Trabalho de Tradutores

Frederico Dib - 15/0125925

Profª. Cláudia Nalon

1/2020

1 Objetivos

A disciplina de tradutores ministrada pela professora Cláudia Nalon tem como seu principal objetivo o estudo de como funciona um tradutor de uma linguagem de programação. Este trabalho[1] consiste na implementação de um tradutor que deve traduzir uma liguagem de programação simplificada para um "Código de Três Endereços"[2]. Essa implementação acontecerá em 6 etapas, sendo elas: "Escolha do tema do projeto", "Implementação do analisador léxico", "Implementação do analisador sintático", "Implementação do analisador semântico", "Implementação do gerador de código intermediário".

2 Motivação e Proposta

Atualmente, com avanço das linguagens de programação de alto nível é destacado uma forte facilidade em manipulação de strings. Em Javascript[3], por exemplo, é possível declarar strings de tamanho dinâmico e operar sobre elas operações de concatenação por meio do símbolo '+'.

Essa facilidade na manipulação de strings é muito útil no desenvolvimento de aplicativos móveis, pois é uma forma eficaz de juntar dados vindo do backend por meio de um arquivo .json e strings pré prontas que serão mostradas para o usuário.

Em linguagens de mais baixo nível como C e assembly esse tipo de operação não é possível de forma nativa, para isso o desenvolvedor necessitaria de criar funções ou utilizar bibliotecas que implementam esse funcionamento. A proposta desse trabalho seria criar um tradutor de uma versão simplificada da linguagem C com as funcionalidades mencionadas. Por meio dessa linguagem, seria possível criar códigos como:

```
1 float test(int x, float y) {
2   float value;
3   value = x + y * x + 5.5;
```

[1] https://aprender.ead.unb.br/mod/page/view.php?id=5350
[2] https://github.com/lhsantos/tac/blob/master/doc/tac.pdf
[3] https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/First$_s$teps/*Strings*

```
4   return (value);
5 }
6
7 int main() {
8   /* Isso    um comentario */
9
10  string string1 = "Nice to meet you";
11  string string2;
12  string result;
13  float functionReturn;
14
15  string2 = "do you like blueberries?";
16  result = string1 + ", " + string2;
17  print(result);
18
19  functionReturn = test(4,5);
20  print(functionReturn);
21
22  if (functionReturn >= 20) {
23    print("maior ou igual");
24  } else {
25    print("menor");
26  }
27
28  while (functionReturn < 1000) {
29    functionReturn = functionReturn + 1;
30  }
31
32  print(functionReturn);
33
34  int scanNumero;
35  scan(scanNumero, int);
36
37  return(0);
38 }
```

3 Funcionamento do analisador léxico

Análise léxica é a primeira fase de um compilador, na qual um analisador léxico lê o fluxo de caracteres do código fonte e os agrupa em sequências de lexemas [1]. Um lexema é uma unidade básica de significado para uma linguagem. Dessa forma, o analisador léxico utiliza uma gramática para identificar lexemas da linguagem. Caso identifique algo que não faça parte da linguagem, o analisador deve informar a localização da ocorrência.

Nessa implementação primeiro é realizado um parser no código feito pelo programa Flex[4], separando os elementos do código em tokens, sendo eles:

1. ID -> Variáveis e declarações de funções.
2. NUM -> Token numérico.

[4]https://westes.github.io/flex/manual/

3. OP -> Token referente a operações aritméticas.
4. COND -> Token referente a operações condicionais.
5. STATEMENT -> if, while, else, return, print, scan.
6. TYPE -> int, float, string.
7. SEP -> Vírgula, parenteses, chaves...

À medida que o parser for sendo realizado o programa imprimirá no terminal os tokens encontrados, seus tipos e valores respectivamente.

