utad UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

Compiladores

Eng.ª Informática - 2.º ano

Teresa A. Perdicoúlis

Projeto Prático

Implementação de um Analisador Sintático

Diogo Medeiros (70633) Eduardo Chaves (70611)

João Rodrigues (70579) Tiago Lameirão (70636)

janeiro 2021

Resumo

Este projeto prático consistiu no desenvolvimento de um analisador sintático para a linguagem C, envolvendo ferramentas como o Flex e o Bison. Dada a linguagem C ser demasiado extensa, optou-se por desenvolver um analisador capaz de verificar uma versão simplificada da mesma.

O analisador em causa é capaz de detetar erros, quer léxicos, quer sintáticos, informando o utilizador da localização dos mesmos. Nalguns casos, produz uma mensagem mais detalhada, permitindo a correção imediata de falhas tais como a ausência de caracteres de finalização em declarações.

Crê-se que o objetivo proposto foi atingido na medida em que os pressupostos do protocolo foram concretizados.

Conteúdo

1 Introdução **2**

2 Análise e Especificação **3**
2.1 Descrição informal do problema 3
2.2 Especificação dos Requisitos 3
2.2.1 Dados 3
2.2.2 Pedidos 3
2.2.3 Relações 3

3 Conceção da Resolução **4**

4 Codificação e Testes **5**
4.1 Alternativas, Decisões e Problemas de Implementação 5
4.2 Testes realizados e Resultados 7

5 Conclusão **9**

A Anexos dos ficheiros fonte **10**

B Equipa **19**

Capítulo 1

Introdução

Este relatório visa descrever o projeto prático realizado, consistindo o mesmo no desenvolvimento de um analisador sintático para a linguagem C.

Para a realização do trabalho prático, foram usados os conhecimentos adquiridos quer nas aulas teóricas, quer nas aulas práticas, recorrendo também à bibliografia disponibilizada pela docente, bem como outras fontes devidamente identificadas neste relatório.

Ao longo do relatório será descrito o processo de conceção do trabalho bem como algumas considerações pertinentes.

O capítulo 3 descreve não só a solução proposta para o problema em questão, bem como o enquadramento teórico da mesma.

Quanto ao capítulo 4, este detalha a implementação do código, evidenciando as decisões tomadas e justificando as mesmas. Uma vez que o objetivo deste trabalho é a análise e consequente deteção de erros num ficheiro de texto escrito em C, esta análise é demonstrada através de exemplos que evidenciam o funcionamento do analisador sintático construído.

Crê-se ter atingido os pressupostos descritos no protocolo do trabalho, na medida em que o analisador sintático desenvolvido cumpre os fins propostos.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O problema é definido pela análise de um texto e pela sua verificação. Neste caso, a solução traduz-se em implementar um analisador sintático capaz de verificar esse mesmo texto, redigido em linguagem C (numa versão simplificada, *naive* C).

2.2 Especificação dos Requisitos

2.2.1 Dados

Dispõe-se de um texto para análise e verificação de eventuais erros. Este texto é escrito na linguagem C.

Também são disponibilizadas ferramentas de geração de reconhecedores léxicos, no caso do Flex/Lex, e sintáticos, no caso do YACC/Bison.

2.2.2 Pedidos

É solicitada a análise do texto ao nível da sintaxe e do léxico (validade dos lexemas) e verificação de possíveis erros.

2.2.3 Relações

Tal como foi referido acima, a solução passa por desenvolver um analisador sintático auxiliado por um reconhecedor léxico, cujo resultado será o texto analisado e tratado em termos de erros. Para tal, serão usadas ferramentas, nomeadamente, Flex/Lex e YACC/Bison.

Capítulo 3

Conceção da Resolução

Por definição, um analisador léxico tem como função a leitura de caracteres, um a um, do buffer e agrupá-los em lexemas/tokens de acordo com a situação e contexto. Assim, a sequência original de caracteres torna-se numa sequência de tokens que é enviada ao analisador sintático [3].

