UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL

Pablo Ferreira - 3480
Samuel Sena - 3494
Thomas Chang - 3052

TRABALHO PRÁTICO 3
COMPILADORES - CCF 441

Florestal
2021

Trabalho Prático 02

Introdução

Nesta etapa do trabalho prático, ficamos encarregados de definir os conceitos e funcionamento de uma linguagem de programação. Para isso, realizaremos a definição dos tipos primitivos de dados aceitos pela nossa linguagem, além de comandos suportados, o paradigma de programação, palavras-chave e palavras reservadas. A gramática da linguagem, juntamente com tokens e padrões de lexemas são exemplos de elementos que também serão especificados.

Nome da Linguagem

O grupo escolheu o nome MAYBE para a linguagem, porque talvez funcione, talvez seja eficiente, talvez a gente consiga entregar tudo e talvez a gente passe na disciplina. A extensão dos arquivos escritos em MAYBE deverá ser “.may”.

Tipos de dados:

O grupo escolheu os seguintes tipos de dados:

Tipo de dados	**Palavra reservada**
Inteiro	int
Ponto flutuante	float
Caractere	char
String	string
Boolean	boolean

Comandos suportados:

Comando	**Ação**
=	Comando de atribuição
+	Comando aritmético de adição
-	Comando aritmético de subtração
*	Comando aritmético de multiplicação
/	Comando aritmético de divisão

mod	Comando aritmético de resto de divisão
==	Comando relacional de igualdade
>=	Comando relacional de superioridade ou igualdade
<=	Comando relacional de inferioridade ou igualdade
>	Comando relacional de superioridade
<	Comando relacional de inferioridade
<>	Comando relacional de diferença
not	Comando lógico de negação
or	Comando lógico de união
and	Comando lógico de intersecção

Paradigma de programação:

O paradigma de programação escolhido por nós para nossa linguagem de programação é o paradigma imperativo e estruturado. Dessa forma, teremos uma linguagem bem objetiva e com legibilidade alta. Nossa linguagem de programação também contará com o recurso de comentários (que serão desconsiderados durante o processo de compilação).

Palavras-chave e palavras reservadas:

As palavras-chave e palavras reservadas serão principalmente aquelas palavras utilizadas para a invocação de comandos durante a implementação do fluxo do código da linguagem. São elas:

- int
- float
- char
- string
- boolean
- void
- return
- mod
- not
- or
- and

- if
- elsif
- else
- while
- true
- false
- break
- continue

Lexemas e tokens

A tabela abaixo relaciona e explicita todos os tokens e lexemas a serem identificados e retornados pelo analisador léxico.

Token	**Lexema**	**Padrão**
INT	int	int
FLOAT	float	float
CHAR	char	char
STRING	string	string
VOID	void	void
RETURN	return	return
BREAK	break	break
CONTINUE	continue	continue
LE	<=	<=
GE	>=	>=
EQ	==	==
NE	<>	<>
GRT	>	>
LESS	<	<
NOT	not	not

OR	or	or
AND	and	and
IF	if	if
ELSIF	elsif	elsif
ELSE	else	else
WHILE	while	while
NUM_I	-?[0-9]+	(números inteiros)
NUM_F	-?[0-9]+.[0-9]+	(números reais)
ID	[a-zA-Z][a-zA-Z0-9]*	sequência de caracteres seguidos ou não por números
MOD	mod	mod
ADD	+	+
SUB	-	-
DIV	/	/
MUL	*	*
ABREPARENTESES	(	(
FECHAPARENTESES	)	)
ABRECOLCHETES	[	[
FECHACOLCHETES	]	]
ABRECHAVES	{	{
FECHACHAVES	}	}
PVIRGULA	;	;
VIRGULA	,	,

Gramática da linguagem

Decidimos que nossa linguagem de programação irá desprezar espaços em branco e tabulações, dessa forma, as expressões deverão ser delimitadas com o uso do caractere terminal “;” (ponto e vírgula). Além disso, os escopos serão delimitados com os caracteres “{“ e “}”. Lembrando que o balanceamento deles será necessário para a correta sintaxe.

Para criar um comentario na linguagem, o mesmo deverá ser precedido dos caracteres “/*” e ao sucedido de “*/”. Dessa maneira, o compilador entenderá todo conteúdo escrito entre tais delimitadores deverá ser desconsiderado para o processo de compilação.

A forma de declaração de uma variável seguirá a seguinte máscara:

Tipo *Nome_da_variável;*

Sendo que, inicialmente o tipo da variável deverá ser definido, e em seguida, o nome do identificador.

