```
25  num:=0;
26  for v := 1 to n do
27    if color[list[v]] = 0 then begin
28      inc(num);
29      Dfs2(list[v]);
30    end;
```

По окончании работы программы переменная `num` будет содержать количество компонент сильной связности графа G, а в массиве `color` будут храниться номера компонент сильной связности, к которым принадлежат соответствующие вершины.

Построение транспонированного графа G^T занимает $O(V+E)$ времени при хранении графа списками смежности или списком ребер и $O(V^2)$ — при хранении графа матрицей смежности.

В заключение отметим, что все предложенные в статье алгоритмы, основанные на поиске в глубину, имеют оценку сложности $O(V+E)$ при хранении графа списками смежности или списком ребер и $O(V^2)$, соответственно, — при хранении графа матрицей смежности. Оценка времени работы в большинстве из предложенных алгоритмов определяется именно временем работы поиска в глубину.

Литература

1. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.
2. *Ахо А.В.*, *Хопкрофт Д.*, *Ульман Дж.Д.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2003.
3. *Макконнелл Дж.* Анализ алгоритмов. Вводный курс. М.: Техносфера, 2002.
4. *Окулов С.М.* Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004.
5. *Оре О.* Теория графов. М.: Наука, 1968.

Подсчет значения арифметического выражения методом рекурсивного спуска

В. А. Матюхин

На практике часто встречается следующая задача: пользователь вводит арифметическое выражение, например: `(1+2*3)*(4+5)+6*(7+8)+9`, нужно вычислить его значение. Если вы когда-нибудь пробовали писать программу, решающую такую задачу, то, наверное, понимаете, что это не так просто, как кажется с первого взгляда. Если никогда не пробовали, то для того чтобы в полной мере оценить изящность метода, который будет описан в этой статье, попробуйте отложить книгу, сесть за компьютер и попытаться написать такую программу.

Мы рассмотрим красивое решение описанной задачи, использующее *метод рекурсивного спуска*. Фрагменты программы будут приводиться на языке Паскаль, однако думаем, что перевод их на язык Си (равно как и на любой другой) не должен составить большого труда даже для тех, кто с языком Паскаль не знаком.

Сначала рассмотрим более простую задачу: предположим, что все числа у нас — натуральные, а из операций встречаются только сложение и умножение.

Разбиение на лексемы

Для начала давайте напишем процедуру `nextlexem`, которая будет из выражения выделять поочередно все лексемы. *Лексема* — это минимальная единица текста, имеющая самостоятельный смысл. В нашем случае лексемами будут являться знаки арифметических операций, скобки и числа (при этом — обратите внимание! — лексемой является не каждая цифра числа, а число целиком). Эта же процедура будет игнорировать все незначимые символы — пробелы, табуляции и т. д. Еще одна лексема, которая нам понадобится, — конец строки — будет соответствовать тому, что мы хотим выделить из строки следующую лексему, а строка кончилась.

Итак, опишем следующий тип данных:

```
type TLexem = (_Num, _Plus, _Mul, _Open,
               _Close, _End);
```

Можно сделать практически то же самое по-другому:

```
type TLexem = byte;
const _Num = 0;          {Число}
      _Plus = 1;         {Знак сложения}
      _Mul = 2;          {Знак умножения}
```

```
_Open = 3;          {Открывающая скобка}
_Close = 4;         {Закрывающая скобка}
_End = 5;           {Конец выражения}
```

Имена `_Num, _Plus` и т. д. мы умышленно начинаем с подчеркивания, чтобы случайно не перепутать их с именем какой-либо переменной, функции и т. д.

Введем две переменных:

```
var curlex:TLexem;          {текущая лексема}
    vl:Longint;             {значение}
```

В переменной `curlex` будем хранить текущую лексему, выделенную из выражения, каждый вызов процедуры `nextlexem` будет устанавливать значение этой переменной. И если в случае, когда текущей лексемой является `_Plus, _Mul` и т. д., эта информация является исчерпывающей, то когда текущая лексема — число (`_Num`), необходимо еще знать значение этого числа. Для этого будет использоваться переменная `vl` (ее значение также будет устанавливаться процедурой `nextlexem`).

Написание процедуры `nextlexem` никаких алгоритмических трудностей не представляет, хотя и не является совсем тривиальным. Скорее написание этой процедуры — довольно рутинная работа, требующая достаточной аккуратности. Мы оставляем ее читателям в качестве самостоятельного упражнения.

В дальнейшем будем считать, что эта процедура уже написана и ее вызов устанавливает значение глобальной переменной `curlex`, а в случае, если `curlex=_Num`, то и переменной `vl` (в остальных случаях в переменной `vl` может быть записано любой значение). Первый вызов процедуры `nextlexem` должен выделять первую лексему. Второй вызов — вторую и т. д.