Para a identificação dos respetivos tokens, torna-se necessária a construção de expressões regulares que caracterizem os diferentes elementos que compõem a linguagem em questão, tais como palavras-chave, operadores, identificadores de variáveis e funções, literais, e até mesmo comentários.

Na prática, dispõe-se da ferramenta Flex/Lex para desenvolver e gerar um reconhecedor léxico para a linguagem C. A cada regra do analisador especificado no flex deverá estar associada a ação de retornar um identificador do token reconhecido ao analisador sintático, para que este possa realizar o parsing.

Por outro lado, o analisador sintático usa os tokens produzidos pelo analisador léxico para criar uma representação intermédia semelhante a uma árvore que evidencia a estrutura gramatical da stream de tokens [1]. Esta representação denomina-se árvore de sintaxe.

Os parsers LR são uma classe de métodos bottom-up que aceita uma amplitude maior de gramáticas relativamente aos reconhecedores LL(1). Por outro lado, estes permitem a declaração externa da precedência dos operadores para resolver questões de ambiguidade, em vez de exigirem que as próprias gramáticas não sejam ambíguas [2]. A maioria dos geradores de parsers LR usa uma versão alargada da construção SLR denominada LALR (1). O "LA"indica uma abreviatura de "lookahead"(oráculo) e o (1) indica que tem a capacidade de antever o próximo símbolo de entrada [2].

A ferramenta utilizada é o YACC, com a capacidade de gerar um AS a partir de uma gramática independente de contexto (GIC), nomeadamente gera um parser LALR(1), pelo que a gramática construída deverá evitar situações tais como a recursividade à direita e a ambiguidade de modo a evitar conflitos.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

Este trabalho desenvolveu-se em várias etapas sucessivas. Inicialmente, procedeu-se à identificação dos tokens de C a serem reconhecidos. Optou-se por identificar palavras-chave relativas à declaração de variáveis e estruturas (`int, char, struct`, etc.), e instruções (`if, for`, etc).

Em termos de operadores de C, fizemos o reconhecimento léxico, não só dos operadores solicitados no protocolo, mas sim de todos os operadores aritméticos, de comparação, lógicos, bitwise, de atribuição e outros operadores de marcação tais como ';', ',', e '.'.

Por outro lado, o nosso reconhecedor léxico reconhece variáveis, literais (números inteiros e reais, caracteres literais e strings), estruturas de dados, funções, e instruções elementares de C. Por questões de simplicidade optou-se por definir as expressões regulares mais extensas na secção de definições do Flex como macros.

Ainda quanto ao analisador léxico, o mesmo possui ERs para filtrar comentários e *whitespace* (indentações, mudanças de linha e espaços) e identificar diretivas do pré-processador de C, nomeadamente `#include`.

Dado o AL funcionar como uma sub-rotina do AS, este deverá devolver, sempre que necessário e adequado, um valor definido pelo AS e associado ao respetivo token. Isto é possível pela diretiva `#include "mycc.tab.h"`, presente nas definições de C do Flex. No caso dos operadores formados por apenas um caractere, optou-se por retornar o mesmo ao AS, dado haver uma correspondência entre este e o respetivo valor decimal, por exemplo, no caso de '=', temos como ação `return *yytext`. No caso dos literais e dos identificadores de variáveis, guardou-se o token identificado pela variável `yytext` na variável `yylval`, do YACC, servindo-se de uma união para distinguir inteiros, reais e

strings, i.e., a ação `yylval.string=strdup(yytext)`. Já na especificação do AS, isto é possível graças ao uso da instrução `%union` e de instruções tais como `%token <string> VAR`.

Relativamente ao analisador sintático, foi usada a ferramenta YACC/Bison que permite gerar um reconhecedor LALR(1) a partir de uma gramática independente de contexto (GIC). Por esta razão, teve-se o cuidado de definir produções com recursividade à esquerda, dado que o reconhecedor deriva mais à direita e assim evita-se a ocupação excessiva da pilha interna do analisador sintático. De referir que se prescindiu de construir um AFD LALR(1) dada a complexidade da gramática desenvolvida bem como da linguagem C.