A forma de declarar um subprograma ou função também segue tal formula:

```
Tipo Nome_da_funcao(Tipo1 param1,Tipo2 param2...){
        /* Corpo da função com retorno */
}
```

Sendo que, inicialmente o tipo do retorno é definido, em seguida é escrito o nome ou identificador da função, a frente os parâmetros deverão ser declarados entre parênteses e separados por vírgulas e o escopo é definido pelas chaves ({,}).

O conteúdo de strings deve ser escrito entre aspas duplas e os caracteres devem ser escritos entre aspas simples (para atribuir conteúdos em variáveis do tipo string e carácter).

Com isso, temos a seguinte gramática para a linguagem MAYBE:

expr
: expr ‘**+**’ expr ;
| expr ‘**-**’ expr ;
| expr ‘*****’ expr ;
| expr ‘**/**’ expr ;
| expr ‘**mod**’ expr ;
| expr ‘**==**’ expr ;
| expr ‘**<>**’ expr ;

| expr ‘**<=**’ expr ;
| expr ‘**>=**’ expr ;
| expr ‘**>**’ expr ;
| expr ‘**<**’ expr ;
| ‘**(**‘ expr ‘**)**’ ;
| ‘**[**‘ expr ‘**]**’ ;
| ‘**{**‘ expr ‘**}**’ ;
| expr ‘**or**’ expr ;
| expr ‘**and**’ expr ;
| expr ‘**not**’ ;
| expr ‘**=**’ expr ;
| term ;

term
: **-?[digit]+ ;**
| **+?[digit]+ ;**
| **+?[digit]+.[digit]+ ;**
| **-?[digit]+.[digit]+ ;**
| **[letter]+ ;**
| **return ;**
| **continue ;**
| **break ;**
| **int ;**
| **float ;**
| **string ;**
| **char ;**
| **boolean ;**
| **void ;**
| ‘**;**’ ;
| ‘,’ ;

while
: **while** '(' expr ')' '{' stmt '}' ;

conditional
: **if** ‘(‘ expr ‘)’ ‘{‘ stmt ‘}’ ;
| **elsif** ‘(‘ expr ‘)’ ‘{‘ stmt ‘}’;
| **else** ‘{‘ stmt ‘}’ ;
stmt
: expr expr expr ;
| term stmt ;
| expr ;
digit
: **"0"** | **"1"** | **"2"** | **"3"** | **"4"** | **"5"** | **"6"** | **"7"** | **"8"** | **"9" ;**

letter

: **[a-zA-Z];**

Gerador de analisador léxico

O nosso analisador léxico foi gerado utilizando o gerador de analisador léxico Flex. Para compilá-lo basta executar “flex lex.l” e em seguida “gcc lex.yy.c”. O binário final “a.out” consiste em nosso analisador léxico. Para executar a análise léxica de um arquivo de código fonte, basta executar o comando na seguinte forma: “./a.out < *nome_arquivo_de_entrada”.*

Realizamos a implementação de cinco arquivos de entrada com o intuito de testar se todos os padrões aqui definidos estão sendo de fato reconhecidos e tendo seus tokens devidamente retornados na tela. Os arquivos de entrada estão nomeados de 1 a 5 e estão na mesma pasta que o arquivo lex.l. As figuras abaixo exibem saídas parciais obtidas para a execução dos arquivos:

Figura 1 - Análise léxica do arquivo entrada4.may.

```
Token: INT -> Lexema: "int"
Token: ID -> Lexema: "func"
Token: ABREPARENTESES -> Lexema: "("
Token: INT -> Lexema: "int"
Token: ID -> Lexema: "a"
Token: VIRGULA -> Lexema: ","
Token: BOOLEAN -> Lexema: "boolean"
Token: ID -> Lexema: "d"
Token: FECHAPARENTESES -> Lexema: ")"
Token: ABRECHAVES -> Lexema: "{"
Token: INT -> Lexema: "int"
Token: ID -> Lexema: "b"
Token: ATRIBUI -> Lexema: "="
Token: NUM_I -> Lexema: "5"
Token: PVIRGULA -> Lexema: ";"
Token: INT -> Lexema: "int"
Token: ID -> Lexema: "c"
Token: ATRIBUI -> Lexema: "="
Token: ID -> Lexema: "b"
Token: ID -> Lexema: "mod"
Token: ID -> Lexema: "a"
Token: PVIRGULA -> Lexema: ";"
Token: IF -> Lexema: "if"
Token: ABREPARENTESES -> Lexema: "("
Token: ID -> Lexema: "b"
Token: OP -> Lexema: ">"
Token: ID -> Lexema: "a"
Token: FECHAPARENTESES -> Lexema: ")"
Token: ABRECHAVES -> Lexema: "{"
Token: ID -> Lexema: "c"
Token: ATRIBUI -> Lexema: "="
Token: NUM_I -> Lexema: "1"
Token: PVIRGULA -> Lexema: ";"
Token: FECHACHAVES -> Lexema: "}"
Token: ELSIF -> Lexema: "elsif"
Token: ABREPARENTESES -> Lexema: "("
Token: ID -> Lexema: "a"
Token: OP -> Lexema: ">"
Token: ID -> Lexema: "b"
Token: AND -> Lexema: "and"
Token: NOT -> Lexema: "not"
Token: ID -> Lexema: "d"
Token: FECHAPARENTESES -> Lexema: ")"
Token: ABRECHAVES -> Lexema: "{"
Token: ID -> Lexema: "c"
Token: ATRIBUI -> Lexema: "="
Token: NUM_I -> Lexema: "2"
```

