Relatório - Analisador Léxico de C*

Nicholas Nishimoto Marques - 15/0019343

05/09/2020

1 Introdução e Área da Computação

A linguagem proposta para o trabalho é a C*, consistindo em uma extensão da linguagem C com a adição de operações sobre tuplas [6] , onde seria possível declarar tuplas com tipos diferentes de dados, somar as tuplas (posição à posição), subtrair, em geral realizar operações básicas nas tuplas. Com isso seria possível otimizar muitos projetos de código, possibilitando arquiteturas de software novas e mais organizadas para os desenvolvedores, otimizando construções em Engenharia de Software.

A linguagem C* destina-se ao interesse do aluno na sub-área da computação de Engenharia de Software. É uma área que atualmente é responsável por projetar a arquitetura dos softwares, o modo como ele será implementado, os testes à serem feitos e a manutenção do mesmo. Tais implementações de estrutura de dados permitem uma maior flexibilidade de operações, organizando o código gerado, que é a ideia por trás das linguagens de alto nível.

As dificuldades que podem ser encontradas são fazer as operações em tuplas somando cada componente de forma certa e tratar erros que possam acontecer em cada caso, já que elas podem ter tipos diferentes de dados a cada posição.

2 Gramática da Linguagem

A gramática proposta se baseia na gramática do ANSI C [2], que consiste em gramáticas de termos e predicados [3] e dos tipos e expressões. A gramática incluiu as expressões a mais necessárias para as tuplas, representadas pelos predicados tuple.

Todos os terminais estão em negrito, sequências entre chaves { } são indicativos de possíveis repetições, e sequências entre colchetes [ ] são indicativos de opcionalidade, adicionados à gramática a lógica de escrita e leitura da linguagem.

1. $letra \coloneqq a \mid b \mid ... \mid z \mid A \mid B \mid ... \mid Z$
2. $digito \coloneqq 0 \mid 1 \mid ... \mid 9$
3. $\mathbf{id} \coloneqq letra \ \{ \ letra \mid digito \mid \_ \ \}$
4. $programa \rightarrow \ \{ \ declaração \ ';' \mid func\_decl \ \}$
5. $declaração \rightarrow$
   $\mid tipo \ var\_decl \{ \ ';' \ var\_decl \ \}$
   $\mid \mathbf{tuple} \ tipo \ \{ \ , \ tipo \ \} \ var\_decl$

6. *var_decl* → **id**

7. *tipo* → **float** | **int** | **char**

8. *parm_tipos* → **void** | *tipo* **id**['[' ']'] { ',' *tipo* **id**['[' ']'] }

9. *func_decl* → *tipo* **id** '(' *parm_tipos* ')' "{*tipovar_decl*{','*var_decl*}';'}{*cod_block*}"
   | **void id** '(' *parm_tipos* ')' "{*tipovar_decl*{','*var_decl*}';'}{*cod_block*}"

10. *cod_block* →
    | **if** '(' *expressão* ')' *cod_block* [**else** *cod_block*]
    | **while** '(' *expressão* ')' *cod_block*
    | **for** '(' [*assign*] ';' [*expressão*] ';' [*assign*] ')' *cod$_b$lock*
    | **return** [*expressão*] ';'
    | *assign* ';'
    | **id** '(' [*expressão* ',' *expressão*] ')' ';'
    | " *cod$_b$lock* "
    | *scan*
    | *print*
    | ';'

11. *assign* → **id** ['[' *expressão* ']'] = *expressão*

12. *expressão* →
    | '–' *expressão*
    | '**!**' *expressão*
    | '&' *expressão*
    | '|' *expressão*
    | *expressão op_aritm expressão*
    | *expressão op_comp expressão*
    | **id** ['(' [*expressão* ',' *expressão*] ')' | '[' *expressão* ']']
    | '(' *expressão* ')'

13. *op_aritm* →
    | +
    | −
    | *
    | /

14. *op_comp* →
    | ==
    | ! =
    | <=
    | <
    | >=
    | >

15. $scan \rightarrow$
    | $scan(id)$

16. $print \rightarrow$
    | $print(\{digito\}\{letra\})$
    | $print(id)$

3 Motivação para a escolha da linguagem/área: que tipos de problemas resolve

A área escolhida é a da Engenharia de Software, área que vem se ampliando cada vez mais e especializada em arquiteturas de códigos, modos de automatizar integrações e melhores maneiras de se fazer testes para um projeto de software. O tipo de problema que a linguagem proposta resolve se dá nessa área, que consiste em otimizar interações de dados que necessitam dessa estrutura (como por exemplo operações de vetores físicos, ou mesmo para criação de objetos de uma classe, por exemplo, um aluno seria um tupla: ("João", 16, "masculino")).

