ОЛИМПИАДА ШКОЛЬНИКОВ «ШАГ В БУДУЩЕЕ»

ТРИНАДЦАТАЯ НАУЧНАЯ КОНФЕРЕНЦИЯ МОЛОДЫХ ИССЛЕДОВАТЕЛЕЙ «ШАГ В БУДУЩЕЕ, МОСКВА»

Секция. Информатика и системы управления

Подсекция. Программное обеспечение и информационные технологии

Синтаксически управляемый анализатор выражений

Автор: Андросова Екатерина Евгеньевна
ГОУ СОШ «Лосиный остров» № 368
11 класс «А»

Научный руководитель: Михайлов Алексей Андреевич
Зам. Директора, учитель ИКТ
ГОУ СОШ «Лосиный остров» № 368

Москва, 2010

Оглавление

1. Постановка проблемы

1.1. Актуальность проблемы

Выражения играют важную роль практически во всех языках программирования. В качестве примера обратимся к синтаксическим определениям наиболее востребованных инструкций языка Паскаль, приведенных на сайте http://pascal.comsci.us/syntax/. В нотации EBNF (Расширенная форма Бэкуса-Наура) эти определения имеют вид:

<assignment-statement> ::= (<variable-reference> | <function-name>) ':=' <expression>

<if-statement> ::= 'IF' <expression> 'THEN' <statement>

<if-else-statement> ::= 'IF' <expression> 'THEN' <statement> 'ELSE' <statement>

<for-statement> ::= 'FOR' <variable-name> ':=' <expression> ('TO' | 'DOWNTO') <expression> 'DO' <statement>

<repeat-statement> ::= 'REPEAT' <statement-list> 'UNTIL' <expression>

<while-statement> ::= 'WHILE' <expression> 'DO' <statement>

Можно заметить, что во всех определениях встречается синтаксическая категория <expression>, т. е. выражение. Синтаксический анализ выражения составляет более половины трудозатрат на анализ перечисленных выше инструкций. Поэтому уметь обрабатывать выражения очень важно.

Выражения играют и самостоятельную роль (вне языка программирования). Разработаны и разрабатываются программы, с помощью которых вычисляются выражения, строятся графики функций, решаются уравнения, выполняется численное дифференцирование и интегрирование, осуществляется поиск экстремумов функций и т. п. В таких программах выражения вводятся в основном двумя способами. При первом из них создается подпрограмма, в теле которой записывается выражение. Затем имя этой подпрограммы передается в качестве фактического параметра библиотечной подпрограмме под видом процедурного типа или указателя на функцию. Естественно, что исходный код библиотечной подпрограммы пользователю неизвестен, а известен только ее интерфейс. Во втором способе выражение вводится в программу в виде символьной строки. Этот способ проще с точки зрения пользователя, но связан с дополнительными временными издержками, так как требуется выполнить грамматический разбор выражения и его интерпретацию.

Программы второго вида, как правило, ориентированы на конкретный синтаксис выражения. Фактически этот синтаксис "зашит" в код программы. Во многих случаях требуется настроить программу на определенный синтаксис. Типичными примерами таких программ являются обучающие и контролирующие программы. В качестве примера можно привести интерактивную проверяющую программу SQLCourse.com, с помощью которой проверяются знания и умения в области SQL программирования. В языке SQL выражения так же важны, как и в других языках программирования, так как именно с помощью фразы WHERE <search condition>, где <search condition> - это логическое выражение, выполняется выборка интересующих пользователя данных. Несмотря на то, что язык реляционных баз данных SQL стандартизован (смотрите стандарт ISO/IEC 9075-1:2003 (E)), каждый производитель СУБД придерживается некоторого его диалекта. Поэтому, работая с программой SQLCourse.com, прежде всего, необходимо настроиться на определенный тип СУБД (MS SQL, Oracle, DB2, Access, MySQL, PostgreSQL, Sybase и др.), т. е. на определенный диалект языка SQL. И только после этого можно формировать запросы и посылать их на проверку. Встроенный интерпретатор соответствующего диалекта языка SQL, проверит запрос и в случае его корректности выполнит в имитационном режиме.

Чтобы программа анализа и интерпретации выражений могла настраиваться на конкретный синтаксис, она должна располагать знаниями об этом синтаксисе. Синтаксис выражений может быть представлен в виде грамматики в форме BNF, EBNF или в какой либо другой форме, но достаточно строгой и не уступающей BNF. Описание грамматики или ее представление либо должно храниться в памяти программы, либо вводиться как исходные данные.

1.2. Анализ особенностей синтаксической структуры выражений

Неформально выражения определяются как конструкции, задающие правила вычисления по значениям констант и текущим значениям переменных других значений путем применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для группировки операций и операндов.

Формально грамматика выражений определяется в виде четверки $G = (N, \Sigma, P, S)$, где N – конечное множество *нетерминальных символов,* или *нетерминалов* (иногда называемых вспомогательными символами, синтаксическими переменными или понятиями); Σ - не пересекающееся с N конечное множество *терминальных символов,* или *терминалов;P* -

конечное подмножество множества (N U Σ)*N(N U Σ)* x (N U Σ)* (элемент (α,β) множества *P* называется *правилом* (или *продукцией)* и записывается в виде $\alpha \to \beta$); *S* - выделенный символ из N, называемый *начальным символом* (или *аксиомой).*

Грамматики выражений являются *контекстно-свободными,* т. е. каждое правило из *P* имеет вид $A \to \alpha$, где $A \in N$, $\alpha \in (N \cup \Sigma)^*$. Обычно грамматики выражений являются однозначными.

В качестве примера рассмотрим учебную грамматику G_0, определенную следующим образом:

$G_0 = (\{E, T, F\}, \{a, +, *, (,)\}, P, E)$, где множество правил *P* имеет вид:

$E \to E + T \mid T \; T \to T * F \mid F \; F \to (E) \mid a$

Нетерминальные символы здесь имеют следующий смысл: *E* – выражение, *T* – терм (или слагаемое), *F* – фактор (или множитель). Анализируя синтаксическую структуру правил вывода, можно сказать, что операция «*» старше операции «+», а выражение, заключенное в круглые скобки, выполняется прежде всего. Эту грамматику можно было бы упростить, исключив цепные правила и заменив все нетерминалы начальным символом грамматики *E*. Полученная грамматика будет иметь вид:

$G = (\{E\}, \{a, +, *, (,)\}, P', E)$, где множество правил *P'* будет таким:

$E \to E + E \mid E * E \mid (E) \mid a$

Очевидно, что грамматика *G* неоднозначна из-за наличия в ней правил $E \to E + E \mid E * E$. Другой недостаток грамматики *G* заключается в том операции «+» и «*» имеют один и тот же приоритет. Но эти недостатки не будут препятствием для многих алгоритмов грамматического разбора, если явно оговорить приоритет (старшинство) и ассоциативность операций.

Практически все языки программирования отталкиваются от этих определений и дополняют их новыми нетерминальными символами, знаками операций и расширяют понятие атомарного объекта, вводя определения идентификаторов и литералов (констант). Различие между грамматиками выражений в основном заключается в определении операций (обозначения операций, старшинство, ассоциативность, типы операндов, характер выполнения операций). Рассмотрим несколько примеров.