Figura 2 - Análise léxica do arquivo entrada5.may.

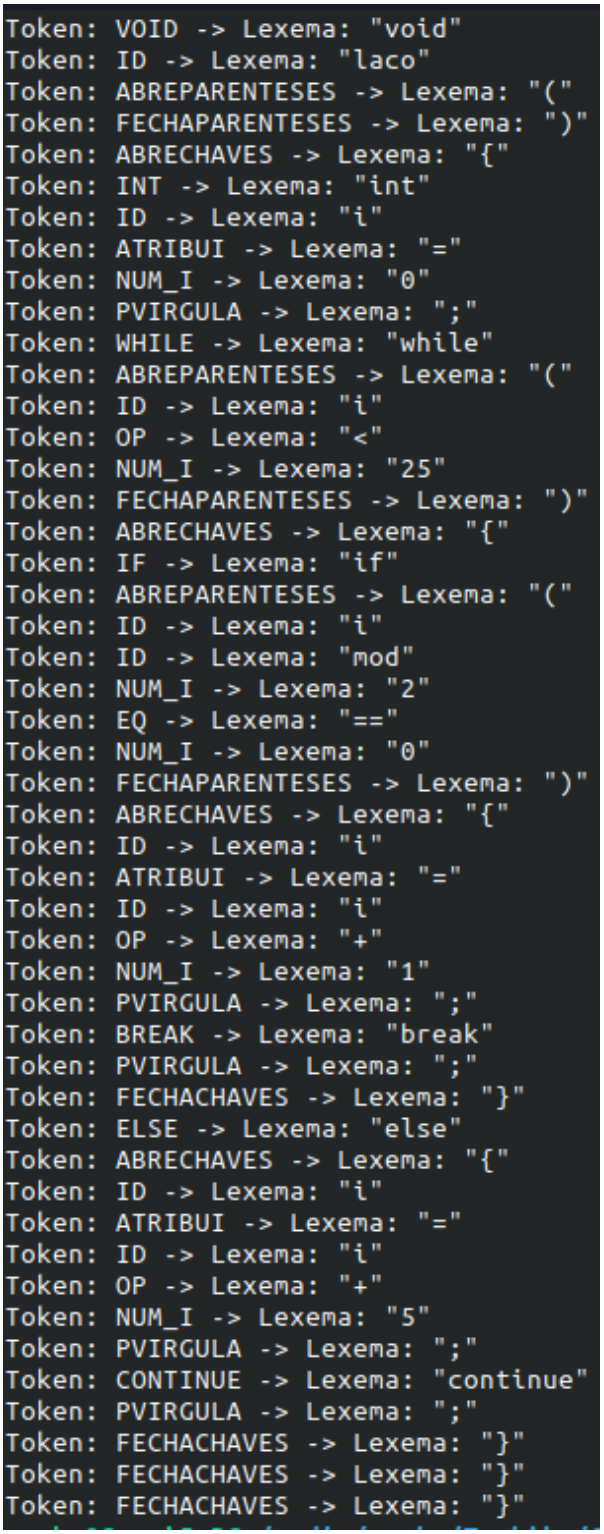


Figura 3 - Análise léxica do arquivo entrada2.may.