3.1 Funções adicionadas

Além do flex, o analisador léxico criado conta com as seguintes funções e estrutura de dados para funcionar:

1. Um array de structs de erros chamado "ErrorStruct"para armazenar todos os erros encontrados pelo analisador léxico.
2. Uma variável chamada "line"que armazena a linha atual e uma chamada "word-position"que armazena a posição da palavra dentro dessa linha.
3. Uma variável chamada "errorsnum"que armazena o número de erros encontrados.
4. Uma função "printError"que imprime no terminal os erros encontrados.

3.2 Tratamento de Erros

Ao identificar lexemas que não pertencem à linguagem, o analisador implementado grava o erro no vetor de struct de erros e imprime no terminal o lexama incorreto, junto da linha e da posição da palavra. O analisador reconhece os seguintes erros:

1. IDs iniciados com números.
2. caracteres não permitidos.
3. strings não fechadas.

4 Gramática

A gramática implementada no analisador léxico sofreu algumas mudanças em relação à apresentada na Escolha do Tema. A mudança foi a adição do tipo INT (inteiro) e DEC (decimal), que são tipos de números restritos. O tipo NUM representa qualquer valor numérico, seja inteiro ou decimal.

1. $initializer \rightarrow global\text{-}list$

2. $\textit{global-list} \rightarrow \textit{global-list}\ \textit{var} \mid \textit{var} \mid \textit{global-list}\ \textit{func} \mid \textit{func} \mid \textit{comment} \mid \textit{global-list}\ \textit{comment}$

3. $\textit{comment} \rightarrow /[*].*[*]/$

4. $\textit{var} \rightarrow \textit{string}\ \textbf{NAME};\ \mid \textit{string}\ \textbf{NAME}\ =\ \textbf{STRING};\ \mid \textit{int}\ \textbf{NAME};\ \mid$
   $\textit{int}\ \textbf{NAME}\ =\ \textbf{INT};\ \mid \textit{float}\ \textbf{NAME};\ \mid$
   $\textit{float}\ \textbf{NAME}\ =\ \textbf{DEC};$

5. $\textit{func} \rightarrow \textit{string}\ \textbf{NAME}\ (\ \textit{params-list}\ )\ \{\ \textit{content-list}\ \}\ \mid$
   $\textit{int}\ \textbf{NAME}\ (\ \textit{params-list}\ )\ \{\ \textit{content-list}\ \}\ \mid$
   $\textit{float}\ \textbf{NAME}\ (\ \textit{params-list}\ )\ \{\ \textit{content-list}\ \}$

6. $\textit{params-list} \rightarrow \textit{params} \mid \varepsilon$

7. $\textit{params} \rightarrow \textit{params},\ \textit{param} \mid \textit{param}$

8. $\textit{param} \rightarrow \textit{int}\ \textbf{NAME} \mid \textit{float}\ \textbf{NAME} \mid \textit{string}\ \textbf{NAME}$

9. $\textit{content-list} \rightarrow \textit{content-list}\ \textit{content} \mid \varepsilon$

10. $\textit{content} \rightarrow \textit{var} \mid \textit{add-value} \mid \textit{comand} \mid \textit{print} \mid \textit{scan} \mid \textit{return} \mid \textit{call-func}$

11. $\textit{add-value} \rightarrow \textbf{NAME}\ =\ \textit{expression}$

12. $\textit{expression} \rightarrow \textbf{NAME} \mid \textbf{STRING} \mid \textbf{INT} \mid \textbf{DEC} \mid$
    $\textit{expression}\ \textbf{OP}\ \textit{expression}$

13. $\textit{comand} \rightarrow \textit{comand-if} \mid \textit{comand-ifelse} \mid \textit{comand-while}$

14. $\textit{comand-if} \rightarrow \textit{if}\ (\ \textit{condition}\ )\ \{\ \textit{content-list}\ \}$

15. $\textit{comand-ifelse} \rightarrow \textit{if}\ (\ \textit{condition}\ )\ \{\ \textit{content-list}\ \}\ \textit{else}\ \ \{\ \textit{content-list}\ \}$