В языке Ада определяется шесть классов операций. Эти классы операций приведены ниже в порядке возрастания их старшинства.

```
логическая_операция ::= and or xor
операция_отношения ::= = /= < <= > >=
бинарная_аддитивная_операция ::= + - &
унарная_аддитивная_операция ::= + -
```

```
мультипликативная_операция ::= * / mod rem
операция_высшего_приоритета ::= ** abs not
```

В языке Алгол 60 определяется девять классов операций. Эти классы операций приведены ниже в порядке убывания их старшинства.

```
операция_высшего_приоритета ::= ^
мультипликативная_операция ::= * / div
аддитивная_операция ::= + -
операция_отношения ::= = ~= < <= > >=
отрицание ::= not
конъюнкция ::= and
дизъюнкция ::= or
импликация ::= imp
эквивалентность ::= eqv
```

В языке Basic в составе Rational Software Development Platform определяется восемь классов операций. Эти классы операций приведены ниже в порядке убывания их старшинства. В восьмом классе Boolean_operators собраны Булевские операции различного старшинства, и они перечислены в порядке убывания старшинства. Заметим, что состав операций в языке Basic незначительно отличается от состава операций в языке Алгол 60. Исключения составляют операции Mod и Xor. Однако операции Xor и Eqv имеют противоположные приоритеты.

```
Exponentiation ::= ^
Negation ::= -
Multiplication_and_division ::= * /
Integer_Division ::= \
Modulus ::= Mod
Addition_and_subtraction ::= + -
Comparison_operators ::= = <> < <= > >=
Boolean_operators ::= Not And Or Xor Eqv Imp
```

В языке Паскаль определяется четыре класса операций. Эти классы операций приведены ниже в порядке убывания их старшинства.

```
операция_отношения ::= = <> < <= > >=
аддитивная_операция ::= + - or xor
мультипликативная_операция ::= * / div mod and shl shr
операция_высшего_приоритета ::= not
```

В языках Modula-2, Oberon и Zonnon, являющихся продолжателями языка Паскаль, определены те же классы операций. Но есть и незначительные различия. Отсутствуют операции побитового сдвига shl и shr, операция неравно обозначается символом «#», операция логическое «И» обозначается символом «&», операция логическое «НЕ» обозначается символом «~».

Еще большее разнообразие операций можно обнаружить в языках C/C++ и Java. Многочисленность набора операций не дает возможности перечислить их все в данной работе. Но можно отметить, что в зависимости от стандарта все операции по признаку старшинства делятся на 18 классов.

Приведенные примеры демонстрируют то, что выражение, записанное с помощью одних и тех же операций, в разных языках либо будут давать различные результаты, либо будут признаваться ошибочными.

Помимо старшинства важной характеристикой операций является их ассоциативность. Из формального определения грамматики нельзя извлечь эту характеристику. Ассоциативность является семантическим понятием и обычно задается словесно, например, так: «все операции равного приоритета выполняются слева направо». Обычно так оно и бывает. Мало того, почти все операции являются левоассоциативными. Исключение, как правило, составляют унарные префиксные операции, унарные постфиксные операции и операции наивысшего приоритета, такие, как операция возведения в степень. Например, в языке C++ операции 3-го класса (унарные префиксные операции) и операции 16-го класса (операции присваивания) считаются правоассоциативными. К правоассоциативной отнесена также тернарная операция «?:». В некоторых источниках ассоциативность тернарной операции определяется как нейтральная.

Для того чтобы разрешать проблемы неоднозначности в грамматиках выражений и не закладывать («зашивать») в код синтаксического анализатора и интерпретатора такие характеристики, как приоритет и ассоциативность, необходимо дополнить формальное определение грамматики через атрибуты терминальных символов. Следует отметить, что метод атрибутных грамматик и атрибутных трансляций на сегодняшний день является самым продуктивным при реализации синтаксически управляемого перевода.

1.3. Анализ инструментальных средств построения трансляторов

Написание трансляторов обычно связано с большими усилиями и затратой большого количества времени. При создании некоторых его частей, особенно при разработке лексического анализатора и синтаксического анализатора, можно большую часть работы проделать автоматически. Для этих целей используются программные средства, называемые генераторами компиляторов или системами построения трансляторов.

В настоящее время на «рынке» средств автоматизации построения компиляторов имеется большой ассортимент программных продуктов, различающихся как своими

функциональными возможностями, так и другими характеристиками. Обширная информация по этому вопросу представлена по адресу http://catalog.compilertools.net .

Генераторы лексических и синтаксических анализаторов представляют наиболее широкий и развитый класс средств автоматизации построения компиляторов, хотя эти генераторы охватывают только две фазы.

Исходя из первоочередных задач разрабатываемой программы, особый интерес представляет класс генераторов лексических и синтаксических анализаторов. Анализ программ этого класса показывает, что большинство из них являются модификациями двух программ: генератора лексического анализатора LEX и генератора синтаксического анализатора YACC. Чтобы сузить круг рассматриваемых средств, определим требования, предъявляемых к генераторам. Эти требования должны соответствовать цели и задачам работы.

1. Генератор должен быть бесплатным, т. е. свободно распространяемым.
2. Генератор должен быть хорошо документирован.
3. Генератор должен быть хорошо апробирован.
4. Входные данные для генератора – грамматика языка в БНФ, расширенной БНФ или модифицированной БНФ.
5. Выходные данные генератора – код на языке Delphi.
6. Генератор должен быть представлен совместимой по входу и выходу парой генератор лексического анализатора – генератор синтаксического анализатора.
7. Генератор должен допускать задание семантических правил или семантических программ для проверки контекстно-зависимых условий, статических семантических зависимостей, обработки и нейтрализации ошибок.
8. Платформой для генератора должна быть операционная система WindowsXP.

Сравнительный анализ показал, что лучшими характеристиками в такой многокритериальной задаче (за исключением требования 1) обладают следующие программы: COCO/R, ANTLR, образовательная версия программы PCYACC, Flex в паре с Bison. Эти инструментальные средства оказались совершенно разными по своему внутреннему устройству. COCO/R, ANTLR – это генераторы нисходящего LL(1)-разбора, реализующие метод рекурсивного спуска. PCYACC и Bison – это генераторы восходящего LALR(1)-разбора. Основной недостаток этих инструментов – это их «тяжеловесность», т. е. они ориентированы на построение полноценных компиляторов с возможность генерации кода виртуальной машины, а не на трансляцию таких ограниченных конструкций, как выражения.

2. Цель и содержание работы

Разработать программу синтаксически управляемого анализатора выражений, который в автоматическом режиме может настраиваться на конкретную грамматику выражений.

Для достижения поставленной цели необходимо решить следующие основные задачи:

1. Определить формат входной грамматики и способ ее представления в памяти программы (машины).
2. Разработать и реализовать алгоритм синтаксического разбора (анализа), легко адаптируемого ко входной грамматике.
3. Разработать и реализовать алгоритм построения управляющих структур синтаксического анализатора.
4. Выбрать способ формального описания лексем и трансформацию этих описаний в код лексического анализатора.
5. Выбрать способ внутреннего представления выражения после синтаксического анализа, реализовать метод его получения на основе синтаксически-управляемых определений и реализовать алгоритм интерпретации этого представления.
6. Разработать Windows приложение в среде Code Gear RAD Studio (Delphi 2006), реализующее весь комплекс разработанных алгоритмов.
7. Проверить работоспособность разработанной программы и оценить эффективность принятых решений.

