CES41: Compilador Completo C-

Erick de Araujo Coelho

COMP-23

Introdução

O objetivo deste trabalho é implementar um compilador completo para a linguagem C-. Para isso foi usado o flex para a construção do scanner e o bison para a construção do parser. Assim sendo, caso não haja nenhum erro, o módulo de análise léxica identifica os tokens, a partir disso o analisador sintático constrói a árvore sintática e o analisador semântico percorre a árvore sintática buscando erros sensíveis a contexto e erros de tipo e, por fim, constrói a tabela de símbolos. Caso não haja nenhum erro léxico, sintático ou semântico no código, o gerador de código percorre a árvore sintática e constrói o código intermediário.

Para a construção do compilador de código C- foi utilizado como base o projeto do compilador de código Tiny disponibilizado no Classroom da disciplina CES-41 dentro do arquivo *TinyCompleto.zip*.

Analisador Léxico

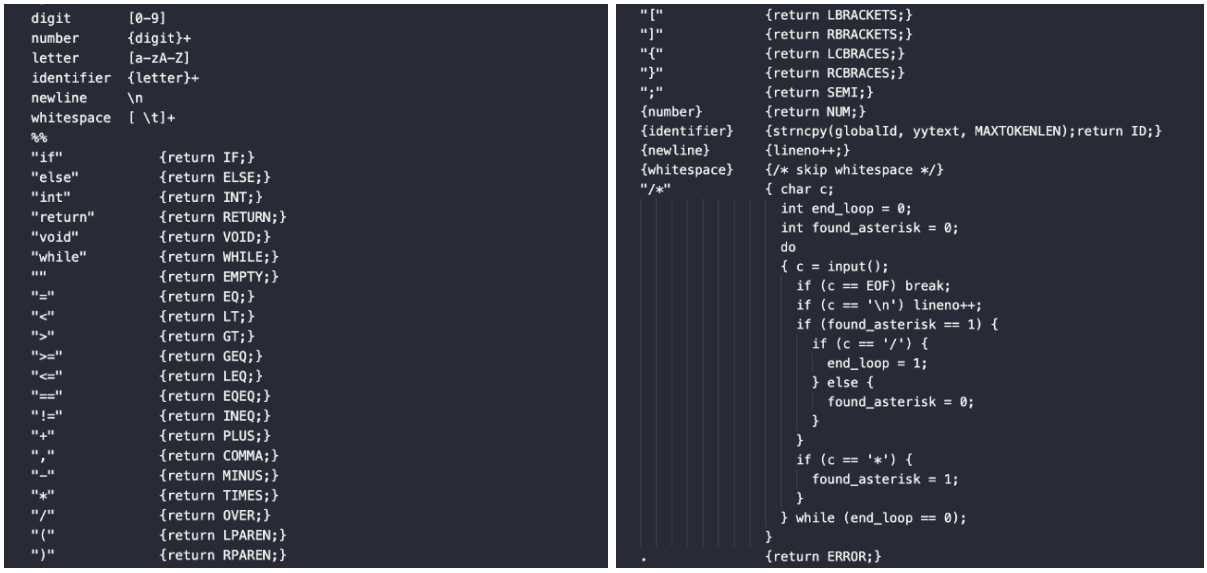
O objetivo do analisador léxico é classificar uma sequência de caracteres, atribuindo um token a essa sequência de caracteres. No caso da linguagem C- são identificados:

- Palavras-chave: if, else, int, return, void, while
- Símbolos especiais: +, -, *, /, <, <=, >, >=, =, !=, ==, ;, ,, (,), [,], {, }, /*, */
- Identificadores: uma ou mais letras
- Números: um ou mais dígitos
- Espaço em branco (ignorados): espaços, tabulação, quebra de linha
- Comentários: texto cercado por /* e */

A identificação de um token é feita através de um autômato finito que pode ser ilustrado graficamente ou de forma escrita onde é descrito as regras que descrevem as possíveis sequências de caracteres que são de uma determinada classe, assim atribuindo um token a ela.

Para a implementação do analisador léxico, é necessária a criação de um arquivo de entrada do Flex que é um gerador de analisador léxico. O arquivo é composto por três partes: definições, regras e rotinas do usuário.