Isso inclui também problemas de operações sobre essas tuplas, como por exemplo a soma de vetores (somar cada atributo da tupla), ou então fazer a média das notas por idade dos alunos (fazer média de todas as notas de alunos que tem o atributo idade da tupla iguais), etc.

4 Descrição breve da semântica da linguagem.

Palavras reservadas da linguagem: int float void tuple if else while return

Símbolos: + - * / ¿ ¿= ¡ ¡= == != = ( ) [ ] ;

A palavra reservada int remete à declaração de uma variável inteira.

A palavra reservada float remete à declaração de uma variável ponto flutuante.

A palavra reservada void remete à declaração de uma variável sem tipo.

A palavra reservada tuple remete à declaração de uma tupla, seguida pelos tipos das variáveis da tupla (exemplo: tuple int,float remete à uma dupla de inteiro com ponto flutuante).

A palavra reservada if seguida de uma expressão booleana remete à uma condição em que, se a expressão booleana que segue o if for verdadeira, então todo bloco de código seguinte ao mesmo será executado.

A palavra reservada else remete à um trecho de código que será executado caso o trecho de código presente num bloco if anterior não seja executado.

A palavra reservada while seguida de uma expressão booleana remete à uma condição em que, enquanto à expressão booleana for verdadeira, o bloco de código seguinte continuará sendo executado.

A palavra reservada return remete à um retorno da chamada de uma função, que é o parâmetro seguinte à palavra return.

Regras de Escopo:

Os escopos na linguagem são definidos a partir da delimitação por chaves após palavras reservadas que indicam um bloco (todas as da sintaxe de cod block ou func decl). Por exemplo:

void id { // escopo de bloco (função) }

if (expressão) { // escopo de bloco }

5 Análise Léxica

5.1 Implementação e Funcionamento do programa tradutor

A linguagem proposta para o trabalho é a C*, consistindo em uma extensão da linguagem C com a adição de operações sobre tuplas [6] , onde seria possível declarar tuplas com tipos diferentes de dados, somar as tuplas (posição à posição), subtrair, em geral realizar operações básicas nas tuplas. Com isso seria possível otimizar muitos projetos de código, possibilitando arquiteturas de software novas e mais organizadas para os desenvolvedores, otimizando construções em Engenharia de Software.

Para isso será feito um programa tradutor para tal linguagem. O Programa tradutor irá funcionar seguindo as etapas de tradução de Análise Léxica, Análise Sintática, Análise Semântica, Gerador de Código intermediário e Gerador de Código de Máquina, como citado em Compilers: Principles, Techniques and Tools [1].

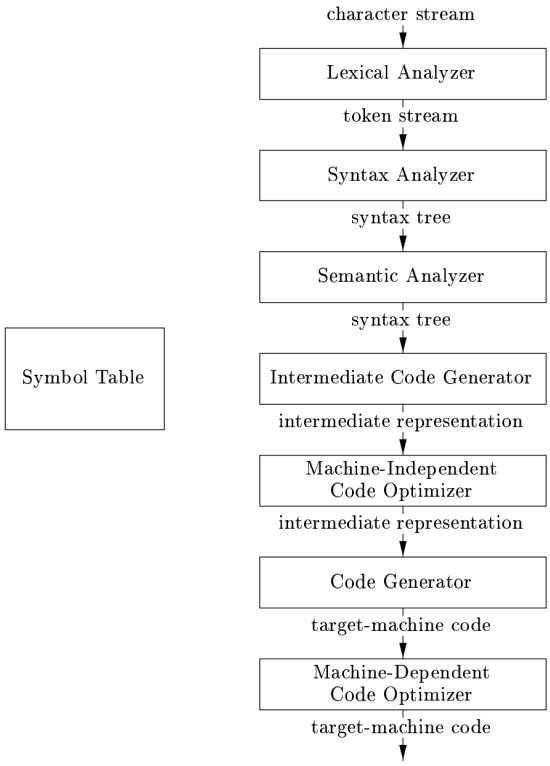


Figura 1: Etapas de Tradução

Na primeira etapa será implementado o analisador léxico. Para essa parte, será utilizado o programa Flex [4] para gerar um programa analisador léxico na linguagem C, a partir da escrita do arquivo .l (lex language [7]) onde é descrita a gramática de C* e as regras dos lexemas.