3. Методы решения проблемы

3.1. Выбор метода грамматического разбора выражений

Среди простых методов грамматического разбора выражений, не требующих привлечения всей «мощи» аппарата теории синтаксического анализа, перевода и компиляции (LL(k) и LR(k) детерминированный разбор), можно выделить три метода, иногда называемых «ранними» методами. Это метод Рутисхаузера [4], метод Бауэра-Замельзона [4] и метод Дейкстры [5].

Особенностью метода Рутисхаузера является предположение о полной скобочной структуре выражения. Под полной скобочной записью выражения понимается запись, в которой порядок действий задается расстановкой скобок. Неявное старшинство операций при этом не учитывается. Обрабатывая выражение, алгоритм присваивает каждому символу из входной цепочки номер уровня по следующему правилу:

```
if  это открывающая скобка or переменная then
   значение увеличивается на 1
else if  это знак операции or закрывающаяся скобка then
   значение уменьшается на 1;
```

Алгоритм Рутисхаузера.

Вход. Выражение *w* (полная скобочная запись);

Выход. Результат вычисленного выражения.

Метод.

```
begin
  Выполнить расстановку уровней;
  repeat
    выполнить поиск элементов с максимальным значением уровня;
    выделить тройку (два операнда с максимальным значением уровня и операцию,
    которая заключена между ними);
    результат вычисления тройки обозначить вспомогательной переменной;
    из входной ленты удалить выделенную тройку вместе с ее скобками;
    на ее место поместить вспомогательную переменную, обозначающую результат,
    со значением уровня на единицу меньше, чем у выделенной тройки;
  until во входной строке не останется одна переменная, обозначающая общий
        результат выражения
end
```

Метод Рутисхаузера неприемлем для решения нашей проблемы, так как совершенно неясно, как он зависит от грамматики входного языка.

Алгоритм Бауэра-Замельзона основан на использовании *стека* и *таблицы переходов*. Пусть имя стека транслятора T, а имя стека интерпретатора E. Тогда транслятор прочитывает выражение один раз слева направо и вырабатывает последовательность команд двух видов: K_{Opr} (где Opr – операнд) является командой «Выбрать операнд Opr и заслать его в стек E».

K_{Op} (где Op – бинарная операция) является командой «Выбрать два верхних операнда из стека E, произвести над ними операцию Op и занести результат в вершину стека E».

Когда при просмотре входной строки считываемый символ является операндом Opr, выдается команда K_{Opr} и транслятор переходит к следующему символу. Когда считываемый символ является операцией Op, производится одно из нескольких фиксированных действий. Как правило, либо Op засылается в стек T, либо выдается команда K_{Op}. В таблице переходов указывается, какие действия должен выполнять транслятор. В этой таблице каждой операции языка соответствует строка и столбец. Элементами таблицы являются директивы транслятору. Возможны 4 действия транслятора после прочтения операцииOp. (Операция в вершине стека T обозначается <Op>).

I	Заслать Op в T; Читать следующий символ.
II	Генерировать $K_{<Op>}$; Заслать Op в T; Читать следующий символ
III	Читать из T; Читать следующий символ (используется для удаления скобок).
IV	Генерировать $K_{<Op>}$; Читать из T; Повторить с тем же самым входным символом.

Имеются еще две специальные команды (*Конец* и *Ошибка)*, которые приказывают транслятору остановиться и выдать сообщение об ошибке. Транслятор выбирает элементы таблицы, используя в качестве одного индекса <Op> (верхнюю операцию стека T), и в качестве второго индекса Op (операцию, прочитанную последней из входной строки).

	Пусто	(	+	-	*	/	)
Пусто	*Конец*	I	I	I	I	I	*Ошибка*
(	*Ошибка*	I	I	I	I	I	III
+	IV	I	II	II	I	I	IV
-	IV	I	II	II	I	I	IV
*	IV	I	IV	IV	II	II	IV
/	IV	I	IV	IV	II	II	IV

Например, для входной строки (A*B+C*D)/(A-D)+B*C будет получена последовательность команд $K_A, K_B, K_*, K_C, K_D, K_*, K_+, K_A, K_D, K_-, K_/, K_B, K_C, K_*, K_+$

Впрочем, более простая запись получается, если опускать все K: AB*CD*+AD-/BC*+

Эта более простая запись программы является постфиксной польской записью.

Метод Бауэра-Замельзона привлекателен тем, что он таблично управляемый и, следовательно, каким-то образом зависит от грамматики входного языка, а на выходе фактически получается постфиксной польской записью. Но четкого алгоритма получения таблицы нет.

Метод Дейкстры – это самый известный метод преобразования инфиксного выражения в постфиксную польскую запись.

Алгоритм Дейкстры.

Вход. Инфиксное выражение *w* с правым концевым маркером $, приоритеты и ассоциативность всех операций.

Выход. Если *w* корректное выражение, то постфиксная польская записи, в противном случае сообщение об ошибке.

Метод. Будем считать, что $ - маркер дна стека и стек пуст. Пусть *a* текущий входной символ, а *b* верхний символ стека.

```
while a <> $ do
begin
  //читаем очередной символ;
  if a – атом (константа или идентификатор) then
    записать a на выходную ленту
  else if a - идентификатор функции then
    перенести его со входа в стек
  else if a = "(" then
    перенести его со входа в стек
  else if a = ")" then
  begin
    while b <> "(" do
      вытолкнуть b из стека и записать на выходную ленту;
    if b = $ then
      error()
    end
  else if a - оператор, например, OP then
  begin
    if OP левоассоциативный then
      while приоритет OP ≤ приоритета b do
        вытолкнуть b из стека и записать на выходную ленту;
    else if OP правоассоциативный then
      while приоритет OP < приоритета b do
        вытолкнуть b из стека и записать на выходную ленту;
    перенести оператор OP со входа в стек;
  end;
end;
вытолкнуть все символы из стека в выходную строку;
В стеке должны были остаться только символы операторов; если это не так, значит
в выражении несогласованы скобки.
```

Как и в случае алгоритма Рутисхаузера алгоритм Дейкстры не таблично управляемый, но за «фасадом» этого алгоритма просматривается разбор, основанный на операторном предшествовании. Поэтому имеет смысл сразу обратиться к этому методу.

Грамматики предшествования являются подклассом LR(k)-грамматик, которые анализируются легко реализуемым алгоритмом типа «перенос-свертка».

Алгоритмом типа «перенос-свертка» для грамматики $G = (\mathbf{N}, \Sigma, P, S)$ называют пару функций $\mathbf{A} = (f, g)$, где f – *функция переноса*, а g – *функция свертки*. Функции f и g определяются так:

1) f отображает $(\Sigma \cup \mathbf{N} \cup \{\$\})^* \times (\Sigma \cup \{\$\})^*$ во множество {Перенос, Свертка, Ошибка, Допуск}.
2) g отображает $(\Sigma \cup \mathbf{N} \cup \{\$\})^* \times (\Sigma \cup \{\$\})^*$ во множество $\{1, 2, \ldots, p,$ ошибка$\}$ при условии, что если $g(\alpha, w) = i$, то правая часть i-го правила является суффиксом цепочки α.

Алгоритм типа «перенос-свертка» использует входную ленту, читаемую слева направо, и стек разбора. На основе того, что находится в стеке и осталось необработанным на входе, функция f решает, перенести ли текущий входной символ в стек или вызвать процедуру свертки. Если принимается последнее из этих решений, то функция g решает, какую сделать свертку.