16. $\textit{comand-while} \rightarrow \textit{while}\ (\ \textit{condition}\ )\ \{\ \textit{content-list}\ \}$

17. $\textit{condition} \rightarrow \textbf{NAME COND NAME} \mid \textbf{NAME COND NUM} \mid$
    $\textbf{NUM COND NAME} \mid \textbf{NUM COND NUM}$

18. $\textit{print} \rightarrow \textit{print}(\textbf{NAME}) \mid \textit{print}(\textbf{NUM}) \mid \textit{print}(\textbf{STRING})$

19. $\textit{scan} \rightarrow \textit{scan}(\textbf{NAME}, \textbf{NUM}) \mid \textit{scan}(\textbf{NAME}, \textbf{STRING})$

20. $\textit{return} \rightarrow \textit{return}(\textbf{NAME}) \mid \textit{return}(\textbf{NUM}) \mid \textit{return}(\textbf{STRING})$

21. $\textit{call-func} \rightarrow \textbf{NAME}(\textit{call-func-params}) \mid$
    $\textbf{NAME}\ =\ \textbf{NAME}(\textit{call-func-params})$

22. $\textit{call-func-params} \rightarrow \textit{call-func-params},\ \textbf{NAME} \mid \textbf{NAME} \mid \varepsilon$

    $\textbf{NAME} = \textit{letter}\ (\textit{letter}|\textit{digit})^*$

INT = $digit^+$

DEC = $digit^+.digit^+$

NUM = $digit^+$ | $digit^+.digit^+$

STRING = "$(all\ characters)^*$"

OP = + | **-** | ***** | /

COND = > | >= | < | <= | == | **!=**

$letter = a \mid ... \mid z \mid A \mid ... \mid Z$

$digit = 0 \mid ... \mid 9$

Palavras reservadas: **print scan int float string if else while return**
Símbolos: = |, |; |(|) |{ |} |" |+ |- |* |/ |< |<= |> |>= |== |!= |

5 Sintático

Para realizar a análise sintática da linguagem descrita, utilizei a biblioteca Bison[5] da linguagem C, junto da biblioteca Flex, previamente usada na construção do analisador léxico. Após a passagem do Flex, é enviado ao Bison uma lista de tokens, em que, por meio delas, é realizado um parser sintático da linguagem. A medida que esse parser for sendo realizado, será montado uma árvore sintática que será utilizada nos passos posteriores.

5.1 Tratamento de Erros

Para lidar com o tratamento de erros no analisador sintático, foi utilizado, por questões de eficiência e qualidade, a função "yyerror", nativa do próprio Bison. Dentro dessa função, o erro é impresso no console e o programma é encerrado por meio de um "exit(1)". Para deixar os erros mais descritivos, foi adicionado no topo do arquivo os seguintes enunciados: "%error-verbose", "%debug", "%locations".

6 Árvore Sintática

Durante a execução da analise sintática a árvore sintátiva vai sendo construída. possui a seguinte estrutura:

```
1 struct node {
2   char *value;
3   char node_type;
4   int type;
5   struct node *node1;
```

[5] https://www.gnu.org/software/bison/manual/

```
6     struct node *node2;
7     struct node *node3;
8 };
```

O campo "value"se refere ao valor daquele nó, o que poderia ser o nome de uma variável, nome de uma função, um número, entre outros. O campo "type"se refere à tipagem de dado daquele nó caso tenha, sendo "0"um valor inteiro, "1"um valor decimal, "2"uma string, "3"tipo não definido. Posteriormente essa tipagem será usada para análise semântica. O campo "node_type"Se refere ao tipo de nó, especificando que tipo de nó está sendo definido, entre eles temos os seguinte valores:

1. F - Declaração de função;
2. v - Declaração de variável;
3. V - Variável;
4. P - Parâmetros de função;
5. R - Regra de ligação;
6. C - IF ou IF-ELSE;
7. c - Expressões condicionais;
8. W - While;
9. I - Inteiro;
10. D - Decimal;
11. S - String;
12. B - Simbolo;
13. E - Expressão aritmética;
14. A - Atribuição de valor;
15. L - Chamada de função;

