编译实验二实验报告——Parser of PCAT

田应涛 10302010029　王曦　10300240014
陈济凡 10300240076　承沐南 10300240003

May 5, 2013

Contents

1 YACC

1.1 YACC 概述

yacc（Yet Another Compiler Compiler），是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器）。yacc 生成的编译器主要是用 C 语言写成的语法解析器（Parser），需要与词法解析器 Lex 一起使用，再把两部份产生出來的 C 程序一并编译。yacc 本来只在 Unix 系统上才有，但现在已普遍移植往 Windows 及其他平台。

yacc 的输入是巴科斯范式（BNF）表达的语法规则以及语法规约的处理代码，Yacc 输出的是基于表驱动的编译器，包含输入的语法规约的处理代码部分。

yacc 是开发编译器的一个有用的工具，采用 LALR(1) 语法分析方法。yacc 最初由 AT&T 的 Steven C. Johnson 为 Unix 操作系统开发，后来一些兼容的程序如 Berkeley Yacc，GNU bison，MKS yacc 和 Abraxas yacc 陆续出现。它们都在原先基础上做了少许改进或者增加，但是基本概念是相同的。

由于所产生的解析器需要词法分析器配合，因此 Yacc 经常和词法分析器的产生器——一般就是 Lex——联合使用。IEEE POSIX P1003.2 标准定义了 Lex 和 Yacc 的功能和需求。

本实验中使用的 GNU bison 是一个自由软件，用于自动生成语法分析器程序，实际上可用于所有常见的操作系统。Bison 把 LALR 形式的上下文无关文法描述转换为可做语法分析的 C 或 C++ 程序。在新近版本中，Bison 增加了对 GLR 语法分析算法的支持。GNU bison 基本兼容 Yacc，并做了一些改进。它一般与 flex 一起使用。

1.2 YACC 文件结构

YACC 文件结构分为三个部分, 分别为 < 定义部分 >< 规则部分 >< 用户子程序部分 >。其中规则部分是必须的, 定义和用户子程序部分是任选的。

- 定义部分
  定义部分的 C 语言部分起始于 %{ 符号, 终止于%} 符号，其间可以是包括 include 语句、声明语句在内的 C 语句。除此之外，还可以包含语义值数据类型定义、单词定义、非终结符定义和优先级定义。

  其中，C 语言代码部分同 LEX，语义值数据类型定义部分定义了进行语法分析时语义栈中元素的数据类型，可用宏 YYSTYPE 定义。结合次序和优先级别的定义，将让 YACC 在编译源文件时，面对由文法的二义性引起的移进 -归约冲突时，能正确的选择移进或归约。

- 规则部分
  规则部分起始于"%%" 符号, 终止于"%%" 符号, 其间则是语法规则。语法规则由模式和动作两部分组成。模式部分由非终结符组成，动作部分是由 C 语言语句组成，这些语句用来对所匹配的模式进行相应处理。动作部分如果有多行执行语句, 也可以用 {} 括起来。YACC 最欢迎的是左递归文法，而用右递归文法则可能会导致分析栈溢出，故要尽量避免用右递归文法。

- 用户子程序部分
  最后一个%% 后面的内容是用户子程序部分，可以包含用 C 语言编写的子程序，而这些子程序可以用在前面的动作中，这样就可以达到简化编程的目的。

1.3 终结符与非终结符

在 YACC 源文件中，有两种方式表示的单词（终结符）：一种是在定义部分通过 YACC 指令%token 定义文法中出现的单词，称为有名单词，如%token NUMBER 就定义了一有名单词 NUMBER，可用它来表示整数的种别码；另一种是单个字符，称为字符单词，如单个字符’+’、’-’,’a’,’4’,’!’等，它们本身作为终结符出现在规则部分，不需要在定义部分说明，可直接加单引使用，如同 C 语言的字符常量。

非终结符一般为不加引号的标识符，习惯上用小写字母组成的字符串表示。如可用 exp 表示用来表示表达式的非终结符。

YACC 在对源文件进行编译时，将对所有的单词和非终结符进行编码，并用该编码建立分析表和语法分析器。单词的编码原则是：字符单词使用其对应的 ASCII 码，有名单词则由分析器进行编码。用户在对有名单词进行命名时，一定要注意不要和使用该单词名的 C 源程序中已有的宏名相同，否则在编译该 C 模块时是会产生宏定义冲突的。

2 实验实现

我们实现的语法规则参照了 http://web.cecs.pdx.edu/ apt/cs302_1999/pcat99/pcat99.html 中给出的 PCAT 标准，并根据该标准写出了 EBNF 语法，之后我们将该 EBNF 语法转换至 BNF 语法并制定了抽象语法树生成规则。