Существует несколько видов грамматик предшествования, которые различаются по следующим признакам: a) какие отношения предшествования в них определены; b) между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения, c) как локализуется левый конец сворачиваемой основы и определяется нужная свертка. По этим признакам выделяют следующие виды грамматик предшествования: a) простого предшествования, b) расширенного предшествования, c) слабого предшествования, d) смешанной стратегии предшествования, e) операторного предшествования.

Техника анализа, основанная на операторном предшествовании, использует для выделения основы правовыводимой цепочки $\alpha\beta w$ три отношения предшествования Вирта-Вебера.

В общем случае *отношения предшествования Вирта-Вебера* $\lessdot\bullet$, $=\bullet$ и $\bullet\gtrdot$ для КС-грамматики $G = (N, \Sigma, P, S)$ определяются на множестве $(N \cup \Sigma)$ следующим образом:

1) $X \lessdot\bullet Y$, если в P есть такое правило $A \to \alpha XB\beta$, что $B \Rightarrow^+ Y\gamma$;
2) $X =\bullet Y$, если в P есть правило $A \to \alpha XY\beta$;

3) отношение •> определяется на (N ∪ Σ) x Σ, так как непосредственно справа от основы в правовыводимой цепочке может быть только терминальный символ; полагаем X •> a, если в P есть правило $A \to \alpha BY\beta$, $B \Rightarrow+ \gamma X$ и $Y \Rightarrow* a\delta$ (заметим, что $Y = a$, если $Y \Rightarrow 0\ a\delta$).

Если грамматика является приведенной (без циклов, без ε-правил и без бесполезных символов) и в правых частях правил не содержатся смежные нетерминалы, то грамматика G называется *операторной грамматикой*.

Для операторной грамматики отношения предшествования можно задать на множестве терминалов плюс символ $ (используется в качестве маркера дна стека и правого концевого маркера входной цепочки), игнорируя нетерминалы, следующим образом:

1) a =• b, если $A \to \alpha a\gamma b\beta \in P$ и $\gamma \in \mathbf{N} \cup \{\varepsilon\}$,
2) a <• b, если $A \to \alpha aB\beta \in P$ и $B \Rightarrow+ \gamma b\delta$, где $\gamma \in \mathbf{N} \cup \{\varepsilon\}$,
3) a •> b, если $A \to \alpha Bb\beta \in P$ и $B \Rightarrow+ \delta a\gamma$, где $\gamma \in \mathbf{N} \cup \{\varepsilon\}$,
4) $ <• a, если $S \Rightarrow+ \gamma a\delta$ и $\gamma \in \mathbf{N} \cup \{\varepsilon\}$,
5) a •> $, если $S \Rightarrow+ \alpha a\gamma$ и $\gamma \in \mathbf{N} \cup \{\varepsilon\}$.

Если между любыми двумя терминальными символами выполняется не более одного отношения операторного предшествования, то операторная грамматика G называется *грамматикой операторного предшествования*.

С помощью алгоритма разбора типа «перенос-свертка» легко выделить терминальные символы, входящие в основу. Однако возникают проблемы в связи с нетерминальными символами, поскольку на них не определены отношения операторного предшествования. Для грамматик операторного предшествования алгоритм типа «перенос-свертка» может строить «остовный» правый разбор. Для этого необходимо выполнить преобразование грамматики, а именно заменить все нетерминалы одним нетерминалом и устранить цепные правила. Например, для грамматики операторного предшествования

$G_0 = (\{E, T, F\}, \{a, +, *, (,)\}, P, E)$, где

$P = \{E \to E + T : (1),\ E \to T : (2),\ T \to T * F : (3),\ T \to F : (4),\ F \to (E) : (5),\ F \to a : (6)\}$

преобразованная грамматика будет иметь вид

$G = (\{E\}, \{a, +, *, (,)\}, P', E)$, где

$P' = \{ E \to E + E : (1),\ E \to E * E : (3),\ E \to (E) : (5),\ E \to a : (6)\}$

Грамматика G, очевидно, не однозначная, но отношения операторного предшествования гарантируют единственность искомого разбора

Для входной цепочки $(a+a)*a$ алгоритм проделает такую последовательность тактов:

[$, (*a*+*a*)***a*$, ε] $|-^{s}$ [$(, *a*+*a*)***a*$, ε] $|-^{s}$ . . . $|-^{r}$ [$*E*, $, 661563] |– **допуск.**

Заметим, что можно было бы дополнить дерево остовного разбора так, чтобы получилось соответствующее дерево в грамматике G_0. Но на практике в этом нет необходимости. Цель построения дерева - трансляция, а естественный перевод нетерминалов *E*, *T* или *F* грамматики G_0 - это программа машины, вычисляющая выводимое из него выражение. Поэтому если применяется правило $E \to T$ или $T \to F$, то перевод правой части будет тот же, что и левой части.

Алгоритм. Построение анализатора, использующего операторное предшествование. *Вход*. Грамматика операторного предшествования $G = (N, \Sigma, P, S)$. *Выход*. Алгоритм разбора **A** = (*f*, *g*) типа "перенос-свертка" для грамматики G_s. *Метод*. Пусть β обозначает *S* или ε.

1) $f(a\beta, b)$ = **Перенос**, если $a <\bullet b$ или $a =\bullet b$.
2) $f(a\beta, b)$ = **Свертка**, если $a \bullet> b$.
3) $f(\$S, \$)$ = **Допуск**.
4) $f(\alpha, w)$ = **Ошибка** в остальных случаях.
5) $g(a\beta b\gamma, w) = i$, если
   a) β - это *S* или ε,
   b) $a <\bullet b$, отношение $=\bullet$ выполняется между последовательными терминальными символами цепочки γ, если они существуют,
   c) $S \to \beta b\gamma$ - правило с номером *i* грамматики G_s
6) $g(\alpha, w)$ - **Ошибка** в остальных случаях.

После уточнения алгоритма разбора возникает вопрос, как вычислить матрицу отношений операторного предшествования по КС-грамматике. На практике матрицу предшествования сложно построить, опираясь непосредственно на определения отношений предшествования. Удобнее воспользоваться двумя дополнительными множествами — множеством крайних левых и множеством крайних правых символов относительно нетерминальных символов грамматики. Этот способ подробно описан в литературе [2] и мы принимаем его в качестве одного из возможных кандидатов на реализацию.

Для языка арифметических выражений в работе [3] предлагается использовать следующую эвристику для создания корректного множества отношений операторного предшествования. Пусть M – матрица отношений операторного предшествования порядка |Σ|

+ 1, Op, Op1, Op2 – операторы, имеющие предопределенный атрибут priority (приоритет) и элементы матрицы индексируются терминальными символами и маркерами $.

Правило 1.
```
if Op1.priority > Op2.priority then begin
   M[Op1, Op2] := >; M[Op2, Op1] := <;
end;
```

Правило 2.
```
if Op1.priority = Op2.priority then
  if Op1 и Op2 левоассоциативны then begin
     M[Op1, Op2] := >; M[Op2, Op1] := >;
end else if Op1 и Op2 правоассоциативны then begin
       M[Op1, Op2] := <; M[Op2, Op1] := <
end else;
```

Правило 3.
```
forall Op do begin
   M[Op, id] := <; M[id, Op] := >; M[Op, (] := <;
   M[(, Op] := <; M[), Op] := >; M[Op, )] := >;
   M[Op, $] := >; M[$, Op] := <;
end;
Кроме того, считаем, что begin
   M[(, )] := =; M[$, (] := <; M[$, id] := <;
   M[(, (] := <; M[id, $] := >; M[),$] := >;
   M[Op, id] := <; M[id, )] := >; M[), )] := >;
end;
```

Эта эвристика так же может быть проверена.