Определение арифметического выражения

Теперь давайте попробуем определить, что же собственно мы хотим вычислить, т. е. что же такое арифметическое выражение. Для этого давайте ненадолго вернемся в детство и вспомним, как же происходит вычисление выражений.

Сначала выполняются все умножения и операции внутри скобок и над этими операциями записываются их результаты. Можно представить, что мы стираем знаки тех операций, которые мы выполнили, и те числа, которые в них участвовали, и записываем вместо этого результаты этих операций. Тогда на последнем шаге нам придется вычислять что-то типа `63+90+9` (здесь и дальше все вычисления мы будем показывать на примере, который уже был приведен в начале статьи: `(1+2*3)*(4+5)+6*(7+8)+9`).

Итак, можно сказать, что *арифметическое выражение* (употребляя слово «выражение», далее будем иметь в виду, конечно, арифметическое выражение) — это последовательность слагаемых (состоящая из одного или более слагаемого), разделенных знаком «+». Формально это записывается так:

```
<Выражение> ::= <слагаемое> {+ <слагаемое>}
```

Здесь фигурные скобки означают, что то, что в них записано, может быть повторено 0 или более раз (повторено 0 раз, т. е. ни разу — например, выражение `6*7` состоит из одного слагаемого).

Давайте теперь заглянем глубже и аналогично попробуем сформулировать, что же такое слагаемое. *Слагаемое* — это последовательность множителей (состоящая из одного или более множителей), разделенных знаком «*».

```
<Слагаемое> ::= <множитель> {* <множитель>}
```

Наконец, множитель — это число или выражение, заключенное в скобки:

```
<Множитель> ::= <число> | (<выражение>)
```

Здесь вертикальная черта | обозначает «или».

Таким образом, понятие выражения мы определили рекурсивно через само это понятие. Это определение кажется очень странным, поэтому мы проиллюстрируем его примером.

Итак, рассмотрим выражение `(1+2*3)*(4+5)+6*(7+8)+9` (см. рис. 1). Оно состоит из трех слагаемых: `(1+2*3)*(4+5)`, `6*(7+8)` и `9`. Первое слагаемое, в свою очередь, состоит из двух множителей: `(1+2*3)` и `(4+5)`. Первый из этих множителей — это есть выражение, взятое в скобки. В свою очередь, это выражение состоит из двух слагаемых: `1` и `2*3`. Первое из них состоит из одного множителя, который является числом `1`. Второе: `2*3` состоит из двух множителей, каждый из которых является числом. Но если бы, скажем, вместо числа `3` было записано, например `(1+2)`, то этот множитель являлся бы выражением, взятым в скобки, и в свою очередь, состоял из двух слагаемых и т. д.

Изображенная на рис. 1 структура называется *деревом разбора выражения* (`мн.` обозначает множитель, `сл.` — слагаемое, `выраж.` — выражение).

Попробуйте взять произвольное выражение и самостоятельно «разложить» его на составные части в соответствии с нашим определением.

Давайте подытожим, что же у нас получилось.

```
<Выражение> ::= <слагаемое> {+ <слагаемое>}
<Слагаемое> ::= <множитель> {* <множитель>}
<Множитель> ::= <число> | (<выражение>)
```

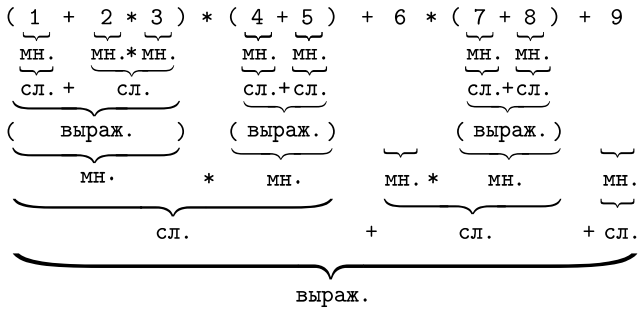

Рис. 1

Говорят, что мы определили *грамматику*, задающую арифметические выражения. А сам способ записи грамматики, который мы использовали, называется *формулами Бэкуса—Науэра*.

Написание программы

Теперь перейдем к написанию программы. Писать программу будем строго следуя нашим определениям.

Напишем три функции.

```
function expr:longint;
```

Будем считать, что при вызове этой функции в `curlex` записана первая лексема арифметического выражения. Тогда эта функция будет вычислять значение этого выражения и возвращать его в качестве своего результата. После выполнения этой функции в `curlex` будет записана первая лексема, следующая за этим выражением.

