Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Compiladores
Trabalho Prático 2
Analisador Sintático

Pedro Araujo Pires
Pedro Henrique Doffemond Costa

Introdução

Em ciência da computação e linguística, análise sintática (também conhecido pelo termo em inglês parsing) é o processo de analisar uma sequência de entrada (lida de um arquivo de computador ou do teclado, por exemplo) para determinar sua estrutura gramatical segundo uma determinada gramática formal. Essa análise faz parte de um compilador, junto com a análise léxica e análise semântica. A análise sintática transforma um texto na entrada em uma estrutura de dados, em geral uma árvore, o que é conveniente para processamento posterior e captura a hierarquia implícita desta entrada. Através da análise léxica é obtido um grupo de tokens, para que o analisador sintático use um conjunto de regras para construir uma árvore sintática da estrutura.

Objetivo

Implementar um analisador sintático para a gramática da linguagem L2012-1, que opere em conjunto com o analisador léxico desenvolvido no trabalho anterior. O analisador sintático deve receber os tokens retornados pelo analisador léxico, e decidir se eles estão ou não de acordo com a gramática.

Ferramentas

As ferramentas utilizadas neste trabalho foram o Lex para gerar o analisador léxico, e o YACC para gerar o analisador sintático.

Análise Sintática

O YACC gera um analisador sintático LALR a partir de um arquivo de entrada, que contém a gramática a ser utilizada. O arquivo é dividido em três partes. A primeira parte provê espaço para declarações de como o parser será gerado. A segunda parte contem as regras da gramática, e a terceira parte contem programas, como por exemplo o analisador léxico.

Para que o YACC funcione em conjunto com o Lex, o analisador léxico precisou sofrer algumas modificações. Para o primeiro trabalho, o analisador léxico lia toda a entrada, e imprimia na tela os tokens encontrados. Para que ele trabalhe em conjunto com o YACC, foi necessário, a cada token encontrado, retornar um valor referente ao token. Esse valores são definidos automaticamente pelo YACC, no momento da declaração dos tokens.

A gramática da linguagem tambem precisou sofrer modificações, para que as funções *built-in* da linguagem fossem incorporadas à gramática. Mais especificamente, a regra

```
function ref par ::= variable " (" expr list " )"
```

foi alterada para:

```
function ref par ::= built_in_function "(" expr ")"
                   | variable " (" expr list " )"
```

Com isso foi criado um novo símbolo não terminal (built_in_function), e sua definição é:

```
built_in_function ::= "sin" | "log" | "cos"
                    | "ord" | "abs" | "sqrt"
                    | "exp" | "eof" | "eoln"
```

Essa modificação foi necessária pois essas funções recebem somente um parâmetro, e a definição da function_ref_par espera uma lista de parâmetros.

Outra modificação necessária foi a introdução do símbolo vazio na regra

```
decl list ::= decl list " ;" decl
            | decl
```

pois sem o vazio, todos os programas na linguagem L2012-1 obrigatoriamente precisariam ter pelo menos uma declaração de variável, ou declaração de procedimento. Como a L2012-1 é um subconjunto das linguagens imperativas mais comuns, e essas linguagens não obrigam o programador a declarar alguma coisa, foi feita a modificação.

Ao executar o YACC, foi avisado que a gramática gerou um conflito shift/reduce no analisador sintático. Mais especificamente, esse conflito foi gerado pela regra:

```
if stmt ::= if cond then stmt
          | if cond then stmt else stmt
```

Quando esse tipo de erro ocorre, o YACC cria o analisador sintático com uma ação *default*: se é possível fazer o shift, ele é feito. Na maioria das linguagens de programação, quando temos `If`'s e `else`'s aninhados, cada `else` é casado com o `if` mais próximo, e isso significa exatamente executar o shift ao invés do reduce, quando há o conflito. Um dos programas utilizados para testar o analisador sintático (`test.lp`) evidencia essa operação.

Modo de Utilização

Para gerar o analisador sintático, deve-se executar os seguintes comandos:

```
lex lexico.lex
yacc sintatico.y
gcc yy.tab.c -ll -DYYDEBUG=1
```

Isso irá gerar o executável `a.out`, que deve ser executado da seguinte forma:

```
./a.out < código_fonte
```

Testes

Para testar o analisador sintático, foram utilizados dois programas de testes, `test.lp` e `test_proc.lp`. Para a análise de cada programa, foi gerado um arquivo com a saída do analisador. Esses arquivos possuem os mesmos nomes que seus respectivos arquivos de teste, mas com a extensão `.result`.

Conclusão

Este trabalho teve como objetivo construir mais uma parte do compilador, o analisador sintático. Para tal, foi necessário a utilização das ferramentas Lex e YACC. O Lex gera o analisador léxico, enquanto o YACC gera o analisador sintático. Isso proporcionou um bom aprendizado das ferramentas citadas acima, pois agilizam bastante a geração dos analisadores. Por fim, este trabalho tambem proporcionou um melhor entendimento das fases de análise léxica e sintática de um compilador.

