УДК 004.4'41

РАЗРАБОТКА ГЕНЕРАТОРА ТРАНСЛЯТОРОВ CAIO

А.К. Демидов

В работе определяются свойства средств разработки трансляторов, важные для целей обучения, и описывается разработка генератора трансляторов Caio, соответствующего выявленным свойствам.

Ключевые слова: трансляторы, лексический анализ, синтаксический анализ, абстрактное синтаксическое дерево.

Средства разработки трансляторов используются не только для создания компиляторов и интерпретаторов, но и для обработки данных, имеющих сложную структуру и/или удобную для ввода и понимания человеком форму. Поэтому в подготовку разработчиков программного обеспечения входит изучение средств для разработки трансляторов. При этом в [1] отмечается, что студенты смогут получить необходимые навыки только при самостоятельной разработке транслятора, что является объемной и сложной задачей, но использование декларативных лексических анализаторов и генераторов синтаксических анализаторов может уменьшить эту сложность. К сожалению, разработчики таких средств мало внимания уделяют декларативности определений, их совместимости со стандартизированными формами описания синтаксиса. Поэтому было проведено исследование по классификации свойств существующих средств разработки трансляторов, определены те возможности, которые будут полезны с точки зрения целей обучения, и предложен генератор трансляторов, упрощающий процесс разработки.

Инструменты для разработки трансляторов можно разделить на три группы:

1. Генераторы кода (YACC, ANTLR, COCO/R, JavaCC), которые создают наиболее эффективный код и позволяют указать действия по созданию абстрактных синтаксических деревьев (AST) в самом правиле.

2. Библиотеки классов и шаблонов (Boost Spirit, Irony), в которых правила грамматики задаются с помощью перегрузки операций или композиции вызовов, в результате чего правила грамматики имеют необычный вид, а также перед началом разбора затрачивается время на построение вспомогательных таблиц.

3. Генераторы времени исполнения (parglare), в которых грамматика задается в форме строки, что дает возможность загружать синтаксис языка из файла, но в результате сложно сделать привязку действий к правилам и также требуется время на построение вспомогательных таблиц перед началом разбора.

Также можно средства разработки можно разделить по типу грамматики и распознавателю (LL(1), LL(k), LR(1) или GLR), но эта характеристика практически не влияет на сложность определения синтаксиса.

Следующей важной характеристикой является форма правил грамматики. При использовании расширенной формы Бэкуса-Наура (РБНФ) определение синтаксиса существенно упрощается. Большинство разработчиков использует вариант РБНФ, не соответствующий стандарту [2], с нотацией заимствованной из регулярных выражений (например, (a)* вместо {a}). В результате читабельность определений снижается, так как необходимо при появлении открывающей скобки необходимо найти соответствующую ей закрывающую и символ после неё, чтобы определить какой вид последовательности начался – повторяющейся, необязательной или группирующей. В Spirit разработчики для формирования правил перегрузили префиксные операции *, + и −, и такое вынужденное изменение помогло разработчикам избежать указанного недостатка. Из популярных инструментов стандарту следует только COCO/R.

Добавление в правило действий для создания AST или вычислений также снижает читабельность, но эту проблема обычно решается разделением текста определений на два столбца – собственно правила и действия или тип создаваемого узла AST. Большей проблемой является необходимость объявления типа атрибутов для генераторов в языках со строгой типизацией. В YACC и Spirit такие объявления выполняются отдельно от правил грамматики, но в JavaCC и COCO/R атрибуты указываются непосредственно у каждого символа, что существенно снижает декларативность определений правил грамматики. В тех инструментах, где AST создается автоматически по определению правил грамматики (ANTLR, Irony), для правильного формирования дерева возникает необходимость указывать имена для некоторых последовательностей внутри правила, что также снижает декларативность.

Также важной характеристикой инструмента является возможность задать приоритет и ассоциативность операторов. Если такая возможность доступна, то можно обойтись меньшим количеством вспомогательных символов и правил.

Перед выполнением синтаксического анализа необходимо выделить лексемы. Определить лексемы можно как с помощью регулярных выражений, так и с помощью правил грамматики. Во втором случае можно использовать единую нотацию для описания как синтаксиса, так и лексики языка, но с другой стороны, такое описание становится слишком сложным, что видно на примере определения РБНФ в [2]. Поэтому разработчики инструментов для описания лексем используют либо регулярные выражения, либо комбинацию из РБНФ и нотации из регулярных выражений. Лексический анализатор должен иметь возможность выделять лексемы в зависимости от текущего контекста (как в ANTLR) или определять распознающие подавтоматы (как в LEX).