Na linguagem C, cada operador tem um nível de precedência e um tipo de associatividade ligado ao mesmo. No nosso caso, recorremos a instruções de declaração de tokens, tais como `%token`, `%left` e `%right`, a fim do analisador sintático reconhecer esta informação para cada operador, resolvendo assim problemas de ambiguidade e possíveis conflitos, `%left '+'` e `%right '='`, presentes nas definições do ficheiro "mycc.y". O uso de tokens definidos por `%nonassoc`, por exemplo, `%nonassoc UMINUS`, permite a distinção entre operadores distintos que partilham a mesma grafia, por exemplo o operador de subtração '-' e o operador unário '-', em conjunto com a instrução `%prec`, presente na respetiva produção, i.e., `alg_cond_expr : '-' alg_cond_expr %prec UMINUS`.

Relativamente ao tratamento de erros, no caso do AL, optou-se por identificar e contabilizar os erros léxicos presentes, sem informar o AS da sua existência permitindo a este prosseguir com o *parsing*, ainda que no final, o utilizador seja informado de que o *parsing* não foi bem-sucedido.

A ferramenta YACC/Bison recorre a uma função, `yyerror`, sempre que é detetado um erro no *parsing*. Na nossa especificação, são identificados erros de sintaxe detalhados, graças às diretivas `%define parse.lac none` e `%define parse.error verbose`, presentes nas definições do "mycc.y". Quando um erro sintático é detetado, o *parsing* termina e o utilizador é informado da linha em que foi detetado o erro e, por vezes, a causa do mesmo. Como a linguagem C ignora a existência de *whitespace*, sendo este unicamente útil para o programador estruturar o seu código, ocorre que o erro pode ser detetado na linha seguinte, o que dificulta a identificação por parte do utilizador. Contudo, este tipo de identificação é comum à maioria dos compiladores de C, i.e., o GCC (GNU Compiler Collection).

4.2 Testes realizados e Resultados

Apresentam-se, de seguida, alguns exemplos dos testes realizados, demonstrando a deteção de erros, quer léxicos, quer sintáticos.

Exemplo 1

Neste exemplo verifica-se a análise bem sucedida, demonstrando a inexistência de erros.

Input:

```
#include <stdio.h>

struct aluno
{
        int num;
        char nome;
};

int main() {
        int num;        /*declaracao de inteiro com nome num*/
        float real;
        char c = 'a';
        double ec;
        printf("Vou fazer um ciclo");
        for (i=1;i<10;i=i+1){
                printf("ola", a+5);
        }
        while ( num >= real)
        {
                b++;
                a=b/c;
        }
        if( c != '\n'+4.8)
                a%=a*(5+b) & c;
        else
                return 0;
}
```

Output:

```
Parsing Successful.
```

Exemplo 2

Este exemplo evidencia a deteção de um erro léxico e um erro sintático, explicitando as respetivas linhas.

Input:

```
1  void main()
2  {
3    int a, b=2|8;
4    f = 4ec + 78*(-c);
5    if(!f < g)
6    {
7      for(;;)
8            printf("Teste 2\n", );
9    }
10   return;
11 }
```

Output:

```
Line:  4
Error:  Invalid token in '4ec'
Line:  8
Error:  syntax error, unexpected ')'
Found 1 lexical errors.  Parsing Failed.
```

Exemplo 3

Este exemplo evidencia a ocorrência dum erro sintático de finalização de declaração.