Na figura 1 pode ser visto algumas definições importantes que serão importantes na formação dos tokens no processo de análise léxica, como as definições de tipo (float, int, char, bool, void, tuple) e operações, por exemplo.

```
DELIM [ \t]
ENTER [\n]
ESPACO {DELIM}+
SEPARADOR [;{}(),\[\]]
LETRA [A-Za-z]
DIGITO [0-9]
INT "-"?{DIGITO}+
FLOAT "-"?{DIGITO}+("."{DIGITO}+)*
ID {LETRA}({LETRA}|{DIGITO})*
VAR_DECL {ID}['['({DIGITO})+']']

/* keywords */
BOOL "true"|"false"
TIPO "float"|"int"|"char"|"bool"|"void"|"tuple"
CONDICOES "if"|"else"
LACOS "while"|"for"
RETORNO "return"
PARAM_TIPOS "void"|{TIPO}" "{ID}['']']" "(','{TIPO}" "{ID}['']'])*

/* tuple sendo declarada a partir da palavra reservada */
DECLARACAO {TIPO} {VAR_DECL}(','{VAR_DECL})*|"tuple" {TIPO}(','{TIPO})* VAR_DECL

FUNC_DECL {TIPO}" "{ID}'('" "{PARAM_TIPOS}" "')'
/* COD_BLOCK IF'('EXPRESSION')'COD_BLOCK[ else COD_BLOCK] */
OP_ARITM "+"|"-"|"*"|"/"
OP_COMP "=="|"!="|"<="|"<"|">="|">"
OP_LOG "&&"|"||"
OP_ASSIGN "="
SCAN "scan"'('{DIGITO}','{VAR_DECL}')'|"scan"'('{LETRA}','{VAR_DECL}')'
PRINT "print"'('{VAR_DECL}')'|"print"'('({LETRA}{DIGITO})*')'
```

Figura 2: Código das definições da gramática no arquivo lex

A partir desse código em Lex, com a execução do programa flex um analisador léxico em C é gerado, capaz de identificar os lexemas da linguagem C* e converter nos tokens, como mostrado na figura a seguir como exemplo de leitura e funcionamento do programa de análise léxica.

Ele identifica o texto escrito na linguagem C* e separa essas entradas em tokens, podendo ser tipos (declarações de tipos, float, int), ID (caso de nome de funções e variáveis, operadores (por exemplo o =, operador de atribuição).

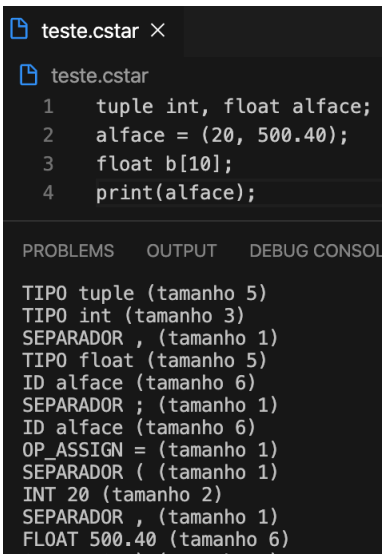
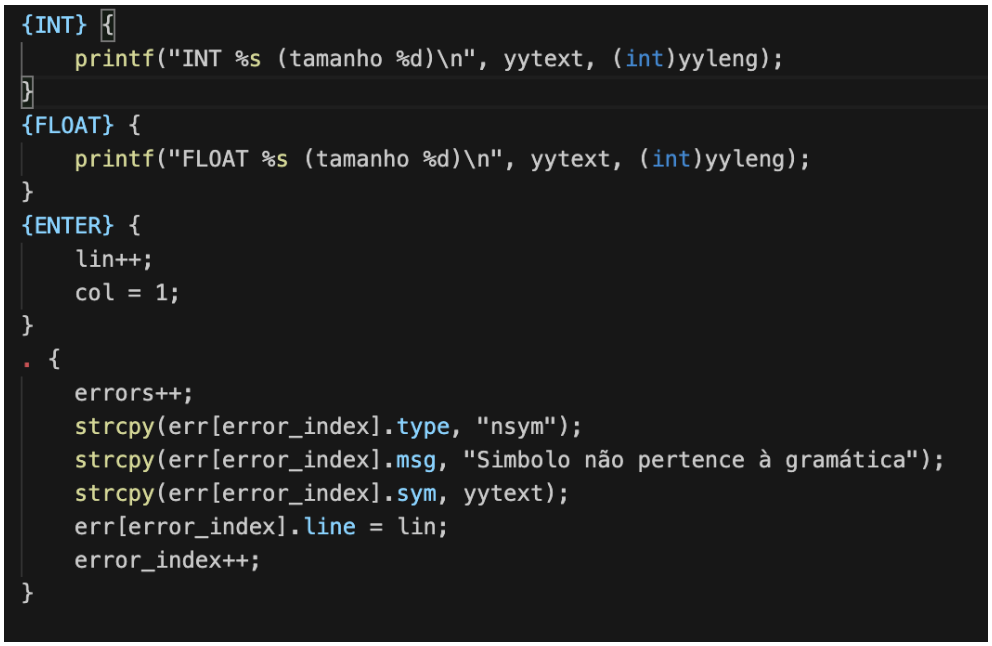