Após a criação da árvore será impresso no terminal uma representação da mesma. Essa é a saída referente ao código de exemplo dado na sessão "Motivação e Proposta":

```
########## Arvore Sintatica ##########
<Função> test, tipo: 1
INICIO PARAMETROS
-<Parametro> x, tipo: 0
-<Parametro> y, tipo: 1
FIM PARAMETROS
```

```
INICIO FUNÇÃO -  test
-<Declaração de Variavel> value, tipo: 1
-<Atribuição> (tipo: 1) value =
--<EXPRESSION>
--<Variavel> x, tipo: 0
---<Simbolo> +
---<EXPRESSION>
---<Variavel> y, tipo: 1
----<Simbolo> *
----<EXPRESSION>
----<Variavel> x, tipo: 0
-----<Simbolo> +
-----<Decimal> 5.5
-<RETURN> tipo: 1
--<Variavel> value, tipo: 1
FIM FUNÇÃO - test

<Função> main, tipo: 0
INICIO PARAMETROS
FIM PARAMETROS
INICIO FUNÇÃO -  main
-<Declaração de Variavel> string1, tipo: 2
--<Atribuição> (tipo: 2) string1 =
---<String> "Nice to meet you"
-<Declaração de Variavel> string2, tipo: 2
-<Declaração de Variavel> result, tipo: 2
-<Declaração de Variavel> functionReturn, tipo: 1
-<Atribuição> (tipo: 2) string2 =
--<String> "do you like blueberries?"
-<Atribuição> (tipo: 2) result =
--<EXPRESSION>
--<Variavel> string1, tipo: 2
---<Simbolo> +
---<EXPRESSION>
---<String> ", "
----<Simbolo> +
----<Variavel> string2, tipo: 2
-<PRINT>
--<Variavel> result, tipo: 2
-<Atribuição> (tipo: 3) functionReturn =
--<CHAMA FUNÇÃO> test--INICIO PARAMETROS
---<Inteiro> 4
---<Inteiro> 5
--FIM PARAMETROS
```

```
-<PRINT>
--<Variavel> functionReturn, tipo: 1
-<IF>
-IF - INICIO CONDIÇÃO
--<CONDIÇÃO>
--<Variavel> functionReturn, tipo: 1
--<Simbolo> CGE
--<Inteiro> 20
-IF - FIM CONDIÇÃO
-IF - INICIO BLOCO
--<PRINT>
---<String> "maior ou igual"
-IF - FIM BLOCO
ELSE - INICIO BLOCO
--<PRINT>
---<String> "menor"
-ELSE - FIM BLOCO
-<WHILE>
-WHILE - INICIO CONDIÇÃO
--<CONDIÇÃO>
--<Variavel> functionReturn, tipo: 1
--<Simbolo> CLT
--<Inteiro> 1000
-WHILE - FIM CONDIÇÃO
-WHILE - INICIO BLOCO
--<Atribuição> (tipo: 1) functionReturn =
---<EXPRESSION>
---<Variavel> functionReturn, tipo: 1
----<Simbolo> +
----<Inteiro> 1
-WHILE - FIM BLOCO
-<PRINT>
--<Variavel> functionReturn, tipo: 1
-<Declaração de Variavel> scanNumero, tipo: 0
-<SCAN> tipo: 0
--<Variavel> scanNumero, tipo: 0
-<RETURN> tipo: 0
--<Inteiro> 0
FIM FUNÇÃO - main
```

7 Tabela de Símbolos

Durante a execução da analise sintática a tabela de símbolos vai sendo construída. possui a seguinte estrutura:

```
struct scope {
  char *scope_name;
  int type; // 0 = int, 1 = float, 2 = string
  struct scope *next;
};

struct symbol_node {
  char *key; // hash key -> scope_name + @ + id
  char *id; // vari vel
  char *scope_name; // nome do scopo
  int type; // 0 = int, 1 = float, 2 = string
  UT_hash_handle hh;
};
```

A struct "scop"é usada para a implementação da pilha de escopos sinalizando em qual escopo está sendo considerado no momento. O compo "scop_name"identifica qual é aquele escopo, sendo geralmente o nome da função. O campo type se refere ao tipo da função, que será usado no analisador semântico no proximo passo.