Apêndice: entrada para o YACC

Abaixo está o arquivo utilizado com entrada para o YACC:

```
/* declarations */

%{
      #include <stdio.h>
      int yydebug=1;
%}

%debug

%union
{
      int integer;
      float real;
      char *string;
}

%token COMMA
%token COLON
%token SEMICOLON
%token PROC
%token SIN
%token LOG
%token COS
%token ORD
%token ABS
%token SQRT
%token EXP
%token EOFILE
%token EOLN
```

```
%token PROGRAM
%token INTEGER
%token REAL
%token BOOLEAN
%token CHAR
%token VALUE
%token REFERENCE
%token BEGIN_TOK
%token END
%token IF
%token THEN
%token ELSE
%token REPEAT
%token UNTIL
%token READ
%token WRITE
%token FALSE
%token TRUE
%token ATRIB
%token LPAR
%token RPAR
%token NOT
%token EQ
%token NE
%token GT
%token LT
%token GE
%token LE
%token <integer> ADDOP
%token <integer> MULOP
%token <string> ID
%token <integer> INTEGER_CONSTANT
%token <real> REAL_CONSTANT
%token <integer> CHAR_CONSTANT

%start program
%%

/* rules */
program    :       PROGRAM ID SEMICOLON decl_list compound_stmt
                                { printf("reduced program\n"); }
           ;
decl_list   :       decl_list SEMICOLON decl
            |       decl
            |           vazio
            ;
decl    :       dcl_var
        |       dcl_proc
        ;
dcl_var     :       ident_list COLON type
            ;
ident_list  :       ident_list COMMA ID
            |       ID
            ;
type  :           INTEGER
            |           REAL
            |           BOOLEAN
            |           CHAR
```

```
            ;
dcl_proc    :       tipo_retornado PROC ID espec_parametros corpo
            ;
vazio   :
        ;
tipo_retornado  :       INTEGER
                |       REAL
                |       BOOLEAN
                |       CHAR
                |       vazio
                ;
corpo   :       COLON decl_list SEMICOLON compound_stmt id_return
        |       vazio
        ;
id_return   :       ID
            |       vazio
            ;
espec_parametros    :       LPAR lista_parametros RPAR
                    ;
lista_parametros  :           parametro
                  |           lista_parametros COMMA parametro
                  ;
parametro   :           modo type COLON ID
                ;
modo  :           VALUE
            |           REFERENCE
            ;
compound_stmt     :           BEGIN_TOK stmt_list END
                  ;
stmt_list   :           stmt_list SEMICOLON stmt
                |           stmt
                ;
stmt  :           assign_stmt
            |           if_stmt
            |           repeat_stmt
            |           read_stmt
            |           write_stmt
            |           compound_stmt
            |           function_ref_par
            ;
assign_stmt :           ID ATRIB expr
                ;
if_stmt     :           IF cond THEN stmt
            |           IF cond THEN stmt ELSE stmt
            ;
cond  :           expr
            ;
repeat_stmt :           REPEAT stmt_list UNTIL expr
                ;
read_stmt   :           READ LPAR ident_list RPAR
                ;
write_stmt  :           WRITE LPAR expr_list RPAR
                ;
expr_list   :           expr
                |           expr_list COMMA expr
                ;
expr  :           simple_expr
            |           simple_expr EQ simple_expr
```

```
            |           simple_expr NE simple_expr
            |           simple_expr GT simple_expr
            |           simple_expr LT simple_expr
            |           simple_expr GE simple_expr
            |           simple_expr LE simple_expr

            ;
simple_expr :           term
                  |           simple_expr ADDOP term
                  ;
term  :           factor_a
            |           term MULOP factor_a
            ;
factor_a    :           '-' factor
                  |           factor
                  ;
factor      :           ID
            |           constant
            |           LPAR expr RPAR
            |           NOT factor
            |           function_ref_par
            ;
function_ref_par  :           built_in_function LPAR expr RPAR
                              |           variable LPAR expr_list RPAR
                              ;
built_in_function :           SIN
                              |           LOG
                              |           COS
                              |           ORD
                              |           ABS
                              |           SQRT
                              |           EXP
                              |           EOFILE
                              |           EOLN
                              ;
variable    :           simple_variable_or_proc
                  ;
simple_variable_or_proc :           ID
                                    ;
constant    :           INTEGER_CONSTANT
                  |           REAL_CONSTANT
                  |           CHAR_CONSTANT
                  |           boolean_constant
                  ;
boolean_constant  :           TRUE
                              |           FALSE
                              ;
%%
/* programs */
#include "lex.yy.c"
main() {
   yyparse();
   return 0;
}

yyerror(s) char *s; {
      fprintf( stderr, "%s\n", s );
}
```

Bibliografia

http://pt.wikipedia.org/wiki/Análise_sintática_(computação)
http://dinosaur.compilertools.net/lex/index.html
http://dinosaur.compilertools.net/yacc/index.html
http://tldp.org/HOWTO/Lex-YACC-HOWTO-1.html