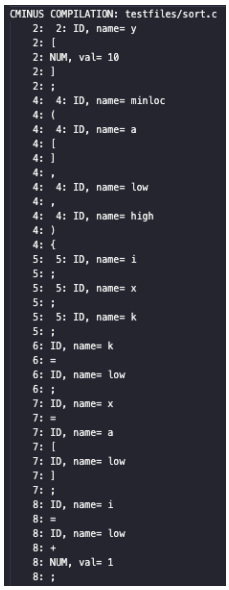
No projeto o arquivo Flex da linguagem tiny, tiny.l, foi adaptado dando origem ao arquivo cminus.l para a linguagem C-. As principais alterações foram feitas na parte das regras e estão ilustradas na imagem abaixo.



```
digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+
%%
"if"            {return IF;}
"else"          {return ELSE;}
"int"           {return INT;}
"return"        {return RETURN;}
"void"          {return VOID;}
"while"         {return WHILE;}
""              {return EMPTY;}
"="             {return EQ;}
"<"             {return LT;}
">"             {return GT;}
">="            {return GEQ;}
"<="            {return LEQ;}
"=="            {return EQEQ;}
"!="            {return INEQ;}
"+"             {return PLUS;}
","             {return COMMA;}
"-"             {return MINUS;}
"*"             {return TIMES;}
"/"             {return OVER;}
"("             {return LPAREN;}
")"             {return RPAREN;}
"["             {return LBRACKETS;}
"]"             {return RBRACKETS;}
"{"             {return LCBRACES;}
"}"             {return RCBRACES;}
";"             {return SEMI;}
{number}        {return NUM;}
{identifier}    {strncpy(globalId, yytext, MAXTOKENLEN);return ID;}
{newline}       {lineno++;}
{whitespace}    {/* skip whitespace */}
"/*"            { char c;
                  int end_loop = 0;
                  int found_asterisk = 0;
                  do
                  { c = input();
                    if (c == EOF) break;
                    if (c == '\n') lineno++;
                    if (found_asterisk == 1) {
                      if (c == '/') {
                        end_loop = 1;
                      } else {
                        found_asterisk = 0;
                      }
                    }
                    if (c == '*') {
                      found_asterisk = 1;
                    }
                  } while (end_loop == 0);
                }
.               {return ERROR;}
```

Figura 1. Código do arquivo cminus.l

Para gerar o analisador sintático é necessário rodar no terminal o comando flex cminus.l que gera o arquivo lex.yy.c onde são implementados o autômato a partir das regras descritas. Para gerar o executável compilamos o arquivo e ligamos ele a biblioteca libfl. Para exibir a os tokens identificados a função printToken no arquivo util.c foi adaptada da linguagem tiny para a linguagem C-. O resultado obtido está ilustrado na imagem abaixo:



```
CMINUS COMPILATION: testfiles/sort.c
    2:  2: ID, name= y
    2: [
    2: NUM, val= 10
    2: ]
    2: ;
    4:  4: ID, name= minloc
    4: (
    4:  4: ID, name= a
    4: [
    4: ]
    4: ,
    4:  4: ID, name= low
    4: ,
    4:  4: ID, name= high
    4: )
    4: {
    5:  5: ID, name= i
    5: ;
    5:  5: ID, name= x
    5: ;
    5:  5: ID, name= k
    5: ;
    6: ID, name= k
    6: =
    6: ID, name= low
    6: ;
    7: ID, name= x
    7: =
    7: ID, name= a
    7: [
    7: ID, name= low
    7: ]
    7: ;
    8: ID, name= i
    8: =
    8: ID, name= low
    8: +
    8: NUM, val= 1
    8: ;
```

Figura 2. Trecho do arquivo sort.txt obtido através da análise léxica do código de ordenação (sort.c) disponibilizado no classroom