В конечном итоге результат синтаксического анализа должен быть представлен в виде AST, и наличие в инструменте встроенных средств для этого существенно упрощает разработку трансляторов. Многие инструменты ограничиваются реализацией шаблона проектирования слушатель или выполнением действий при свёртке, что ограничивает их область применения загрузкой данных сложной структуры. С другой стороны, инструменты, в которых происходит такое дерево строится автоматически (ANTLR, Irony), требуют специальных указаний в правилах грамматики. Поэтому более удобной является возможность явным образом задать структуру узлов AST и указывать их атрибуты при создании, как это сделано в инструменте Memphis [3].

Для целей обучения генератор трансляторов Caio (сокращение от Compilers: All in One), с одной стороны, должен для повышения декларативности определений синтаксиса и лексики максимально использовать стандарты, а с другой стороны, должен быть совместим с каким-либо распространенным инструментом разработки, чтобы полученные в результате обучения умения и навыки можно было легко применить на практике. Наименьшие изменения для поддержки РБНФ и AST необходимо внести в формат определений YACC, у которого есть широко используемые улучшенные версии для разных языков программирования (Bison/Lemon/ GPPG/fsyacc). С учетом вышеназванных целей, лучше не писать прямой перевод определений на целевой язык, а выполнить генерацию исходного кода для существующего инструмента. Для лексического анализа лучше использовать RE/flex [4], который достаточно полно реализует стандарт ECMAscript для регулярных выражений, поддерживаемый во многих языках программирования. Так как работа лексического анализатора зависит от структур данных и опций, задаваемых в синтаксическом анализаторе, для упрощения определений код для RE/flex также лучше генерировать, используя единый файл с определениями.

Структура описаний для Caio выглядит следующим образом:

объявления
%%
правила для лексического анализатора
%%
правила для синтаксического анализатора
%%
код подпрограмм

В первом разделе можно написать следующие виды объявлений:

1. %option список – указание опций, которые будут применяться при генерации кода или передаваться соответствующим инструментам. Среди опций можно указать тип грамматики (bnf или ebnf), нечувствительность к регистру букв (case-insensitive), включить определение позиции лексемы (yylineno, locations), запретить автоматическую генерацию правил для ли-

тералов (noliteral-rules) или вспомогательных подпрограмм (nomain, yywrap, yyerror), ожидаемое количество конфликтов в грамматике (expect(n)) и т.д.

2. %operator <вид> список – указание приоритета, позиции и ассоциативности операторов. Здесь вид может принимать одно из следующих значений xfx, xfy, yfx, fx, fy, xf, yf, где f – положение оператора, x – выражение с приоритетом оператора строго выше определяемого, y – выше или равного по приоритету. Этот способ описания применяется в языке Prolog и имеет большую наглядность и гибкость, чем объявления YACC.

3. %type <тип> список – объявление типа символов грамматики и/или узлов дерева. Так как разные виды символов грамматики в РБНФ отличаются визуально, можно не делать отдельные объявления для токенов языка, как это было реализовано в YACC. Также можно не определять структуру для хранения атрибутов символов грамматики, а создать её автоматически по объявлениям, как это сделано в последних версиях Bison. Также по данным объявлениям и действиям, указанных в правилах, автоматически создается иерархия классов для хранения узлов AST и функции для их создания. Имена символов грамматики могут содержать русские буквы в целях повышения наглядности.

4. %{ } и %code назначение {} – код, который помещается в заголовочный файл или в генерируемый код для инструмента, указанного в назначении.

Правила для лексического и синтаксического анализатора записываются в виде двух столбцов, в левом столбце записывается регулярное выражение или правило грамматики, а в правом – действие. Для большей совместимости с YACC пробелы в именах нетерминальных символов запрещены, в правилах грамматики можно указывать разделитель двоеточие вместо = и не писать запятых. Для наиболее частого варианта действия – создания узла AST – используется конструкция <имя_узла(аргументы)>. В лексическом анализаторе перед этой конструкцией можно написать также распознанный терминальный символ. Если символ не указан, то выполняется поиск распознанной лексемы среди литералов с помощью вызова yyliteral(yytext). В остальных случаях можно написать произвольный код в { } или %{ } (при использовании РБНФ), который может содержать обращение к атрибутам результирующего символа в форме $$ и к атрибутам символов правил грамматики в форме $n, как в YACC. В Caio данная возможность применяется и в действиях лексического анализатора вместо неудобного обращения к полям yylval. Также в действиях для лексического анализатора можно написать конструкцию «return терминальный_символ;», что позволяет полностью избавиться от альтернативных имен для терминальных символов, которые необходимо определять и использовать для взаимодействия LEX и YACC.