Метод операторного предшествования вполне подходит для решения обозначенной проблемы, так как он определяется грамматикой, таблично управляемый, легко реализуемый. Очень важно то, что в основе этого метода лежит алгоритм типа «перенос-свертка», для которого нетрудно сформулировать семантические правила перевода во внутреннюю форму типа абстрактное синтаксическое дерево, префиксная польская запись, постфиксная польская запись, код виртуальной машины [3].

3.2. Разработка блока чтения и обработки грамматики

Чтобы выполнить генерацию управляющей таблицы разбора в виде матрицы отношений предшествования в автоматическом режиме, а не ручным способом, необходимо прочитать грамматику и придать ей такую структуру, которая будет удобной для последующей работы. Если грамматика будет представлена в виде обычного текстового

файла, то программа чтения такой грамматики по своей сложности будет подобна небольшому компилятору со своим собственным лексическим и синтаксическим анализаторами. Придание текстовому документу нужной структуры – одна из задач XML-технологии.

Существует две стандартных технологии манипулирования документами XML: первая подразумевает использование DOM (Document Object Model), вторая основана на использовании SAX (Simple API *for* XML). Суть этих подходов в следующем:

- DOM (Document Object Model) загружает весь документ в память в виде иерархического дерева узлов. Имеется возможность прочитать любой из узлов и выполнить в его отношении некоторые процедуры, включая операции редактирования. Этот подход применяется к данным небольшого объема.
- SAX (Simple API *for* XML) не предусматривает загрузку документа в память. Вместо этого данный механизм самостоятельно осуществляет грамматический разбор XML-кода и генерирует событие для каждого логического элемента XML-кода. Когда грамматический разбор документа при помощи SAX завершается, документ бесследно исчезает из системы и дальнейший доступ к нему без повторной загрузки невозможен. Этот подход применяется к данным, извлекаемым из баз данных.

Так как объем грамматики невелик и при ее обработке необходимо будет перемещаться по структуре XML-документа, редактировать ее, а также создать новый XML-документ с нуля, то удобнее применять DOM.

В Delphi можно использовать несколько различных реализаций DOM. Первая реализация разработана Microsoft и входит в состав пакета MSXML SDK. Эта реализация требует подключения специальной библиотеки. Вторая реализация включена в состав компонента-оболочки под названием TXMLDocument.

Работать с компонентом TXMLDocument можно на одном из двух уровней:

- Низкий уровень предусматривает использование свойства DOMDocument. Это свойство принадлежит к типу IDOMDocument и обеспечивает доступ к стандартному интерфейсу W3C Document Object Model. Официальный интерфейс DOM определяется в модуле xmldom и включает в себя такие интерфейсы, как IDOMNode, IDOMNodeList, IDOMAttr, IDOMElement и IDOMText.
- На более высоком уровне можно воспользоваться интерфейсом IXMLDocument компонента TXMLDocument. Этот интерфейс подобен стандартному интерфейсу

Поскольку высокоуровневый интерфейс, упрощает выполнение некоторых операций DOM, так как заменяет множественные последовательно выполняемые обращения к стандартным вызовам DOM единственным методом или свойством, то в работе используется именно этот подход.

Таким образом, мы представляем BNF- или EBNF-грамматику в виде XML-документа. Пример такого документа для грамматики G_0 изображен ниже.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<grammar name="G0">
  <terminalsymbols>
    <term type="atom" name="ident" spell="a" />
    <term type="operator" name="add" priority="2" assoc="left" spell="+" />
    <term type="operator" name="mul" priority="1" assoc="left" spell="*" />
    <term type="punctuator" name="lparen" spell="(" />
    <term type="punctuator" name="rparen" spell=")" />
  </terminalsymbols>
  <nonterminalsymbols>
    <nonterm name="E" />
    <nonterm name="T" />
    <nonterm name="F" />
  </nonterminalsymbols>
  <productions>
    <production>
      <lhs name="E" />
      <rhs>
        <symbol type="nonterm" name="E" />
        <symbol type="term" name="add" />
        <symbol type="nonterm" name="T" />
      </rhs>
    </production>
    <production>
   <lhs name="E" />
   <rhs>
     <symbol type="nonterm" name="T" />
      </rhs>
    </production>
    <production>
      <lhs name="T" />
      <rhs>
        <symbol type="nonterm" name="T" />
        <symbol type="term" name="mul" />
        <symbol type="nonterm" name="F" />
      </rhs>
    </production>
    <production>
      <lhs name="T" />
      <rhs>
        <symbol type="nonterm" name="F"/>
      </rhs>
    </production>
    <production>
```

```
      <lhs name="F" />
      <rhs>
        <symbol type="nonterm" name="ident" />
      </rhs>
    </production>
    <production>
      <lhs name="F" />
      <rhs>
        <symbol type="term" name="lparen" />
        <symbol type="nonterm" name="E" />
        <symbol type="term" name="rparen" />
      </rhs>
    </production>
  </productions>
  <startsymbol name="E" />
</grammar>
```

Основную часть этого документа составляют правила вывода. Но существенная роль отводится и терминальным символам, для которых определены такие атрибуты, как тип (`type`), название (`name`), приоритет (`priority`), ассоциативность (`assoc`) и побуквенное изображение (`spell`). Обязательными атрибутами являются только тип и название. Для некоторых терминалов (идентификаторов, числовых и Булевских литералов) невозможно указать их побуквенное изображение. Для этих целей предусматривается механизм регулярных выражений. В данном примере это не показано. В простейшем случае определения лексем в виде регулярных выражений «зашиваются» в коде лексического анализатора.

Эта грамматика загружается из файла в память в виде объекта `XMLDocument1`, а затем структура документа отображается при помощи объекта `TreeView1`. Сказанное демонстрируется следующей процедурой:

```
procedure TFormXmlTree.btnLoadClick(Sender: TObject);
begin
  OpenDialog1.InitialDir := ExtractFilePath (Application.ExeName);
  if OpenDialog1.Execute then
  begin
    XMLDocument1.LoadFromFile(OpenDialog1.FileName);
    TreeView1.Items.Clear;
    DomToTree (XMLDocument1.DocumentElement, nil);
    TreeView1.FullExpand;
  end;
end;
```

Процесс отображения объекта `XMLDocument1` в объект `TreeView1` осуществляется с помощью вспомогательной процедуры `DomToTree`, которая использует для этого интерфейс `IXMLNode` и объект типа `TTreeNode`. Данная процедура является еще одним наглядным

примером того, насколько выгодно использовать XML-документ вместо обычного текстового файла.

Для упрощения работы с грамматикой разработан класс TGrammar, который под видом свойств хранит компоненты грамматики в виде объектов типа TStringList. В модуле UnitGrammar предварительно определены перечислимые типы, значения которых используются в XML-документе грамматики:

```
type  TTerminalKind = (Atom, Operator, Punctuator);
type  TAssocType = (Left, Right, Undef);
type  TSymbolType = (Term, Nonterm);
```

Возникает естественный вопрос, как гарантировать соответствие элементов XML-документа набору правил, обеспечивающих "допустимость" данных. Допустимость XML-документа не связана напрямую с синтаксической правильностью его XML-элементов (например, с требованием о том, что все открывающие элементы должны иметь закрывающие их дескрипторы). Допустимость документов связана с правилами форматирования (например, поле name должно быть атрибутом и не вложенным элементом), которые обычно задаются XSD-схемой или DTD-файлом.

С помощью XSD-генератора Microsoft была создана XSD-схема для XML-документа, описывающего грамматику. Эту схему необходимо было ручным способом скорректировать, чтобы изменить значения, определяемые по умолчанию, и ввести определения перечислимых типов. Ниже приведена эта схема.