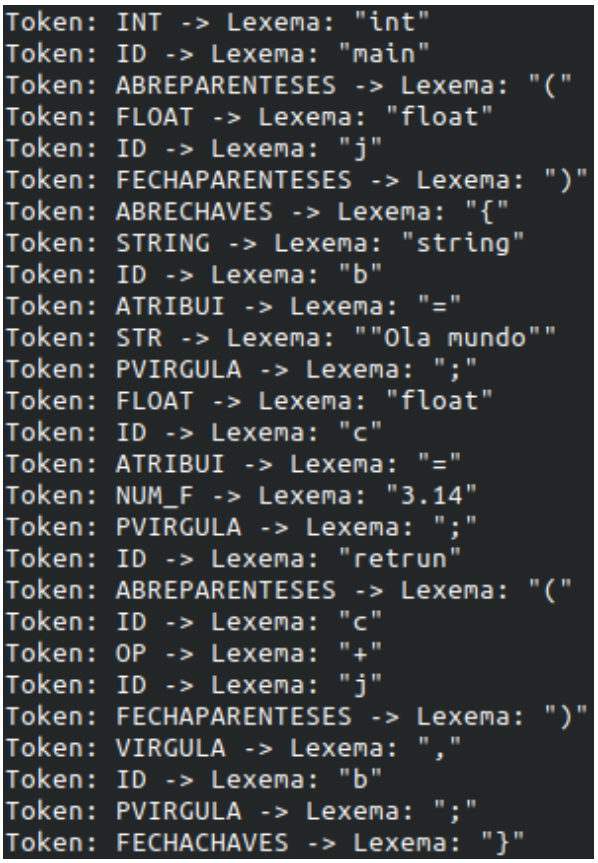


Figura 4 - Análise léxica do arquivo entrada3.may.

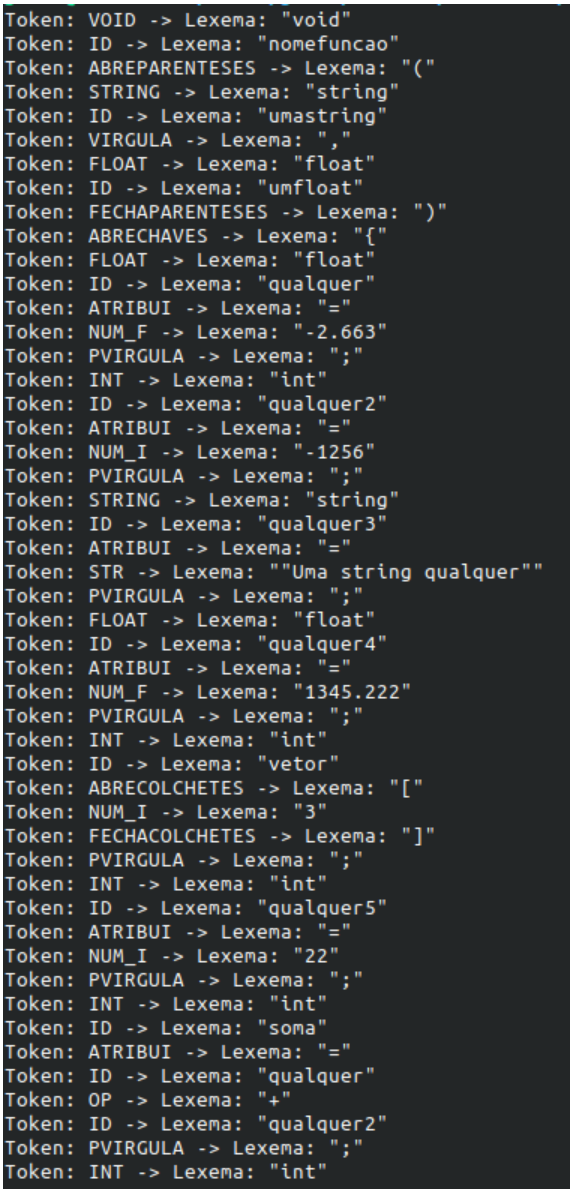
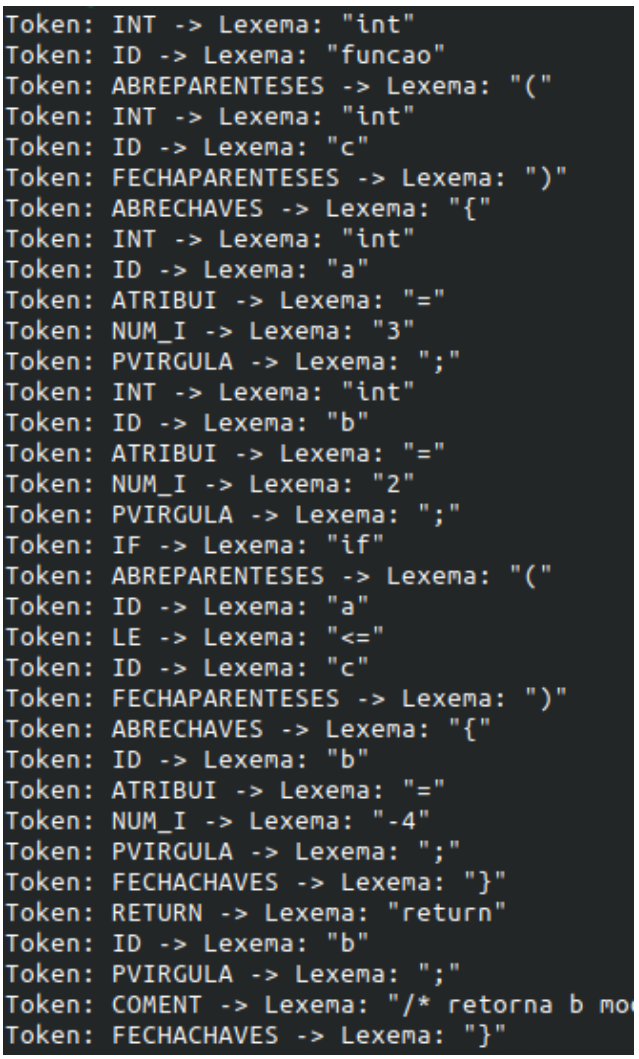