Код подпрограмм, содержащийся в последнем разделе, может включать две специальные конструкции для обработки узлов AST:

match указатель_на_узел {
rule имя_узла(имена_аргументов): код
...
}
visitor имя<тип_узла,тип_результата> {
visit имя_узла(имена_аргументов): код
...
}

Первая конструкция реализуется через динамическую проверку типа узла (dynamic_cast), вторая конструкция используется для создания шаблонного класса с виртуальными методами для обработки каждого вида узлов AST и перегруженной операцией вызова функции, а также экземпляра этого класса с указанным именем. Вторая конструкция может иметь большую эффективность при обработке деревьев с большим количеством видов узлов.

Пример использования Caio для создания интерпретатора языка Basic:

```
%option yylineno case-insensitive
%operator <yfx> '+' '-'
%operator <yfx> '*' '/'
%operator <fy> '-'
%type <int> ?число? ?идент?
%type <Statement> оператор
%type <Expr> выр
%{
#include <string>
#include <iostream>
#include <map>
#include <cctype>
using namespace std;
void interp(List<Statement> p);
int find_id(string name);
}
%% // правила для лексического анализатора
[ \t\r]                ; // пропустить пробелы
\d+                    ?число?<stoi(yytext)>
[a-zA-Z]\w*            ?идент?<find_id(yytext)>
.|\n                        // вернуть как литерал
%% // правила для синтаксического анализатора
программа = код                    %{ interp($1); }
          ;
код = { оператор '\n'                <$1>
```

```
      }                                 <$1>
      ;
оператор =
  | ?идент? '=' выр                     <assign($1,$3)>
  | "input" ?идент?                     <input($2)>
  | "print" выр                         <print($2)>
  | "while" выр '\n' код "wend"         <whilestmt($2,$4)>

  | "if" выр ["then"] '\n'  код
     [ "else" '\n' код                  <$3>
     ] "end" "if"                       <ifstmt($2,$5,$6)>

  ;
выр = выр '+' выр                       <plus($1,$3)>
    | выр '-' выр                       <minus($1,$3)>
    | выр '*' выр                       <mult($1,$3)>
    | выр '/' выр                       <divide($1,$3)>
    | '-' выр                           <neg($2)>
    | '(' выр ')'                       <$2>
    | ?число?                           <number($1)>
    | ?идент?                           <ident($1)>
    ;
%%
int mem[1000]; // память для данных
visitor eval<Expr,int> { // вычисление выражения
  visit plus(x, y): return eval(x)+eval(y);
  visit minus(x, y): return eval(x)-eval(y);
  visit mult(x, y): return eval(x)*eval(y);
  visit divide(x, y): return eval(x)/eval(y);
  visit neg(x):  return -eval(x);
  visit number(x):  return x;
  visit ident(x): return mem[x];
}
void interp(List<Statement> p) // выполнение операторов
{ for(auto s:p)
    match s {
      rule assign(v, e): mem[v]=eval(e);
      rule input(v): cin>>mem[v];
      rule print(e): cout<<eval(e)<<endl;
      rule whilestmt(e,p1): while(eval(e)) interp(p1);
      rule ifstmt(e,p1,p2): if(eval(e)) interp(p1);
                            else interp(p2);
    }
}
```

```
map<string,int> ti; // таблица идентификаторов
int find_id(string name) // поиск в таблице
{ for(auto &ch:name) ch=toupper(ch);
  int k=ti[name];
  if(k==0) k=ti[name]=ti.size();
  return k;
}
```

Код примера содержит 74 строки в одном файле, из них 29 строк приходится на 4 правила грамматики и вспомогательные объявления. При прямом использовании инструментов Bison, RE/flex и Memphis необходимо написать 139 строк в 5 файлах, из них 55 строк – объявления и 6 правил, а 34 строки обеспечивают стандартное взаимодействие инструментов, запуск программы и будут повторяться в разных проектах. Таким образом, сложность разработки учебных проектов уменьшена почти в два раза.

Разработанный генератор трансляторов Caio интегрирован в среду разработки MinIDE для C/C++, которую можно скачать на сайте [5].

Библиографический список

1. Рекомендации по преподаванию программной инженерии и информатики в университетах = Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering; Computing Curricula 2001: Computer Science. – М.: ИНТУИТ.РУ «Интернет-Университет Информационных Технологий», 2007. – 462 с.

2. ISO/IEC 14977:1996, Information technology – Syntactic metalanguage – Extended BNF. – 12 с.

3. The MEMPHIS Tree Builder & Tree Walker Tool [Электронный ресурс]. – URL: http://memphis.compilertools.net/.

4. RE/flex guide [Электронный ресурс]. – URL: https://www.genivia.com/doc/reflex/html/index.html.

5. Учебные материалы [Электронный ресурс]. – URL: https://ipc.susu.ru/learn.html.

К содержанию