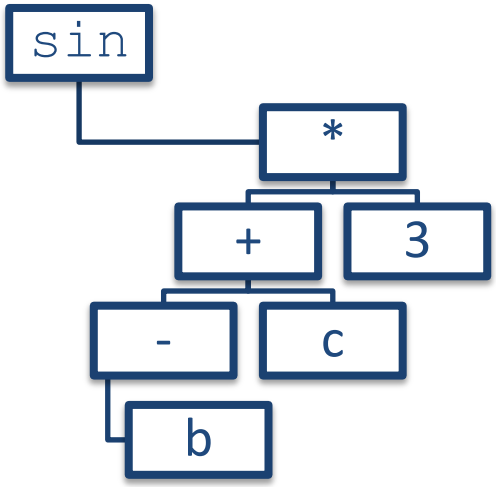
В конце концов в стеке остается одно число — результат.

В нашем примере сначала в стек попадают операнды 9, 5, 2 и 3 (верхний элемент – 3). Покажем содержимое стека после выполнения каждого действия алгоритма подсчета. Именно таким образом в большинстве компиляторов осуществляется разбор арифметических и логических выражений.

Операция	Стек
*	9 5 6
+	9 11
8	
–	
/	

Метод рекурсивного спуска

Альтернативой стековому представлению выражения является так называемое *дерево разбора*. Для арифметического выражения его можно описать так. В корне бинарного дерева находится знак операции, которая должна быть выполнена последней. Левым поддеревом (сыном) является дерево, представляющее первый из операндов последней операции, правым — второй из операндов. Если эта операция унарная, то левое поддерево отсутствует. Для функции одной переменной правое поддерево представляет его аргумент. Концевые узлы дерева (листья) представляют числа или имена переменных. Например, для выражения sin((–b + c)•3) дерево разбора выглядит так:



Подсчет значения арифметического выражения, основанный на таком способе представления рекурсивен и фактически совпадает с рекурсивным определением самого выражения. Так, выражение представляет из себя сумму слагаемых (здесь под суммой подразумевается выполнение операций как сложения так и вычитания). Каждое слагаемое представляет из себя произведение множителей (сюда же отнесена операция деления). А множитель представляет из себя либо число, либо переменную, либо выражение в скобках (при наличии унарного минуса множителем является также выражение со стоящим перед ним знаком “минус”).

Ниже приведены две реализации рекурсивного спуска. В левой из них производится только подсчет значения арифметического выражения. В качестве элементарных операндов используются только числа. В правой — сначала строится дерево разбора, элементарными операндами в котором являются однозначные цифры или однобуквенные имена переменных. Унарный минус в обоих случаях допускается. Если в построенном с помощью правой программы дереве все листья — однозначные числа, то значение выражения также будет верно подсчитано. Бинарное дерево строится с использованием динамических переменных. Преимущество второй программы заключается в том, что с помощью построенного дерева можно решать и другие задачи, например, перебирать все возможные способы расстановки скобок или распараллеливать на насколько процессоров вычисление значения выражения.

В данной лекции мы рассмотрели лишь применение конечных автоматов и некоторые способы разбора выражений, не доказывая строго возможность их использования в тех или иных задачах. О теории конечных автоматов, формальных способах описания выражений и методах их разбора более подробно можно прочитать в [1 – 4].

Литература

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2001.
2. *Бауэр Ф.Л., Гооз Г.* Информатика. Вводный курс. Часть 2, М.: Мир, 1990.
3. *Кук Д., Бейз Г.* Компьютерная математика. М.: Наука, 1990.
4. *Шень А.* Программирование: теоремы и задачи. М.: МЦНМО, 1995.

```
var s: string; {исходное выражение}
    i: integer; {номер текущего символа}

function Mul: longint; forward;
function Factor: longint; forward;

///Суммирует слагаемые
function Add: longint;
var
  q, res: longint;
  c: char;
begin
  res := Mul; {первое слагаемое}
  while s[i] in ['+', '-'] do
  begin
    c := s[i];
    i := i + 1;
    q := Mul; {очередное слагаемое}
    case c of
      '+': res := res + q;
      '-': res := res - q;
    end
  end; {while}
  Add := res
end;

///Перемножает множители
function Mul: longint;
var
  q, res: longint;
  c: char;
begin
  res := Factor; {первый множитель}
  while s[i] in ['*', '/'] do
  begin
    c := s[i];
    i := i + 1;
    q := Factor; {очередной множитель}
    case c of
      '*': res := res * q;
      '/':
        if q = 0 then
        begin
          writeln('деление на 0');
          halt
        end
        else res := res div q
    end {case}
  end; {while}
  Mul := res
end;

///Выделяет число
function Number: longint;
var
  res: longint;
begin
  res := 0;
  while (i <= length(s)) and
        (s[i] in ['0'..'9']) do
  begin
    res := res * 10 + (ord(s[i]) - ord('0'));
    i := i + 1
  end;
  Number := res
end;

///Выделяет множитель
function Factor: longint;
```

```
var
  q: longint;
  c: char;
begin
  case s[i] of
    '0'..'9': Factor := Number;
    '(':
      begin
        i := i + 1;Factor := Add;
        i := i + 1; {пропустили ')'}
      end;
    '-':
      begin
        i := i + 1;
        Factor := -Factor;
      end
  else begin
      writeln('ошибка');
      halt
    end
  end {case}
end;

begin {основная программа}
  readln(s); i := 1;
  writeln(Add)
end.
```

А теперь программа, строящая дерево разбора.

```
var
  s: string; {исходное выражение}
  i: integer; {номер текщего симвовла}

function Mul: longint; forward;
function Factor: longint; forward;

///Суммирует слагаемые
function Add: longint;
var
  q, res: longint;
  c: char;
begin
  res := Mul; {первое слагаемое}
  while s[i] in ['+', '-'] do
  begin
    c := s[i];
    i := i + 1;
    q := Mul; {очередное слагаемое}
    case c of
      '+': res := res + q;
      '-': res := res - q;
    end
  end; {while}
  Add := res
end;

///Перемножает множители
function Mul: longint;
var
  q, res: longint;
  c: char;
begin
  res := Factor; {первый множитель}
  while s[i] in ['*', '/'] do
  begin
    c := s[i];
    i := i + 1;
```

```
    q := Factor; {очередной множитель}
    case c of
      '*': res := res * q;
      '/':
        if q = 0 then
        begin
          writeln('деление на 0');
          halt
        end
        else res := res div q
    end {case}
  end; {while}
  Mul := res
end;

///Выделяет число
function Number: longint;
var
  res: longint;
begin
  res := 0;
  while (i <= length(s)) and
        (s[i] in ['0'..'9']) do
  begin
    res := res * 10 + (ord(s[i]) - ord('0'));
    i := i + 1
  end;
  Number := res
end;

///Выделяет множитель
function Factor: longint;
var
  q: longint;
  c: char;
begin
  case s[i] of
    '0'..'9': Factor := Number;
    '(':
      begin
        i := i + 1;Factor := Add;
        i := i + 1; {пропустили ')'}
      end;
    '-':
      begin
        i := i + 1;
        Factor := -Factor;
      end
  else begin
      writeln('ошибка');
      halt
    end
  end {case}
end;

begin {основная программа}
  readln(s); i := 1;
  writeln(Add)
end.
```