Figura 5 - Análise léxica do arquivo entrada.may.



Conclusão

A realização deste trabalho foi de grande ajuda para uma melhor compreensão do processo inicial de criação e definição de um compilador de uma linguagem de programação. Esperamos aprender ainda mais sobre o assunto para que possamos, nas etapas seguintes, realizar as modificações necessárias para um funcionamento mais correto e assim implementar de maneira melhor nossa linguagem de programação.

Todas as dúvidas que tivemos durante o processo de desenvolvimento desta parte do trabalho foram esclarecidas com diversas pesquisas pela Web, e as principais fontes de tais esclarecimentos foram devidamente referenciadas.

Trabalho Prático 03

Modificações:

Nessa primeira etapa, ficamos primeiro encarregados de consertar erros que tivemos em relação a gramática da linguagem. E essas mudanças foram fundamentais para o analisador sintático funcionar.

Uma nova gramática para a linguagem MAYBE foi escrita, tal necessidade se deve ao mal funcionamento e inúmeras redundâncias causadas pela gramática anterior (além de inconsistências em sua declaração). A nova gramática implementada no Bison:

```
start:
   | lista start
lista: dec_func
   | dec_var
   | ref_var
   | ref_func PVIRGULA
   | condicional
lista_escopo: dec_var lista_escopo
   | ref_var lista_escopo
   | ref_func PVIRGULA lista_escopo
   | condicional lista_escopo
   | comando lista_escopo
   | dec_var
   | ref_var
   | ref_func PVIRGULA
   | comando
   | condicional
dec_func: tipos_ids dec_id ABREPARENTESES dec_parametro FECHAPARENTESES
ABRECHAVES lista_escopo FECHACHAVES
dec_parametro:
   | tipos_ids dec_id
   | tipos_ids dec_id VIRGULA dec_parametro
ref_parametro:
   | ref_id
   | ref_id VIRGULA ref_parametro
dec_var: tipos_ids dec_id PVIRGULA
   | tipos_ids dec_id ATRIBUI valor_ou_id PVIRGULA
   | tipos_ids dec_id ATRIBUI op PVIRGULA
   | tipos_ids dec_id ABRECOLCHETES NUM_I FECHACOLCHETES PVIRGULA
ref_var: ref_id ATRIBUI valor_ou_id PVIRGULA
```

```
   | ref_id ATRIBUI op PVIRGULA
ref_func: ref_id ABREPARENTESES ref_parametro FECHAPARENTESES
op: ops_c_parenteses
   | op_s_parenteses
ops_c_parenteses:   ABREPARENTESES   valor_ou_id   op_arit   valor_ou_id
FECHAPARENTESES
     | ABREPARENTESES valor_ou_id op_arit valor_ou_id FECHAPARENTESES
op_arit op
     | ABREPARENTESES valor_ou_id op_arit valor_ou_id FECHAPARENTESES
op_arit valor_ou_id
   | ABREPARENTESES valor_ou_id op_arit op FECHAPARENTESES
   | ABREPARENTESES valor_ou_id op_arit op FECHAPARENTESES op_arit op
     | ABREPARENTESES valor_ou_id op_arit op FECHAPARENTESES op_arit
valor_ou_id
   | ABREPARENTESES ops_c_parenteses FECHAPARENTESES
   | ABREPARENTESES ops_c_parenteses FECHAPARENTESES op_arit op
  | ABREPARENTESES ops_c_parenteses FECHAPARENTESES op_arit valor_ou_id
op_s_parenteses: valor_ou_id op_arit valor_ou_id
   | valor_ou_id op_arit op
op_arit: ADD
   | SUB
   | MUL
   | DIV
   | MOD
valor: NUM_F
   | NUM_I
   | STR
   | TRUE
   | FALSE
   | CH
valor_ou_id: valor
   | ABREPARENTESES SUB ref_id FECHAPARENTESES
   | SUB ref_id
   | ref_id
   | ABREPARENTESES SUB ref_func FECHAPARENTESES
   | SUB ref_func
   | ref_func