Figura 3: Exemplo da geração de tokens de um arquivo cstar (C*)

5.2 Tratamento de Erros

Para o tratamento de erros, em um primeiro momento, será implementado um identificador de erros léxicos (padrões que não pertencem à gramática de linguagem, como variáveis que começam com números, símbolos como @, , etc.)

Com a linguagem lex isso fica bem fácil de ser implementado, dado que se nos padrões de formações dos tokens o lexema não se encaixar em nenhum, ele será um erro léxico, que pode ser traduzido pela regra ”.”do arquivo lex:



```
{INT} {
    printf("INT %s (tamanho %d)\n", yytext, (int)yyleng);
}
{FLOAT} {
    printf("FLOAT %s (tamanho %d)\n", yytext, (int)yyleng);
}
{ENTER} {
    lin++;
    col = 1;
}
. {
    errors++;
    strcpy(err[error_index].type, "nsym");
    strcpy(err[error_index].msg, "Simbolo não pertence à gramática");
    strcpy(err[error_index].sym, yytext);
    err[error_index].line = lin;
    error_index++;
}
```

Figura 4: Regra .

Essa regra basicamente, de expressões regulares [5], significa qualquer lexema. Como ela se encontra no final do arquivo, terá a menor prioridade possível, logo tudo que não se encaixar em uma regra da linguagem cairá nessa regra, onde serão feitos os tratamentos de erro.

Na figura 4 vemos que, ao cair nessa regra ”.”, aumentamos o contador de erros e o índice do erro, atualizando também a linha em que ele ocorre (que é identificada pois a cada quebra de linha na linguagem aumentamos o índice da linha, visto na regra ENTER). Uma estrutura denominada lerror (lexical error) grava o tipo do erro (nesse caso nsym, ou no symbol, representando que não há esse lexema na linguagem), a mensagem de erro a ser exibida e qual o caracter, salvo na variável yytext, nativa do programa flex.

Um vetor de erros (lerror) é gerado, e a cada erro identificado adiciona-se ele ao vetor de erros, exibido no final da análise léxica do arquivo da linguagem.

5.3 Funções Introduzidas

As funções introduzidas no código do analisador léxico gerado são mostradas na figura 5. Basicamente temos a estrutura de erro e funções para o tratamento e para mostrar tais erros, além de importações de algumas bibliotecas do C.

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int errors = 0;
int error_index = 0;
int col = 1; int lin = 1;

struct lerror {
    char type[50];
    char msg[100];
    char sym[100];
    int line;
};

struct lerror err[500];