```
   <?xml version="1.0" encoding="utf-8"?>
   <xs:schema id="NewDataSet" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-
com:xml-msdata">
     <xs:element name="grammar">
       <xsd:simpleType name="assoc">
         <xsd:restriction base="xsd:staring">
        <xsd:enumeration value="left" />
        <xsd:enumeration value="right" />
        <xsd:enumeration value="undef" />
         </xsd:restriction>
       </xsd:simpleType>
       <xsd:simpleType name="type">
         <xsd:restriction base="xsd:staring">
           <xsd:enumeration value="atom" />
        <xsd:enumeration value="operator" />
        <xsd:enumeration value="punctuator" />
         </xsd:restriction>
       </xsd:simpleType>
       <xs:complexType>
         <xs:sequence>
           <xs:element name="terminalsymbols" minOccurs="1" maxOccurs="1">
             <xs:complexType>
               <xs:sequence>
```

```
                <xs:element name="term" minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:attribute name="type" type="xs:string" use="required"/>
                    <xs:attribute name="name" type="xs:string" use="required"/>
                    <xs:attribute name="spell" type="xs:string" />
                    <xs:attribute name="priority" type="xs:string" />
                    <xs:attribute name="assoc" type="xs:string" />
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="nonterminalsymbols" minOccurs="1" maxOccurs="1">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="nonterm" minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:attribute name="name" type="xs:string" />
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="productions" minOccurs="1" maxOccurs="1">
            <xsd:simpleType name="type">
             <xsd:restriction base="xsd:staring">
              <xsd:enumeration value="term" />
              <xsd:enumeration value="nonterm" />
             </xsd:restriction>
            </xsd:simpleType>
            <xs:complexType>
              <xs:sequence>
                <xs:element name="production" minOccurs="1"
maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element name="lhs" minOccurs="1" maxOccurs="1">
                        <xs:complexType>
                          <xs:attribute name="name" type="xs:string"
use="required"/>
                        </xs:complexType>
                      </xs:element>
                      <xs:element name="rhs" minOccurs="1" maxOccurs="1">
                        <xs:complexType>
                          <xs:sequence>
                            <xs:element name="symbol" minOccurs="1"
maxOccurs="unbounded">
                              <xs:complexType>
                                <xs:attribute name="type" type="xs:string"
use="required"/>
                                <xs:attribute name="name" type="xs:string"
use="required"/>
                              </xs:complexType>
                            </xs:element>
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
```

```
            </xs:complexType>
          </xs:element>
          <xs:element name="startsymbol" minOccurs="1" maxOccurs="1">
            <xs:complexType>
              <xs:attribute name="name" type="xs:string" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" />
      </xs:complexType>
    </xs:element>
    <xs:element name="NewDataSet" msdata:IsDataSet="true"
msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="grammar" />
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

Эта схема может рассматриваться как шаблон для формирования правильного XML-документа грамматики.

3.3. Разработка блока лексического анализа

Основная задача лексического анализа – разбить входной текст, состоящий из последовательности одиночных символов, на последовательность минимально значимых единиц текста языка, называемых лексемами. В общем случае работа лексического анализатора описывается формализмом конечных автоматов. Однако непосредственное описание конечного автомата неудобно практически. Поэтому для описания лексических анализаторов, как правило, используют либо формализм регулярных выражений, либо формализм автоматных, или регулярных, грамматик. Во втором случае требуется дополнительное преобразование грамматики к регулярным выражениям с помощью алгоритма решения стандартной системы уравнений с регулярными коэффициентами. Последний подход признан нами очень сложным и за основу взяты регулярные выражения.

К сожалению в Delphi нет встроенного модуля/компонента для работы с регулярными выражениями. Это существенное упущение разработчиков Delphi. В качестве решения на форумах предлагают воспользоваться либо библиотекой Perl Compatible Regular Expressions (PCRE), либо модулем TRegExpr с официального сайта http://www.regexpstudio.com, либо

обратиться к пространству имен System.Text.RegularExpressions для Delphi *for .NET*. И то и другое на данный момент является проблематичным.

Было принято такое решение. Для каждого класса лексем записать регулярноевыражение, порождающее все лексемы данного класса. При записи регулярныхвыражений использовались следующие соглашения:

1. Любой символ соответствует самому себе, если только он не является метасимволом со специальным значением (такими метасимволами являются \, |, (,), [, {, *, +, ^, $, ? и .).
2. Любой метасимвол можно "защитить", то есть рассматривать его как обыкновенный символ, поставив перед метасимволом обратную косую черту \.
3. Символы могут быть сгруппированы в классы - совокупности символов, заключенные в квадратные скобки [и]. Можно указывать как отдельные символы, так и их диапазон (диапазон задается двумя крайними символами, соединенными тире).
4. Если сразу после открывающей квадратной скобки стоит символ ^, то смысл меняется на противоположный.
5. Для задания альтернативных шаблонов используется символ | как разделитель.
6. Чтобы было ясно, где начинается и где заканчивается набор альтернативных шаблонов, их заключают в круглые скобки.
7. Чтобы указать, что тот или иной шаблон может повторяться используются квантификаторы: * (ноль или несколько совпадений), + (одно или несколько совпадений), ? (ноль совпадений или одно совпадение), {n} (ровно n совпадений), {n,} (по крайней мере n совпадений), {n,m} (от n до m совпадений).
8. \d используется для сокращение для [0-9], \w используется для сокращения [A-Za-z0-9_], \D – отрицание \d, \W – отрицание \w.

   В качестве примера приведем регулярные выражения трех типов лексем

   token Int \d+

   token Real ((\d+\.)|(\.\d+)|(\d+\.\d+))([Ee]([+-])?\d+)?)|(\d+[Ee]([+-])?\d+)

   token Ident [A-Za-z_]\w*.

Далее для каждого регулярного выражения строилась стилизованная блок-схема, называемая диаграммой переходов. Состояния в диаграмме переходов изображались кружками и последовательно нумеровались. Состояния соединялись дугами, которые имели метки, указывающие символы, которые могут появиться во входном потоке по достижении состояния. Метка other означала появление любого символа, не указанного другими

исходящими дугами. На рис. 1 приведены диаграммы переходов, соответствующие регулярным выражениям.

Диаграмма переходов соответствуют диаграмме детерминированного конечного автомата, имеющее одно начальное состояние и несколько заключительных. При попадании в некоторое состояние осуществляется считывание очередного входного символа и, если имеется исходящая дуга с меткой, соответствующей этому символу, выполняется переход в следующее состояние. Если такой дуги нет, то входящий символ некорректен и произошла ошибка.