tipos_ids:INT
   | FLOAT
   | STRING
   | CHAR
   | BOOLEAN
   | VOID
```

```
dec_id: ID
ref_id: ID
   | ID ABRECOLCHETES NUM_I FECHACOLCHETES
comando: RETURN valor_ou_id PVIRGULA
   | RETURN op PVIRGULA
   | RETURN PVIRGULA
   | CONTINUE PVIRGULA
   | BREAK PVIRGULA
expressao_logica: valor_ou_id op_logica valor_ou_id
expressao_relacional: expressao_relacional_s_parenteses
   | expressao_relacional_c_parenteses
expressao_relacional_c_parenteses:     ABREPARENTESES  expressao_logica
FECHAPARENTESES op_relacional expressao_relacional
   | ABREPARENTESES expressao_relacional_c_parenteses FECHAPARENTESES
    | ABREPARENTESES expressao_relacional_c_parenteses FECHAPARENTESES
op_relacional expressao_relacional
   | ABREPARENTESES expressao_logica FECHAPARENTESES
    | NOT ABREPARENTESES expressao_logica FECHAPARENTESES op_relacional
expressao_relacional
          |   NOT   ABREPARENTESES   expressao_relacional_c_parenteses
FECHAPARENTESES
          |   NOT   ABREPARENTESES   expressao_relacional_c_parenteses
FECHAPARENTESES op_relacional expressao_relacional
   | NOT ABREPARENTESES expressao_logica FECHAPARENTESES
expressao_relacional_s_parenteses:    expressao_logica    op_relacional
expressao_relacional
   | expressao_logica
   | NOT expressao_logica op_relacional expressao_relacional
   | NOT expressao_logica
op_relacional: OR
   | AND
op_logica: GRT
   | LESS
   | LE
   | GE
   | EQ
   | NE
condicional:     op_condicao     ABREPARENTESES     expressao_relacional
FECHAPARENTESES ABRECHAVES lista_escopo FECHACHAVES
   | ELSE ABRECHAVES lista_escopo FECHACHAVES
op_condicao: IF
   | ELSIF
   | WHILE
```

Em seguida, modificações foram realizadas no arquivo de análise léxica “lex.l”. Removemos todos os prints referentes a segunda etapa do trabalho e adicionamos returns para retornar as respectivas tokens.

```
{int} {return INT;}
{float} {return FLOAT;}
{char} {return CHAR;}
{string} {return STRING;}
{boolean} {return BOOLEAN;}
{void} {return VOID;}
{return} {return RETURN;}
{break} {return BREAK;}
{continue} {return CONTINUE;}
{not} {return NOT;}
{or} {return OR;}
{and} {return AND;}
{if} {return IF;}
{elsif} {return ELSIF;}
{else} {return ELSE;}
{while} {return WHILE;}
{NumPos} {return NUM_I;}
{NumNeg} {return NUM_I;}
{PontoF_P} {return NUM_F;}
{PontoF_N} {return NUM_F;}
{String} {return STR;}
{Ch} {return CH;}
```

Figura 6 - Retorno de tokens no “lexl.l”

Também implementamos uma função chamada “*Valor_Semantico()*” encarregada de repassar o valor semântico (o nome) dos identificadores para o analisador sintático, sendo que ela é invocada apenas no momento em que uma token de identificador é retornada.

```
char * Valor_Semantico(char* yytext, int yyleng) //funcao criada
{
    char * id_name = (char*)malloc((yyleng+1)*sizeof(char));
    if (id_name!=NULL){
        strcpy(id_name, yytext);
    }
    return id_name;
}
```

Figura 7 - Função para repassar o valor semântico para o analisador sintático.

Após essas correções, começamos a realizar a implementação do gerador sintático utilizando Bison (para a análise sintática) em conjunto com o Flex (como analisador léxico), juntamente com uma implementação na linguagem C.