Например, если дано выражение

```
(1+2*3)*(4+5)+6*(7+8)+9
```

и при вызове функции в `curlex` записана первая открывающая скобка, то в качестве результата функции будет возвращено значение всего выражения (т. е. число `162`). А в `curlex` будет записано `_End`. Если же при вызове текущая лексема — единица (т. е. `curlex=_Num, vl=1`), то результатом будет значение выражения `1+2*3`, т. е. число `7`, а в `curlex` будет стоять закрывающая скобка, идущая после этого выражения.

Также напишем функцию

```
function item:longint;
```

которая аналогично функции `expr` будет вычислять значение слагаемого, первая лексема которого является при вызове текущей, и функцию

```
function mult:longint;
```

которая будет вычислять значение множителя.

Начнем с функции `expr`, считая, что остальные функции уже написаны.

```
function expr:longint;
var a:longint;  {эта переменная будет использоваться
                 для хранения промежуточных результатов}
begin
```

Смотрим на определение. Выражение всегда начинается со слагаемого. Заметим, что первая лексема выражения всегда является первой лексемой слагаемого. А вычислить значение слагаемого мы можем с помощью функции `item`:

```
a:=item;
```

Заметим, что после этого вызова в переменной `a` записано значение этого слагаемого, а `curlex` указывает на первую лексему за этим слагаемым. Теперь возможны две ситуации: если текущая лексема — знак «+», то за ним идет еще одно слагаемое, дальше снова может идти «+» и т. д. Второй вариант — не «+», но тогда это уже к выражению отношения не имеет, т. е. выражение закончилось. Итак:

```
while curlex=_Plus do begin
  {Раз текущая лексема - плюс, то мы должны, запомнив это,
                                перейти к следующей лексеме}
  nextlexem;
  {теперь curlex - первая лексема слагаемого, которое надо
                        прибавить к тому, что записано в a}
  a:=a+item;
  end;
expr:=a;
end;
```

Итого, нашу функцию мы написали буквально в 5 строк.

Теперь напишем функцию `item`. Заметьте, что определение слагаемого полностью аналогично определению выражения. Значит, и функция будет полностью аналогична функции `expr`:

```
function item:longint;
var a:longint;
begin
a:=mult;
while curlex=_Mul do begin
  nextlexem;
  a:=a*mult;
  end;
item:=a;
end;
```

Осталось совсем чуть-чуть: написать функцию mult. Сделаем это тоже в соответствии с определением.

```
function mult:longint;
begin
case curlex of
_Num: begin
  {текущая лексема - число, значит, множитель равен числу,
                                        хранящемуся в переменной vl}
  mult:=vl;
  {но дальше надо не забыть сдвинуться на следующую лексему -
   ведь curlex должно указывать на следующую после множителя лексему}
  nextlexem;
  end;
_Open: begin
  {текущая лексема - открывающая скобка}
  nextlexem; {сдвигаемся на следующую лексему}
  {теперь мы стоим на первой лексеме выражения, значение которого
                                    и будет значением нашего множителя}
  mult:=expr;
  {теперь мы стоим на первой лексеме после выражения.
  Это должна быть закрывающая скобка, которую нужно "проглотить" -
  ведь мы должны закончить на следующей за нашим множителем лексеме,
                               а скобка - последняя лексема множителя}
  if curlex=_Close then nextlexem
  else error; {если вдруг оказалось, что текущая лексема была
  не закрывающая скобка, значит, наше выражение содержит ошибки,
  например: (1+2, т.е. скобка не закрывается, тогда мы должны выдать
         сообщение об ошибке, пусть у нас это делает процедура error}
  end
else error;
  {если первая лексема вычисляемого множителя - не число
  и не открывающая скобка, то это опять же является признаком того,
                               что в выражении ошибка, например: 1**2}
end;
```

Осталось написать основную программу.

Помимо описанных ранее типа TLexem и переменных curlex и vl, нам потребуется еще одна переменная:

```
var v:longint;
begin
{здесь должна идти инициализация переменных,
            ввод выражения пользователем - оставляем это вам}
nextlexem; {счиываем первую лексему. Теперь в curlex записана
           первая лексема нашего выражения,
                  и его можно вычислить, вызвав функцию expr}
v:=expr;
{теперь, по идее, мы должны оказаться в конце выражения,
             если это не так, значит, в выражении есть ошибки}
if curlex<>_End then error;
writeln(v);  {печатаем результат}
end. {все!}
```


Это действительно все. Похоже на фокус.

Осталась, пожалуй, лишь одна техническая проблема. Функция expr использует функцию item, item использует mult, а mult использует expr. Какая из них должна быть описана раньше? Оказывается, что Паскаль дает возможность решить эту проблему. Можно, например, описать функции в таком порядке: сначала mult, потом — item, и наконец expr. А для того чтобы уже в функции mult компилятор знал, что функция expr присутствует в программе, но будет описана позднее, нужно перед описанием mult написать прототип (заголовок) функции expr и ключевое слово forward, которое как раз и говорит компилятору, что функция будет описана позднее. Таким образом,

```
function expr:longint; forward;
```

Пример

Давайте посмотрим, как это все будет работать.