2.1 EBNF 语法

```
program          -> PROGRAM IS body ';'
body             -> {declaration} BEGIN {statement} END
declaration      -> VAR {var-decl}
                 -> TYPE {type-decl}
                 -> PROCEDURE {procedure-decl}
var-decl         -> ID { ',' ID } [ ':' typename ] ':=' expression ';'
type-decl        -> ID IS type ';'
procedure-decl   -> ID formal-params [':' typename] IS body ';'
typename         -> ID
type             -> ARRAY OF typename
                 -> RECORD component {component} END
component        -> ID ':' typename ';'
formal-params    -> '(' fp-section {';' fp-section } ')'
                 -> '(' ')'
fp-section       -> ID {',' ID} ':' typename
statement        -> lvalue ':=' expression ';'
                 -> ID actual-params ';'
                 -> READ '(' lvalue {',' lvalue} ')' ';'
                 -> WRITE write-params ';'
                 -> IF expression THEN {statement}
```

```
                   {ELSIF expression THEN {statement}}
                   [ELSE {statement}] END ';'
                -> WHILE expression DO {statement} END ';'
                -> LOOP {statement} END ';'
                -> FOR ID ':=' expression TO expression [ BY expression
                     ]
                   DO {statement} END ';'
                -> EXIT ';'
                -> RETURN [expression] ';'
write-params    -> '(' write-expr {',' write-expr } ')'
                -> '(' ')'
write-expr      -> STRING
                -> expression
expression      -> number
                -> lvalue
                -> '(' expression ')'
                -> unary-op expression
                -> expression binary-op expression
                -> ID actual-params
                -> ID record-inits
                -> ID array-inits
lvalue          -> ID
                -> lvalue '[' expression ']'
                -> lvalue '.' ID
actual-params   -> '(' expression {',' expression} ')'
                -> '(' ')'
record-inits    -> '{' ID ':=' expression { ';' ID ':=' expression} '}'
array-inits     -> '[<' array-init { ',' array-init } '>]'
array-init      -> [ expression OF ] expression
number          -> INTEGER | REAL
unary-op        -> '+' | '-' | NOT
binary-op       -> '+' | '-' |  '*' | '/' | DIV | MOD | OR | AND
                -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

2.2 BNF 语法

```
program                     -> PROGRAM IS body ';'
body                        -> declaration-S BEGIN statement-S END
declaration-S               -> declaration-S declaration
                            ->
statement-S                 -> statement-S statement
                            ->
declaration                 -> VAR var-decl-S
                            -> TYPE type-decl-S
                            -> PROCEDURE procedure-decl-S
var-decl-S                  -> var-decl-S var-decl
                            ->
type-decl-S                 -> type-decl-S type-decl
                            ->
porcedure-decl-S            -> porcedure-decl-S porcedure-decl
var-decl                    -> ID var-decl-id-S var-decl-type-O ':='
    expression ';'
var-decl-id-S               -> var-decl-id-S ',' ID
                            ->
var-decl-type-O             -> ':' typename
                            ->
type-decl                   -> ID IS type ';'
```

```
procedure-decl              -> ID formal-params procedure-decl-type-O
    IS body ';'
procedure-decl-type-O       -> ':' typename
                            ->
typename                    -> ID
type                        -> ARRAY OF typename
                            -> RECORD component component-S END
component                   -> ID ':' typename ';'
formal-params               -> '(' fp-section fp-section-S ')'
                            -> '(' ')'
fp-section-S                -> fp-section-S ';' fp-section
                            ->
fp-section                  -> ID fp-section-id-S ':' typename
fp-section-id-S             -> fp-section-id-S ',' ID
                            ->
statement                   -> lvalue ':=' expression ';'
                            -> ID actual-params ';'
                            -> READ '(' lvalue statement-lvalue-S ')'
                                ';'
                            -> WRITE write-params ';'
                            -> IF expression THEN statement-S
                               statement-elsif-S
                               statement-else-O END ';'
                            -> WHILE expression DO statement-S END ';'
                            -> LOOP statement-S END ';'
                            -> FOR ID ':=' expression TO expression
                                statement-by-O
                               DO statement-S END ';'
                            -> EXIT ';'
                            -> RETURN expression-O ';'
statement-lvalue-S          -> statement-lvalue-S ',' lvalue
                            ->
statement-elsif-S           -> statement-elsif-S ELSIF expression THEN
     statement-S
                            ->
statement-else-O            -> ELSE statement-S
                            ->
statement-by-O              -> statement-by-O BY expression
                            ->
write-params                -> '(' write-expr write-params-expr-S ')'
                            -> '(' ')'
write-params-expr-S         -> write-params-expr-S ',' write-expr
                            ->
write-expr                  -> STRING
                            -> expression
expression-O                -> expression
                            ->
expression                  -> number
                            -> lvalue
                            -> '(' expression ')'
                            -> unary-op expression
                            -> expression binary-op expression
                            -> ID actual-params
                            -> ID record-inits
                            -> ID array-inits
lvalue                      -> ID
                            -> lvalue '[' expression ']'
```