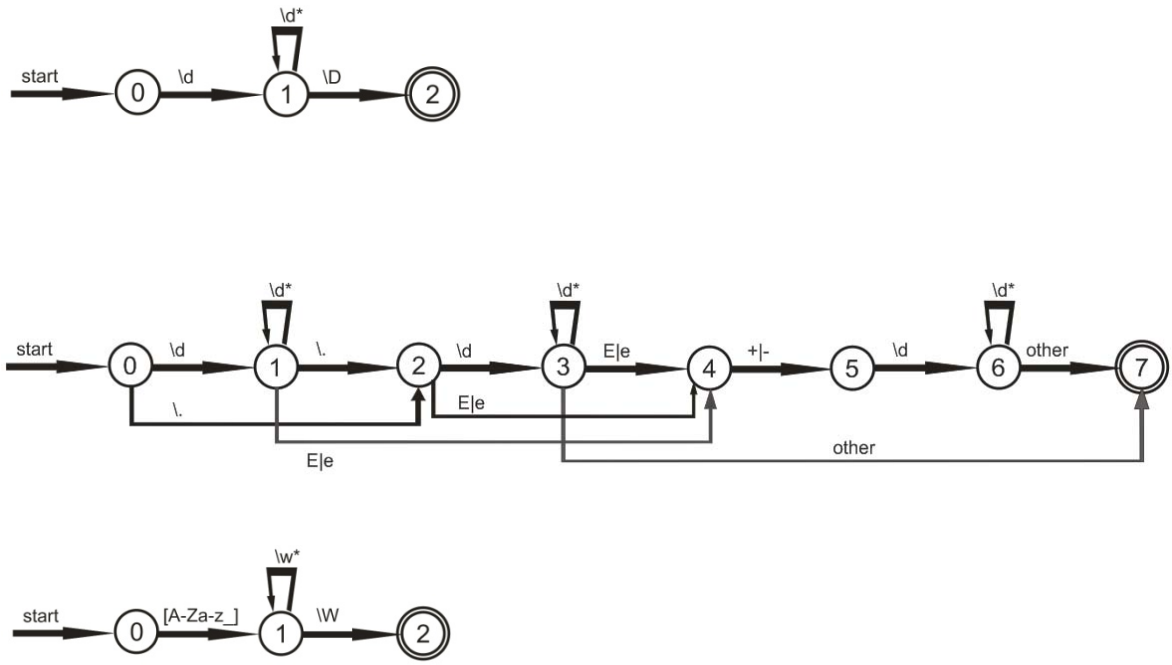

Рис. 1. Диаграммы переходов

Лексема для данного класса лексем должна быть максимально возможной длины. Например, лексический анализатор не должен останавливаться после того, как встретит 987 или даже 987.6, если введено число 987.6e-5.

Последовательность диаграмм переходов была преобразована в функцию GetNextToken (основной метод класса TScanner) поиска классов лексем, определяемых диаграммами переходов.

Ключевые слова распознаются так: сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов. Если да, то выдается признак соответствующего ключевого слова. Аналогичным образом распознаются имена

стандартных функций. Если два раза подряд выдается ответ нет, то возвращается признак идентификатора, а сам идентификатор и его атрибуты помещается в таблицу символов.

Лексический анализатор работает как самостоятельная фаза трансляции, и его выходом является список токенов, сохраняемый в файле в бинарном формате и в формате XML. Например, при сканировании выражения `a+b*(c+d*(e+f))` на выходе лексического анализатора будет получен поток токенов, представленный в виде следующего XML-документа:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<TokenList>
  <Token Type="Identifier" Value="a" Line="1" StartPos="1" EndPos="1" />"
  <Token Type="Operator" Value="+" Line="1" StartPos="2" EndPos="2" />"
  <Token Type="Identifier" Value="b" Line="1" StartPos="3" EndPos="3" />"
  <Token Type="Operator" Value="*" Line="1" StartPos="4" EndPos="4" />"
  <Token Type="LParen" Value="(" Line="1" StartPos="5" EndPos="5" />"
  <Token Type="Identifier" Value="c" Line="1" StartPos="6" EndPos="6" />"
  <Token Type="Operator" Value="+" Line="1" StartPos="7" EndPos="7" />"
  <Token Type="Identifier" Value="d" Line="1" StartPos="8" EndPos="8" />"
  <Token Type="Operator" Value="*" Line="1" StartPos="9" EndPos="9" />"
  <Token Type="LParen" Value="(" Line="1" StartPos="10" EndPos="10" />"
  <Token Type="Identifier" Value="e" Line="1" StartPos="11" EndPos="11" />"
  <Token Type="Operator" Value="+" Line="1" StartPos="12" EndPos="12" />"
  <Token Type="Identifier" Value="f" Line="1" StartPos="13" EndPos="13" />"
  <Token Type="RParen" Value=")" Line="1" StartPos="14" EndPos="14" />"
  <Token "15" Type="RParen" Value=")" Line="1" StartPos="15" EndPos="15" />"
</TokenList>
```

Этот поток токенов вместе с таблицей идентификаторов становится доступным синтаксическому анализатору.

3.4. Разработка блока синтаксического анализа и интерпретации

Синтаксический анализатор воспринимает выход лексического анализатора в виде потока токенов (исследуются, как правило, только первые компоненты токенов - их типы) и разбирает его в соответствии с входной грамматикой. В процессе разбора устанавливается, удовлетворяет ли цепочка лексем структурным условиям, явно сформулированным в определении синтаксиса языка. Выходом синтаксического анализатора является абстрактное синтаксическое дерево и его линейное представление в виде постфиксной польской записи. Если исключить такие технические детали как построение дерева, то процедуры грамматического разбора оказались на порядок проще основного метода класса TScanner GetNextToken. Причина здесь очевидна – таблично управляемый метод разбора. Весь

«интеллект» синтаксического анализатора заложен в управляющей таблице разбора – матрице отношений операторного предшествования.

Работа основных методов класса TParser, таких, как CreatePRN (создать обратную польскую запись) и CreateAST (создать абстрактное синтаксическое дерево) сводится к следующему. Организуется цикл просмотра входного потока токенов. На каждом шаге этого цикла исследуется верхний терминал в стеке и текущий входной символ. Происходит обращение к управляющей таблице разбора, которая «говорит», какое действие необходимо предпринять, а именно: **Перенос, Свертка, Допуск, Ошибка**. В случае свертки выполняются и семантические действия, связанные с построением постфиксной польской записи и очередного узла абстрактного синтаксического дерева. Надо сказать, что абстрактное синтаксическое дерево можно реконструировать по постфиксной польской записи, и наоборот постфиксную польскую запись можно получить путем соответствующего обхода абстрактного синтаксического дерева. Здесь возникает одна небольшая проблема для исследования – оценить, какая из внутренних форм представления выражения является более эффективной с точки зрения интерпретации.

Например, для выражения `a+b*(c+d*(e+f))` постфиксная польская запись будет иметь вид `abcdef+*+*+`, а ее эквивалент в виде XML-документа представится так:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<TokenList>
  <Token Type="Identifier" Value="a" Line="1" StartPos="1" EndPos="1" />"
  <Token Type="Identifier" Value="b" Line="1" StartPos="3" EndPos="3" />"
  <Token Type="Identifier" Value="c" Line="1" StartPos="6" EndPos="6" />"
  <Token Type="Identifier" Value="d" Line="1" StartPos="8" EndPos="8" />"
  <Token Type="Identifier" Value="e" Line="1" StartPos="11" EndPos="11" />"
  <Token Type="Identifier" Value="f" Line="1" StartPos="13" EndPos="13" />"
  <Token Type="Operator" Value="+" Line="1" StartPos="12" EndPos="12" />"
  <Token Type="Operator" Value="*" Line="1" StartPos="9" EndPos="9" />"
  <Token Type="Operator" Value="+" Line="1" StartPos="7" EndPos="7" />"
  <Token Type="Operator" Value="*" Line="1" StartPos="4" EndPos="4" />"
  <Token Type="Operator" Value="+" Line="1" StartPos="2" EndPos="2" />"
</TokenList>
```