Analisador Sintático

O analisador sintático verifica se os tokens foram usados na sequência correta e de forma completa. Para isso é montada a árvore sintática do código através das regras da linguagem que está sendo compilada. Para descrever as regras da linguagem comumente é usada a forma de Backus-Naur (BNF). Na imagem abaixo está a gramática BNF do C- que foi fornecida no Classroom.

```
programa → declaração-lista
declaração-lista → declaração-lista declaração | declaração
declaração → var-declaração | fun-declaração
var-declaração → tipo-especificador ID ; | tipo-especificador ID [ NUM ] ;
tipo-especificador → int | void
fun-declaração → tipo-especificador ID ( params ) composto-decl
params → param-lista | void
param-lista → param-lista, param | param
param → tipo-especificador ID | tipo-especificador ID [ ]
composto-decl → { local-declarações statement-lista }
local-declarações → local-declarações var-declaração | vazio
statement-lista → statement-lista statement | vazio
statement → expressão-decl | composto-decl | seleção-decl | iteração-decl | retorno-decl
expressão-decl → expressão ; | ;
seleção-decl → if ( expressão ) statement | if ( expressão ) statement else statement
iteração-decl → while ( expressão ) statement
retorno-decl → return ; | return expressão;
expressão → var = expressão | simples-expressão
var → ID | ID [ expressão ]
simples-expressão → soma-expressão relacional soma-expressão | soma-expressão
relacional → <= | < | > | >= | == | !=
soma-expressão → soma-expressão soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → * | /
fator → ( expressão ) | var | ativação | NUM
ativação → ID ( args )
args → arg-lista | vazio
arg-lista → arg-lista , expressão | expressão
```

Figura 3. Gramática BNF para C-

Essas regras são descritas no código cminus.y, adaptado do código tiny.y fornecido como é ilustrado nos exemplos abaixo obtidos a partir de trechos do código.

```
program     : stmt_seq
                { savedTree = $1;}
            ;
stmt_seq    : stmt_seq SEMI stmt
                { YYSTYPE t = $1;
                  if (t != NULL)
                  { while (t->sibling != NULL)
                       t = t->sibling;
                    t->sibling = $3;
                    $$ = $1; }
                    else $$ = $3;
                }
            | stmt  { $$ = $1; }
            ;
stmt        : if_stmt { $$ = $1; }
            | repeat_stmt { $$ = $1; }
            | assign_stmt { $$ = $1; }
            | read_stmt { $$ = $1; }
            | write_stmt { $$ = $1; }
            | error  { $$ = NULL; }
            ;
```

Figura 4. Trecho do código cminus.y

Ao rodar no terminal o comando bison -d cminus.y, esse arquivo é processado pelo bison, construtor de parser, gerando os arquivos cminus.tab.c e cminus.tab.h. Para exibir a

árvore sintática a função printTree no arquivo util.c foi adaptada da linguagem tiny para a linguagem C-. O resultado obtido está ilustrado na imagem abaixo:

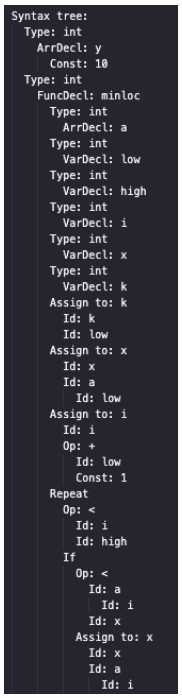


Figura 5. Trecho do arquivo sort.txt exibindo a árvore sintática obtida através da análise sintática do código de ordenação (sort.c) disponibilizado no classroom

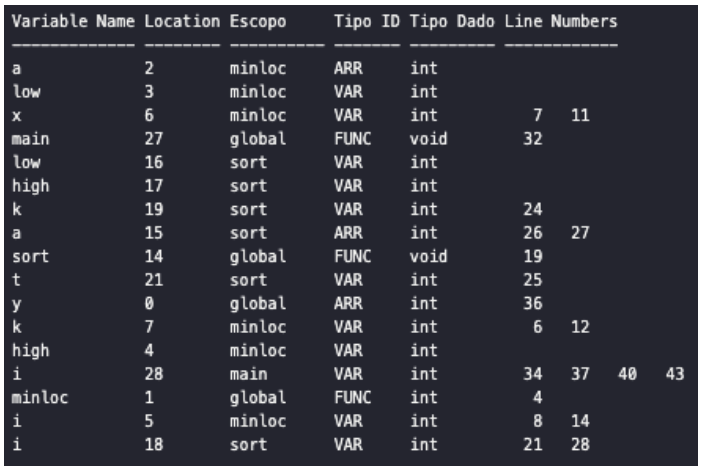
Analisador Semântico

Como a definição da sintaxe é livre de contexto, o analisador semântico serve para identificar as regras sensíveis a contexto da linguagem como a compatibilidade de tipos de variáveis, regras de escopo e se a variável foi declarada antes de ser usada. Para isso é necessário percorrer a árvore sintática e criar uma tabela de símbolos para registrar informações sobre cada variável declarada no código.