```
                              -> lvalue '.' ID
actual-params                 -> '(' expression actual-params-expr-S ')'
                              -> '(' ')'
actual-params-expr-S          -> actual-params-expr-S ',' expression
                              ->
record-inits                  -> '{' ID ':=' expression record-inits-
    pair-S '}'
record-inits-pair-S           -> record-inits-pair-S ';' ID ':='
    expression
                              ->
array-inits                   -> '[<' array-init array-inits-array-init-
    S '>]'
array-inits-array-init-S      -> array-inits-array-init-S ',' array-init
                              ->
array-init                    -> array-init-expr-of-S expression
array-init-expr-of-S          -> array-init-expr-of-S expression OF
                              ->
number                        -> INTEGER | REAL
unary-op                      -> '+' | '-' | NOT
binary-op                     -> '+' | '-' |  '*' | '/' | DIV | MOD | OR
     | AND
                              -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

2.3 抽象语法树生成规则

```
==========================
Abstruct Syntax - Modified
==========================

program                  -> body
body                     -> ( BodyDef ( declarations-list statements-
    list ) )
declarations-list        -> ( DeclareList ( {declarations} ) )
declarations             -> ( VarDecs ( {var-dec} ) )
                         -> ( TypeDecs ( {type-dec} ) )
                         -> ( ProcDecs ( {proc-dec} ) )
var-dec                  -> ( VarDec ( ID type expression ) )
type-dec                 -> ( TypeDec ( ID type ) )
proc-dec                 -> ( ProcDec ( ID formal-param-list type body
    ) )
type                     -> ( NamedTyp ( ID ) )
                         -> ( ArrayTyp ( type ) )
                         -> ( RecordTyp ( component-list ) )
                         -> ( NoTyp () )
component-list           -> ( CompList ( { component } ) )
component                -> ( Comp ( ID type ) )
formal-param-list        -> ( FormalParamList ( {formal-param } ) )
formal-param             -> ( Param ( ID type ) )
statements-list          -> ( StList ( { statements } ) )
statement                -> ( AssignSt ( lvalue expression ) )
                         -> ( CallSt ( ID expression-list ) )
                         -> ( ReadSt ( lvalue-list ) )
                         -> ( WriteSt  ( expression-list ) )
                         -> ( IfSt ( expression statement statement ) )
                         -> ( WhileSt ( expression statement ) )
                         -> ( LoopSt ( statement ) )
```

```
                        -> ( ForSt ( ID expression expression
                            expression statement ) )
                        -> ( ExitSt () )
                        -> ( RetSt ( expression ) )
                        -> ( RetSt () )
                        -> ( SeqSt ( statements-list ) )
expression-list         -> ( ExprList ( { expression } ) )
expression              -> ( BinOpExp ( binop expression expression )
    )
                        -> ( UnOpExp  ( unop expression ) )
                        -> ( LvalExp ( lvalue ) )
                        -> ( CallExp ( ID expression-list ) )
                        -> ( RecordExp ( ID record-init-list ) )
                        -> ( ArrayExp  ( ID array-init-list ) )
                        -> ( IntConst ( INTEGER ) )
                        -> ( RealConst ( REAL ) )
                        -> ( StringConst ( STRING ) )
record-init-list        -> ( RecordInitList ( { record-init } ) )
record-init             -> ( RecordInit ( ID expression ) )
array-init-list         -> ( ArrayInitList ( { array-init-list } ) )
array-init              -> ( ArrayInit ( expression expression ) )
lvalue-list             -> ( LvalList ( { lvalue } ) )
lvalue                  -> ( Var ( ID ) )
                        -> ( ArrayDeref ( lvalue expression ) )
                        -> ( RecordDeref ( lvalue ID ) )
binop                   -> GT | LT | EQ | GE | LE | NE | PLUS | MINUS
    | TIMES | SLASH
                        -> DIV | MOD | AND | OR
unop                    -> UPLUS | UMINUS | NOT

note:
Remove 'line'
using REAL for real const instead of STRING
format: ( tag ( arg1 arg2....) )

for ops (binop and unop), formats are:
binop -> GT  = ( GT () )
unop -> NOT  = ( NOT () )
they are single nodes without children


IntConst,RealConst,StringConst is not need because AST provides int_ast
    ,real_ast,str_ast


====
tags
====
BodyDef,decle
```

3 运行结果