Итак, пусть нам дали наше любимое выражение:

(1+2*3)*(4+5)+6*(7+8)+9.

Первый вызов nextlexem из основной программы установит curlex==_Open. Дальше мы вызываем expr, expr устроена так, что она сразу вызывает item, а item вызывает mult. mult видит открывающую скобку, «проглатывает» ее, в curlex оказывается _Num и vl=1. Дальше мы вызываем expr, expr устроена так, что она сразу вызывает item, а item вызывает mult. mult видит, что curlex=_Num, тогда она возвращает значение vl=1 и вызывает nextlexem. Теперь curlex=_Plus. После этого mult завершается и мы попадаем в item, из которого произошел вызов. Заметьте, что мы как раз стоим на первой лексеме после считанного множителя. _Plus — это не _Mul, поэтому item считает, что слагаемое закончилось (тем самым, оно состоит из одного множителя, и это действительно так) и мы возвращаемся в expr.

expr видит _Plus и понимает, что в выражении есть еще одно слагаемое. Запоминая в a значение 1, которое является значением первого слагаемого, она вызывает nextlexem, после чего curlex=_Num, vl=2. Вызываем item, item вызывает mult. mult возвращает vl=2 и делает nextlexem,

после чего `curlex=_Mul`. Тем самым, в `item` мы попадаем в цикл `while`, делаем `nextlexem` (в результате `curlex=_Num, vl=3`) и снова вызываем `mult`. `mult`, делая `nextlexem` (в результате `curlex=_Close`), возвращает нам 3. `item` умножает 2 на 3 и, поскольку `curlex<>_Mul`, цикл завершается. Значением `item` является `6`, и, возвращаясь в `expr`, мы к имевшемуся там числу `1` прибавляем `6`. После чего `curlex<>_Plus`, и `expr` также завершается со значением `1+6=7`.

Мы вернулись в `mult`, и в `curlex` должна быть закрывающая скобка, что и имеет место быть. Мы делаем `nextlexem` и возвращаемся со значением `7` в `item`. Дальше попробуйте смоделировать работу программы самостоятельно (кажется, что это легче проделать самому, чем про это читать).

Заметьте, что весь подсчет происходит строго в соответствии с нашим определением.

Обобщение

Что же делать, если в нашем выражении присутствуют вычитание и деление, возведение в степень, унарные минусы, дробные числа, символы переменных и т. д.? Это потребует изменения определения того, что же является выражением, а соответственно, и вычисляющих его значение функций. Покажем, например, как изменится определение выражения, а соответственно, и фукнция `expr`, если добавить еще и вычитание.

```
<Выражение> ::= <слагаемое> {(+|-) <слагаемое>}
```

Еще раз напомним, что вертикальная черта обозначает «или».

```
function expr:longint;
var a,b:longint;
    c:TLexem;
begin
a:=item;
while (curlex=_Plus) or (curlex=_Minus) do begin
  c:=curlex;
  nextlexem;
  b:=item;
  if c=_Plus then a:=a+b else a:=a-b;
  end;
expr:=a;
end;
```

Деление, естественно, добавляется в определение слагаемого (там же, где умножение). Полезно не забыть, выполняя деление, проверить, что делитель не равен 0. При появлении вещественных чисел нужно не забыть

изменить тип результата, возвращаемого функциями, и типы переменных для промежуточных значений, а также добавить разбор дробных чисел в процедуру `nextlexem`.

Добавление переменных и вызовов функций отразится на вычислении множителя (в функции `mult` в операторе `case` добавятся новые ветви). Например, если в выражении возможны переменные, то можно завести специальный тип лексем — переменная, а в переменную `vl` в этом случае записывать числовой номер этой переменной (храня отдельный массив соответствий между именами переменных и их номерами). Немного иначе добавляется операция возведения в степень — она потребует добавления нового правила в грамматику.

Метод рекурсивного спуска можно использовать и при вычислении логических выражений, и даже для разбора программы на языке Паскаль (хотя грамматика языка в этом случае будет значительно сложнее). Однако стоит заметить, что бывают такие языки, для которых метод рекурсивного спуска не применим при разборе программ. Соответствующие примеры можно найти в литературе.

В заключение хотелось бы поблагодарить А. В. Чернова и В. М. Гуровица за ряд ценных замечаний по тексту этой статьи.

Литература

1. *Ахо А.*, *Сети Р.*, *Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Издательский дом «Вильямс», 2001.