void printErrors(){
    int i;
    for(i=0; i<errors; i++){
        printf("\nErro número %d\n", i+1);
        printf("Simbolo: %s\n", err[i].sym);
        printf("%s\n", err[i].type);
        printf("%s\n", err[i].msg);
        printf("Linha: %d\n", err[i].line);
        printf("\n");
    }
}
```

Figura 5: Códigos introduzidos no analisador léxico

A variável errors denota a quantidade de erros encontrada na análise léxica, identificados como descrito na seção anterior. A variável error_index denota o índice do erro encontrado, utilizado para facilitar a inserção dos erros no vetor de erros e a identificação de cada um. As variáveis col e lin são variáveis que irão ser iteradas durante a análise léxica, para caso um erro seja encontrado seja possível retornar a linha e coluna onde esse erro foi encontrado, para facilitar o tratamento dos erros.

A struct lerror é a estrutura de dados utilizada para salvar os erros, contendo o tipo do erro (type), a mensagem de erro à ser exibida (msg), o símbolo encontrado no erro (sym) e a linha onde o erro foi encontrado (line). A função printErrros é chamada no final da análise léxica, onde o vetor denominado err (vetor de lerrors) é iterado e são mostrados todos os erros encontrados.

5.4 Tabela de Símbolos

A tabela de símbolos tem sua implementação dada por um vetor de estruturas de dados do tipo symbol_item, que possuem o símbolo a ser colocado na tabela (por exemplo o nome da variável

se for um ID), a posição na tabela (referência do endereço) e o escopo em que se encontra (global, de função).

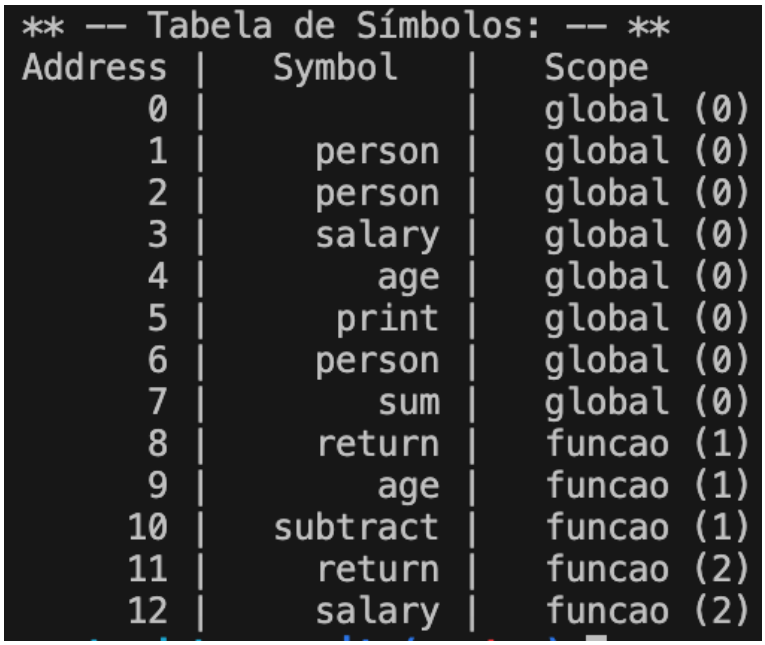


Figura 6: Exemplo de saída da tabela de símbolos sendo mostrada

5.5 Descrição dos Arquivos de Teste

Alguns arquivos de teste foram gerados, denominados teste.cstar, teste_error.cstar e teste2_error.cstar. Os arquivos com error no nome contém alguns erros léxicos identificados na execução do programa:
No arquivo teste_error.cstar temos:

```
int 5a;
```

Um erro de padrão, onde uma variável está sendo declarada com número no começo.
No arquivo teste2_error.cstar temos:

```
int ab@cax# = 0;


int returnAbacaxi() {
  return abacaxi;
 }
```

Onde temos um erro de símbolos que não pertencem à gramática.
No arquivo teste.cstar temos um programa que funciona normalmente, tendo os padrões encontrados na gramática:

```
tuple int, float alface;
alface = (20, 500.40);
float b[10];
print(alface);
```

Onde quando executado gera os tokens mostrados na figura 3.

Referências

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques and Tools*. Vol. 7. Addison Wesley, 1986.

[2] P. Baudin et al. *ACSL: ANSI C Specification Language*. Vol. 2. Domaine de Voluceau, 78150 Rocquencourt, France: Institut National de Recherche en Informatique et en Automatique, 2008.

[3] Arizona Edu. ≪C– Language Specification≫. Em: (abr. de 2005). URL: `http://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/cminusminusspec.html`.

[4] Shivani Mittal. *Fast Lexical Analyser Generator*. URL: `https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/`. (acesso em: 05.09.2020).

[5] Ray Toal. *Regular Expressions*. URL: `https://cs.lmu.edu/~ray/notes/regex/`. (acesso em: 05.09.2020).

[6] Wikipédia. *Énuplo*. URL: `https://pt.wikipedia.org/wiki/%C3%89nuplo`. (acesso em: 03.03.2020).

[7] Wikipédia. *Lex (software)*. URL: `https://en.wikipedia.org/wiki/Lex_(software)#:~:text=Lex%5C%20is%5C%20commonly%5C%20used%5C%20with,part%5C%20of%5C%20the%5C%20POSIX%5C%20standard.`. (acesso em: 05.09.2020).