Input:

```
1  main() {
2      do
3        for(i=10; i>0; i--)
4          if(b>=c)
5            printf("Teste 3\n", b+c);
6      while(1)
7      return (-1); }
```

Output:

```
Line:  7
Error:  syntax error, unexpected RETURN, expecting ';'
Parsing Failed.
```

Capítulo 5

Conclusão

Consideramos que este trabalho foi um desafio na medida em que o protocolo solicitava a execução de um AS que verifique uma versão simplificada da linguagem C, discriminando os conteúdos presentes nos programas a testar, no entanto, o grupo esforçou-se para incluir outras especificações de C, enriquecendo assim o projeto desenvolvido.

Outra exceção ao protocolo, foi a inclusão de ações adicionais ao AL em conformidade com a especificação da variável global do YACC, `yylval`, que, embora não surtam efeito presentemente, preparam o AS para ser adaptado futuramente e permitir o funcionamento da fase seguinte de compilação, a análise semântica. Uma possível alteração a fazer ao AS seria guardar a informação relativa a variáveis tais como o tipo, valor e identificador, e a funções, tais como o tipo de valor retornado, o identificador e os seus parâmetros. Esta informação seria vantajosa pelo facto de ser armazenada numa tabela de símbolos e posteriormente usada pelo Asem para identificar erros semânticos.

De referir que não foram incluídos, neste trabalho, algoritmos ou estruturas de dados pelo facto de serem usadas as ferramentas Flex e Bison, prescindindo da sua execução.

Em suma, se, por um lado aprofundámos os nossos conhecimentos relativamente ao sistema de preparação de documentos LaTeX, melhorando significativamente o uso do mesmo, por outro a execução do projeto prático tornou-se aliciante no que concerne ao uso das ferramentas de auxílio à construção de reconhecedores, pelo que o código desenvolvido foi constantemente alterado e melhorado à medida que foi revisto, acompanhando a evolução das nossas competências.

Apêndice A

Anexos dos ficheiros fonte

Especificação do analisador léxico em Flex.

```
%option noyywrap yylineno

%{
        #include <string.h>
        #include "mycc.tab.h"
        int lex_error = 0;
        int lexical_error(void);
%}

Var                             [_a-zA-Z][_a-zA-Z0-9]*
VarInvalid                      [0-9][_0-9a-zA-Z]+
Int                             [0-9]+
Real                            [0-9]+(\.[0-9]+)?([Ee][+-]?[0-9]+)?
String                          \"(\\(.|\n)|[^\n\\"])*\"
Char                            \'([^\'\\]|[\\]n|[\\]t|[\\]\'|[\\][\\])\'
CharInvalid                     \'[\'\\]\'|\'(..+)\'
Header                          "#include"([ ]+)?((<(\\.|[^>])+>)|(\"(\\.|[^"])+\"))
SingleLineComment               \/\/.*
MultiLineComment                "/*"(.|\n)*?"*/"
Whitespace                      [ \t\r\n]+
Marks                           [}{(),;]
Operators                       [\^|&~.!<=>+\-*\/%]

%%

{MultiLineComment}              |

{SingleLineComment}             |

{Whitespace}                    ;
```

```
{Header}                    {return HEADER;}
int                         {return INT;}
float                       {return FLOAT;}
double                      {return DOUBLE;}
char                        {return CHAR;}
void                        {return VOID;}
struct                      {return STRUCT;}
if                          {return IF;}
else                        {return ELSE;}
for                         {return FOR;}
do                          {return DO;}
while                       {return WHILE;}
return                      {return RETURN;}
printf                      {return PRINTF;}
{Var}                       {yylval.string=strdup(yytext); return VAR;}
{Int}                       {yylval.ival=atoi(yytext); return NINT;}
{Real}                      {yylval.dval=atof(yytext); return NREAL;}
{String}                    {yylval.string=strdup(yytext); return STRING;}
{Char}                      {yylval.string=strdup(yytext); return CHARL;}
{Marks}                     |
{Operators}                 {return *yytext;}
"+="                        {return ADD_ASG;}
"-="                        {return SUB_ASG;}
```

```
"*="                            {return MUL_ASG;}

"/="                            {return DIV_ASG;}

"%="                            {return MOD_ASG;}

"&="                            {return AND_ASG;}

"|="                            {return OR_ASG;}

"^="                            {return XOR_ASG;}

"<<="                           {return LEFSFT_ASG;}

">>="                           {return RIGSFT_ASG;}

"<<"                            {LEFT_SHIFT;}

">>"                            {RIGHT_SHIFT;}

"<="                            {return LEQ;}

">="                            {return GEQ;}

"=="                            {return EQ;}

"!="                            {return NEQ;}

"&&"                            {return AND;}

"||"                            {return OR;}

"++"                            {return INC;}

"--"                            {return DEC;}

{VarInvalid}                    |

{CharInvalid}                   |

.                               {lexical_error();}

%%

int lexical_error(void)
{
```

```
        printf("Line: %d\nError: Invalid token '%s'\n", yylineno, yytext);
        ++lex_error;
        return(0);
}
```