Analisador Sintático:

Inicialmente no arquivo “translate.y” importamos e definimos itens que utilizamos e declaramos uma função e variável externas (advindas do analisador léxico):

```
#define YYSTYPE char*
#include "tabela_simbolos.h"
#include <stdlib.h>
#include <stdio.h>
//#define YYDEBUG 1

extern int yylex();
extern int yylineno;
```

Figura 8 - Importações necessárias.

Em seguida declaramos todos os tokens que utilizamos e entramos com a nova gramática descrita acima logo em seguida tendo como variável de partida “start”. Abaixo temos algumas tokens declaradas:

```
%token INT
%token FLOAT
%token CHAR
%token STRING
%token BOOLEAN
%token VOID
%token RETURN
%token BREAK
%token CONTINUE
%token NOT
%token OR
%token AND
%token IF
%token ELSIF
%token ELSE
%token WHILE
%token NUM_I
%token NUM_F
%token STR
%token CH
%token TRUE
%token FALSE
```

Para a exibição em tela dos possíveis erros obtidos durante a análise sintática implementamos 3 funções: “*yyerror()”* (invocada pelo próprio analisador sintático), “*Erro_N_Dec()*” (invocada quando um identificador é invocado sem antes ser declarado, ou seja, sem estar presente na tabela de símbolos) e “*Erro_Redec()*” (invocada quando um identificador já declarado é re-declarado, ou seja, quando uma tentativa de adicionar na tabela de símbolos acontece com o identificador já presente).

```c
void yyerror(const char* s) {
    printf("Programa sintaticamente incorreto!\n");
    fprintf(stderr,"Erro sintático na linha %d -> %s\n", yylineno,s);
    exit(1);
}

void Erro_N_Dec(const char* s) {
    printf("Programa sintaticamente incorreto!\n");
    fprintf(stderr,"Erro o identificador '%s' na linha %d nao foi declarado!\n", s, yylineno);
    exit(1);
}

void Erro_Redec(const char* s) {
    printf("Programa sintaticamente incorreto!\n");
    fprintf(stderr,"Erro o redeclaração do identificador '%s' na linha %d !\n", s, yylineno);
    exit(1);
}
```

Figura 9 - Funções de erros sintáticos.

Tabela de Símbolos:

Inicialmente, definimos estrutura (uma linha da lista) no arquivo “*tabela_simbolos.h”* que vai conter o nome do identificador (valor semântico) e o tipo do mesmo. Além disso, outra struct (a tabela) foi declarada, esta contém um vetor com 1024 posições da primeira estrutura passada, dessa forma temos nossa tabela de símbolos.

Além disso, temos a função chamada *Printa_Tabela_Simbolos()* que basicamente percorre a tabela de símbolos e imprime cada um dos seus respectivos valores de id e type, que são o nome e o tipo, respectivamente, de cada variável ou função declarada no código de entrada.

```c
typedef struct
{
    char type[10];
    char id[100];
} Entrada_Tabela;

typedef struct
{
    Entrada_Tabela Tabela[1024];
    unsigned int Proxima_Entrada;
} Tabela_Simbolos;

void Nova_Tabela(Tabela_Simbolos *Tabela_Simbolos){
  Tabela_Simbolos->Proxima_Entrada = 0;
}

void Printa_Tabela_Simbolos(Tabela_Simbolos *Tabela_Simbolos){
  printf("Tabela de Simbolos:\n");
  printf("| N | ID | Tipo\n");
  printf("------------\n");
  for (unsigned int i = 0; i < Tabela_Simbolos->Proxima_Entrada; i++){
      printf("| ");
      printf("%d | ",i+1);
      printf("%s", (*Tabela_Simbolos).Tabela[i].id);
      printf(" | ");
      printf("%s",(*Tabela_Simbolos).Tabela[i].type);
      printf(" | ");
      printf("\n");
```

Figura 10 - Estruturas básicas da tabela de símbolos

E por último, foram implementadas as funções *Adiciona_tipo_tabela(), Adiciona_Entrada_tabela_Simbolos()* e *Entrada_Existente_Tabela()*, que são responsáveis por adicionar um tipo a uma linha da tabela, adicionar um identificador na tabela e verificar se um identificador está presente na tabela, respectivamente.

```c
void Adiciona_tipo_tabela(Tabela_Simbolos *Tabela_Simbolos, char *type){
  strcpy(Tabela_Simbolos->Tabela[Tabela_Simbolos->Proxima_Entrada].type,type);
}

void Adiciona_Entrada_Tabela_Simbolos(Tabela_Simbolos *Tabela_Simbolos, char *id){
  strcpy(Tabela_Simbolos->Tabela[Tabela_Simbolos->Proxima_Entrada].id, id);
  Tabela_Simbolos->Proxima_Entrada++;
}


int Entrada_Existente_Tabela(Tabela_Simbolos *Tabela_Simbolos, char *id){
  for (int i = 0; i < Tabela_Simbolos->Proxima_Entrada; i++){
    if (strcmp(Tabela_Simbolos->Tabela[i].id, id) == 0 ? 1 : 0)
      return 1;
  }
  return 0;
}
```