Para a análise semântica as principais alterações foram feitas nos arquivos analyse.c e symtab.c a partir da implementação do escopo e da criação da tabela de símbolos que guarda onde cada variável foi declarada (linha e escopo) e onde ela está sendo usada.

A estrutura de dados BucketList é usada para salvar as informações de cada variável e é feito um hash a partir do nome e do escopo de cada variável. Para fazer qualquer verificação ou alteração na tabela de símbolos é necessário percorrer o hash e encontrar a variável desejada e caso a variável não seja encontrada pode resultar em um erro ou na inserção de uma nova variável. Para a identificação de erros foram criadas funções auxiliares específicas para os possíveis erros. A função st_lookup encontra a posição da variável e a função st_insert insere uma nova variável.

Para exibir a tabela de símbolos a função printSymTab no arquivo symtab.c foi adaptada da linguagem tiny para a linguagem C-. O resultado obtido está ilustrado na imagem abaixo:



Variable Name	Location	Escopo	Tipo ID	Tipo Dado	Line Numbers
a	2	minloc	ARR	int	
low	3	minloc	VAR	int	
x	6	minloc	VAR	int	7 11
main	27	global	FUNC	void	32
low	16	sort	VAR	int	
high	17	sort	VAR	int	
k	19	sort	VAR	int	24
a	15	sort	ARR	int	26 27
sort	14	global	FUNC	void	19
t	21	sort	VAR	int	25
y	0	global	ARR	int	36
k	7	minloc	VAR	int	6 12
high	4	minloc	VAR	int	
i	28	main	VAR	int	34 37 40 43
minloc	1	global	FUNC	int	4
i	5	minloc	VAR	int	8 14
i	18	sort	VAR	int	21 28

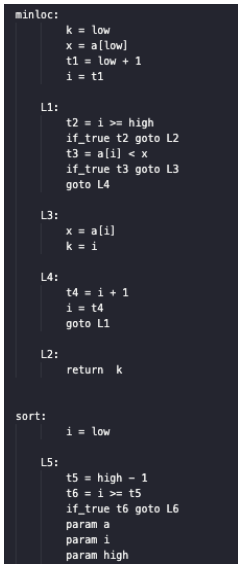
Figura 6. Trecho do arquivo sort.txt exibindo a tabela de símbolos obtida através da análise semântica do código de ordenação (sort.c) disponibilizado no classroom

Gerador de Código

Por fim, caso não haja nenhum erro léxico, sintático ou semântico no código, a árvore sintática é percorrida para a produção do código intermediário, linearização da árvore sintática. Esse código é a primeira etapa da fase de síntese do compilador que tem por objetivo chegar no código de máquina (Assembly).

Para a geração do código intermediário foram feitas alterações no arquivo cgen.c que define funções recursivas específicas para cada tipo de nó da árvore sintática, visto que cada tipo possui uma organização onde seus filhos são partes necessárias para completar a função como por exemplo as regras de uma condicional ou de uma repetição.

O resultado obtido pelo gerador de código está ilustrado na imagem abaixo:



```
minloc:
        k = low
        x = a[low]
        t1 = low + 1
        i = t1

    L1:
        t2 = i >= high
        if_true t2 goto L2
        t3 = a[i] < x
        if_true t3 goto L3
        goto L4

    L3:
        x = a[i]
        k = i

    L4:
        t4 = i + 1
        i = t4
        goto L1

    L2:
        return  k


sort:
        i = low

    L5:
        t5 = high - 1
        t6 = i >= t5
        if_true t6 goto L6
        param a
        param i
        param high
```

Figura 7. Trecho do arquivo sort.cmm exibindo o código intermediário do código de ordenação (sort.c) disponibilizado no classroom

Conclusão

A partir das adaptações descritas foi possível adaptar o compilador de tiny, criando um compilador completo de C- capaz de identificar os tokens, construir uma árvore sintática, construir uma tabela de símbolos e por fim entregar um código intermediário, além de identificar todos erros léxicos, sintáticos e semânticos bem como onde ocorrem esses erros.

O único ponto que faltou no projeto foi a criação de uma variável temporária em declarações de arrays conforme ilustrado abaixo.

Código em C-	**Como deveria ficar:**	**Como ficou:**
exarr[5] = 10;	t1 = 5 * 4; exarr[t1] = 10;	exarr[5] = 10;

Link do repositório: https://github.com/ErickCoelho/Compiler-Project