Especificação do analisador sintático em Bison.

```
%define parse.lac none
%define parse.error verbose

%{
        #include <stdio.h>
        int yylex(void);
        int yyerror(const char *s);
        extern int lex_error;
        int test = 1;
%}

%union {
        int ival;
        double dval;
        char *string;
}

%token HEADER
%token INT FLOAT DOUBLE CHAR VOID STRUCT
%token <string> VAR
%token <ival> NINT
%token <dval> NREAL
%token <string> STRING
%token <string> CHARL
%token IF FOR DO WHILE RETURN PRINTF
%nonassoc REDUCE
%nonassoc ELSE
%left ','
%right '=' MUL_ASG DIV_ASG MOD_ASG ADD_ASG SUB_ASG
           AND_ASG OR_ASG XOR_ASG LEFSFT_ASG RIGSFT_ASG
%left OR
%left AND
%left '|'
%left '^'
%left '&'
%left EQ NEQ
%left LEQ '<' '>' GEQ
%left LEFT_SHIFT RIGHT_SHIFT
%left '+' '-'
```

```
%left '*' '/' '%'
%nonassoc UPLUS UMINUS
%right '!' '~'
%left INC DEC '.'

%%

program : decl_list
        | header_list decl_list
        ;

header_list : HEADER
            | header_list HEADER
            ;

decl_list : declaration
          | decl_list declaration
          ;

declaration : struct
            | function
            ;

struct : STRUCT VAR '{' struct_args '}' ';'
       ;

struct_args :
            | struct_args type struct_arg ';'
            ;

struct_arg : VAR
           | struct_arg ',' VAR
           ;

function : function_type VAR '(' function_args ')' closed_statement
         ;

function_type :
              | VOID
              | type
              ;

function_args :
              | VOID
              | arguments
              ;
```

```
arguments : type VAR
          | arguments ',' type VAR
          ;

type : INT
     | FLOAT
     | DOUBLE
     | CHAR
     ;

closed_statement : '{' '}'
                 | '{' statements '}'
                 ;

statements : statement
           | statements statement
           ;

statement : definition
          | expression_stmt
          | if_cond
          | for
          | while
          | do_while
          | return
          | printf
          | closed_statement
          ;

definition : type def_list ';'
           | STRUCT VAR struct_def_list ';'
           ;

def_list : def_item
         | def_list ',' def_item
         ;

def_item : VAR
         | VAR '=' expr
         ;

struct_def_list : VAR
                | struct_def_list ',' VAR
                ;
```

```
expression_stmt : ';'
                | expression ';'
                ;

expression : expr
           | VAR '=' expr
           | VAR MUL_ASG expr
           | VAR DIV_ASG expr
           | VAR MOD_ASG expr
           | VAR ADD_ASG expr
           | VAR SUB_ASG expr
           | VAR LEFSFT_ASG expr
           | VAR RIGSFT_ASG expr
           | VAR AND_ASG expr
           | VAR OR_ASG expr
           | VAR XOR_ASG expr
           ;

expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | expr '%' expr
     | expr AND expr
     | expr OR expr
     | expr EQ expr
     | expr NEQ expr
     | expr LEQ expr
     | expr GEQ expr
     | expr '<' expr
     | expr '>' expr
     | expr '&' expr
     | expr '|' expr
     | expr '^' expr
     | expr LEFT_SHIFT expr
     | expr RIGHT_SHIFT expr
     | expr INC
     | expr DEC
     | '!' expr
     | INC expr %prec '!'
     | DEC expr %prec '!'
     | '-' expr %prec UMINUS
     | '+' expr %prec UPLUS
     | '(' expr ')'
     | id
     ;
```

```
id : VAR
   | VAR '.' VAR
   | NINT
   | NREAL
   | CHARL
   ;

if_cond : IF '(' expression ')' statement %prec REDUCE
        | IF '(' expression ')' statement ELSE statement
        ;

for : FOR '(' expression_stmt expression_stmt ')' statement
    | FOR '(' expression_stmt expression_stmt expression ')' statement
    ;

while : WHILE '(' expression ')' statement
      ;

do_while : DO statement WHILE '(' expression ')' ';'
         ;

return : RETURN expression_stmt
       ;

printf : PRINTF '(' STRING ')' ';'
       | PRINTF '(' STRING ',' print_expr ')' ';'
       ;

print_expr : expression
           | print_expr ',' expression
           ;

%%

int main()
{
        yyparse();
        if(lex_error > 0)
                printf("Found %d lexical errors. Parsing Failed.\n", lex_error);
        else if(test)
                printf("Parsing Successful.\n");
  else
    printf("Parsing Failed.\n");
        return 0;
}
```

```
int yyerror(const char *msg)
{
        extern int yylineno;
        printf("Line: %d\nError: %s\n", yylineno, msg);
        test = 0;
        return(0);
}
```