A tabela de símbolos tem a função de identificar os IDs declarados, levando em conta o escopo que aquele ID foi inicializado e o tipo do mesmo. Para implementar essa tabela foi utilizado a estrutura "Hash Table"junto da struct "symbol_node"para recuperar com mais facilidade e eficiência cada um desses símbolos. O campo "id"se refere ao nome da variável. O campo "scope_name"se refere ao escopo ao qual foi declarado aquela variável, permitindo que tenha variáveis com mesmo nome em diferentes escopos. O campo "key"é a chave para se recuperar essa variável, essa campo é construído por meio da concatenação (escopo da função) + "@"+ (nome da variável).

Após o fim da execução a tabela de símbolos é impressa no terminal. Esta é a tabela de símbolos referente ao exemplo de código mostrado na sessão "Motivação e Proposta":

```
########## Tabela de Símbolos ##########
Simbolo: test, Tipo: 1, Scopo: Global, key: @test
Simbolo: x, Tipo: 0, Scopo: test, key: test@x
Simbolo: y, Tipo: 1, Scopo: test, key: test@y
Simbolo: value, Tipo: 1, Scopo: test, key: test@value
Simbolo: main, Tipo: 0, Scopo: Global, key: @main
Simbolo: string1, Tipo: 2, Scopo: main, key: main@string1
Simbolo: string2, Tipo: 2, Scopo: main, key: main@string2
Simbolo: result, Tipo: 2, Scopo: main, key: main@result
Simbolo: functionReturn, Tipo: 1, Scopo: main, key: main@functionReturn
Simbolo: scanNumero, Tipo: 0, Scopo: main, key: main@scanNumero
```

8 Semântica

Descrição breve da linguagem. O tipo de variável "int"se remete a valores do tipo inteiro, o tipo de variável "float"se remete a valores do tipo real, o tipo de variável "string"se remete a um vetor de caracteres. As expressões aritméticas(previsto na regra 11) permite realizar operações com nomes, números e strings, a semântica da linguagem deve garantir que strings e nomes do tipo string façam operações somente entre si da mesma forma que números apenas opere com outros números e nomes do tipo int e float.

No caso de uma operação int com float o inteiro deve ser convertido para float. No caso de operação entre duas strings, o único operador permitido será o operador "+", os outros operadores não serão permitidos nesse contexto.

9 Arquivos de teste

O analisador possui 6 arquivos de teste: "input_correto1.txt", "input_correto2.txt", "input_erro_lexico1.txt", "input_erro_lexico2.txt", "input_erro_sintatico1.txt", "input_erro_sintatico2.txt".

Para compilar e executar os arquivos, basta entrar no diretório do analisador léxico e digitar os seguintes comandos:

1. lex lex.l
2. yacc -d lang.y
3. gcc -g lex.yy.c y.tab.c -ll -o lang
4. ./lang input_correto1.txt
5. ./lang input_correto2.txt
6. ./lang input_erro_lexico1.txt
7. ./lang input_erro_lexico2.txt
8. ./lang input_erro_sintatico1.txt
9. ./lang input_erro_sintatico2.txt

No Arquivo "input_erro_lexico1.txt", na linha 3 foi introduzido uma variável que se inicia com número. No Arquivo "input_erro_lexico2.txt", na linha 10 foi introduzido um caractere inválido (@). No Arquivo "input_erro_sintatico1.txt", na linha 11 faltou um ";". No Arquivo "input_erro_sintatico1.txt", na linha 22 o parentese do if não foi fechado.

Referências

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*, volume 7. 2nd edition edition, 1986.