Абстрактное синтаксическое дерево для того же выражения в виде XML-документа представится так:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ASTTree>
  <Node>
    <Operator name="AddOperator" spell="+"/>
    <LeftOperand>
   <Node>
     <Operator name="AddOperator" spell="*"/>
        <LeftOperand>
       <Node>
```

```
          <Operator name="AddOperator" spell="+"/>
          <LeftOperand>
            <Node>
                 <Operator name="AddOperator" spell="*"/>
                 <LeftOperand>
                   <Node>
                  <Operator name="MulOperator" spell="+" />
                  <LeftOperand>
                    <Leaf type="Identifier" name="a" />
                  </LeftOperand>
                  <RightOperand>
                    <leaf type="Identifier" name="b" />
                  </RightOperand>
                </Node>
                 </LeftOperand>
              <RightOperand>
                <Leaf type="Identifier" name="c" />
                 </RightOperand>
            </Node>
          </LeftOperand>
          <RightOperand>
            <Leaf type="Identifier" name="d" />
          </RightOperand>
        </Node>
      </LeftOperand>
      <RightOperand>
        <Leaf type="Identifier" name="e" />
      </RightOperand>
    </Node>
     </LeftOperand>
     <RightOperand>
       <Leaf type="Identifier" name="f" />
     </RightOperand>
  </Node>
</ASTTree>
```

Интерпретаторы (вычислители выражения) являются методами класса TParser. Метод CalcAST вычисляет значение выражения, осуществляя рекурсивный спуск по абстрактному синтаксическому дереву. Метод CalcPRN вычисляет выражение, просматривая список токенов постфиксной польской записи слева направо и используя стек. Оба метода осуществляют контроль типов и, при необходимости, выполняют преобразования типов.

3.5. Обработка и нейтрализация ошибок при синтаксическом анализе

Имеется два условия, когда процесс синтаксического анализа может обнаружить синтаксическую ошибку.

1. Если между терминалом на вершине стека и текущим входным символом не определенно ни одно из отношений предшествования.

2. Если найдена основа, но не имеется правила вывода с найденной основой в правой части.

Программа обработки и нейтрализации ошибок разделена на отдельные части. Одна часть обрабатывает ошибки типа (2), которые могут возникнуть только в процессе свертки. Эта программа выталкивает символы из стека, однако при отсутствии правила вывода для свертки никакие семантические действия не выполняются; вместо этого выводится сообщение об ошибке. Содержание сообщения об ошибке зависит от того, на какое правило вывода «похожи» извлекаемые из стека символы.

Для обработки ошибок типа (1) делается следующее. Для каждой пустой клетки матрицы определяется подпрограмма восстановления после ошибок; одна и та же подпрограмма может использоваться в нескольких клетках. Например, для грамматики

E -> E+E|E-E|E*E|E/E|E^E|(E)|id определены 4 подпрограммы, сущность которых состоит в следующем.

errorl: Вызывается, если выражение пустое.

Действие: Append(Input, id). // Вставить id во входную ленту.

Сообщение: Пропущен операнд.

error2: Вызывается, если выражение начинается с «)».

Действие: Inc(p). // Пропустить «)» на входной ленте.

Сообщение: Несбалансированная правая скобка

error3: Вызывается, когда id или «)» следует за id или «(«

Действие: Append(Input, “+”). // Вставить «+» во входную ленту.

Сообщение: Пропущен оператор

error4: Вызывается при завершении выражения «(«.

Действие: Pop(S). // Вытолкнуть «(« из стека.

Сообщение: Пропущена правая скобка.

3.6. Экспериментальная часть

Проводилась проверка работоспособности программы для различных видов грамматик, для выражений различной степени сложности и для различных видов использования выражений, а именно: для вычисления значения функции в точке, для

уточнения корня уравнения, для приближенного вычисления интеграла, для табулирования функции и для построения графика функции с использованием компонента TChart.

Для проверки быстродействия генерировались три типа входных рекурсивных образцов с увеличивающейся глубиной рекурсии

леворекурсивный: ((((. . . (1+1)+ . . . +1)+1)+1)+1),

праворекурсивный: (1+(1+(1+(1+ . . . +(1+1) . . .))))

"центральный" вида: (1+(1+(1+(1+ . . . +(1+1)+ . . . +1)+1)+1)+1).

Леворекурсивные входные цепочки создают наименьшую нагрузку на стек анализатора, вызывая попеременные сдвиги и свертки, праворекурсивные создают наибольшую нагрузку на стек анализатора, поскольку сначала происходит многократный сдвиг, центральные образцы сочетают "правую" ситуацию с левой. Кроме того, тестированию подлежит и случай с отключенными семантическими правилами для оценки создаваемой ими нагрузки.

4. Выводы

1. Цель работы достигнута – разработана программа, которая по входной грамматике строит управляющую таблицу разбора синтаксического анализатора выражений и, используя легко реализуемый метод грамматического разбора, применимый для широкого класса LR(1)-грамматик, обеспечивает эффективное вычисление выражений.
2. Выбранный формат входной грамматики в виде XML-документа является адекватным формам Бэкуса-Наура, но обеспечивает большую гибкость в работе при использовании объектов DOM-модели и надежность в сочетании с XSD-схемой.
3. Методы и алгоритмы, реализованные в лексическом и синтаксическом анализаторах сопоставимы с методами, используемыми в современных генераторах компиляторов.
4. Визуализация всех этапов работы программы в виде древовидных структур данных и XML-документов позволяет использовать ее не только ради вычисления выражений, но и для изучения внутренних механизмов основных фаз компилятора переднего плана.
5. Работа программы продемонстрирована на ряде примеров вычисления функции для широкого спектра грамматик.

Список использованных источников

1. Белоусов А.И., Ткачев С.Б. Дискретная математика: Учеб. Для вузов / Под ред. В.С. Зарубина,А.П. Крищенко. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001**.**
2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т.1.: Синтаксический анализ. - М.: Мир, 1978.
3. Альфред Ахо, Джеффри Ульман, Рави Сети. Компиляторы: принципы, технологии и инструменты. - М.: Вильямс, 2001.
4. Вайнгартен Ф. Трансляция языков программирования. - М.: Мир, 1977.
5. Баррон Д. Рекурсивные методы в программировании. - М.: Мир, 1974.
6. Кэнту М. Delphi 6 для профессионалов. - Спб.: Питер, 2002.
7. Архангельский А.Я. Программирование в Delphi для Windows. Версии 2006, 2007, Turbo Delphi. - М.: ООО Бином-Пресс, 2007.