Especificação do programa de compilação.

Embora este ficheiro não tenha sido solicitado, decidiu-se anexar o mesmo por constar do trabalho prático.

```
NAME = mycc

CC = gcc
CFLAGS = -g
LEX = flex
LEXFLAGS = -i
YACC = bison
YACCFLAGS = -dtv

all: ${NAME}

${NAME}: ${NAME}.tab.c lex.yy.c
        ${CC} ${NAME}.tab.c lex.yy.c -o ${NAME}

lex.yy.c: ${NAME}.l ${NAME}.tab.h
        ${LEX} ${LEXFLAGS} ${NAME}.l

${NAME}.tab.c: ${NAME}.y
        ${YACC} ${YACCFLAGS} ${NAME}.y

clean:
        rm -f lex.yy.c lex.yy.o ${NAME} ${NAME}.tab.c ${NAME}.tab.h ${NAME}.output
        rm -f ${NAME}.tab.o
```

Apêndice B

Equipa

Relativamente ao grupo, o mesmo foi constituído pelos alunos, Diogo Medeiros, Tiago Lameirão, Eduardo Chaves e João Rodrigues.

Quanto à respetiva contribuição individual, embora determinados campos tenham sido aprofundados por alguns membros, na generalidade, todos acompanharam o desenvolvimento do trabalho, contribuindo pontualmente com sugestões.

Podemos individualizar a atribuição da especificação do reconhecedor sintático usando a ferramenta YACC/Bison ao membro Diogo Medeiros; a especificação do analisador léxico usando a ferramenta Flex/Lex ao membro Tiago Lameirão; e a realização do relatório aos membros Eduardo Chaves e João Rodrigues, frisando, mais uma vez, a contribuição de todos os membros na realização, quer do trabalho, quer do respetivo relatório.

Bibliografia

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science. Addison-Wesley, 1986.

[2] T.Æ. Mogensen and Københavns universitet. Department of Computer Science. *Basics of Compiler Design*. Torben Ægidius Mogensen, 2009.

[3] Y. Su and S.Y. Yan. *Principles of Compilers: A New Approach to Compilers Including the Algebraic Method*. Springer Berlin Heidelberg, 2011.