Figura 11 - Funções auxiliares da tabela de símbolos

As chamadas destas funções são realizadas durante o processo de análise sintática, em momentos onde identificadores são referenciados e declarados, além de ao final, toda tabela ser impressa. Caso em declarações ou referências a variáveis não presentes na tabela aconteçam é invocado as funções que imprimem as mensagens de erro na saída padrão.

Como Executar:

Inicialmente para compilar, é necessário executar o comando dentro do diretório do trabalho:

bison -t -o Codigo/yacc.tab.c -d Codigo/translate.y && flex -o Codigo/lex.yy.c Codigo/lex.l && gcc Codigo/yacc.tab.c Codigo/lex.yy.c -o maybe

Ou executar o arquivo makefile através do comando:

make

Dessa maneira, os arquivos *lex.yy.c*, *yacc.tab.c* e *yacc.tab.h* serão gerados, em seguida compilados e ao final, o executável *maybe* será gerado. Podendo ser executado da seguinte forma:

./maybe < nomedoarquivo.may

Sendo que "*nomedoarquivo.may"* deverá ser substituído pelo nome do arquivo da linguagem MAYBE de entrada. Os arquivos utilizados por nós se encontram na pasta Entradas, sendo necessário o uso do caminho *"Entradas/nomedoArquivo.may".*

Testes de Execução

Realizamos a implementação de diversos arquivos de entrada (errados e certos sintaticamente) para fins de teste e demonstração da eficácia do nosso analisador léxico e sintático.

Execução de arquivos de entrada com erros:

- Erro de variável não declarada:
  Entrada:

```
1  void funcErrada(int a){
2      int b = d; /*Variavel d nao declarada*/
3      return (((a+b)*2)/3);
4  }
```

Figura 12 - Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro o identificador 'd' na linha 2 nao foi declarado!
```

Figura 13 - Saída

- Erro de função não declarada:

Entrada:

```
1  int a = 2;
2  a  = minhafuncao();
3  /*Funcao nao declarada*/
```

Figura 14 - Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro o identificador 'minhafuncao' na linha 2 nao foi declarado!
```

Figura 15 - Saída

- Erro de escopo vazio:

  Entrada:

```
1  int main(){
2      int a=0;
3      if(a == 2){
4          /*Escopo vazio*/
5      }
6  }
```

Figura 16 - Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 5 -> syntax error
```

Figura 17 - Saída

- Erro dupla declaração:
  Entrada:

```
1  void main(){
2      int a b = 3; /*Dupla declaracao, nao suportada*/
3  }
```

Figura 18 - Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 2 -> syntax error
```

Figura 19 - Saída

- Parentese desbalanceado:

  Entrada:

```
1  void main(){
2      int d = 3;
3      if(d == 2
4          d = 3; /* parentese desbalanceado */
5
6  }
```

Figura 20 - Entrada Errada

  Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 4 -> syntax error
```

Figura 21 - Saída

- Else com expressão:

  Entrada:

```
1  int a=2;
2
3  if (a<>2){
4      a=2;
5  }
6  else (a<2){ /*else com expressao*/
7      a = -2;
8  }
```

Figura 22 - Entrada Errada

  Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 6 -> syntax error
```

Figura 23 - Saída

- Função sendo declarada dentro de outra função:

  Entrada:

```
1  int main(){
2      int outrfuncao(){ /*Funcao sendo declarada dentro de outra.*/
3          int b = 2;
4          return b;
5      }
6      return outrfuncao();
7  }
```

Figura 24 - Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 2 -> syntax error
```

Figura 25 - Saída

- Erro ao declarar função sem declarar tipo de parâmetro:

  Entrada:

```
2  int c = 2;
3  c = c + 2;
4  void funcErrada(a){ /*Parametro de função sem declaração de tipo*/
5      int b = a;
6      return b;
7  }
```

Figura 26 - Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 4 -> syntax error
```

Figura 27 - Saída

- Erro ao usar palavra reservada:

  Entrada:

```
1  void umafuncao(int a, string c){
2      int string= 2; /*Uso de palavra reservada*/
3      return;
4  }
```

Figura 28- Entrada Errada

Saída:

```
Programa sintaticamente incorreto!
Erro sintático na linha 2 -> syntax error
```

Figura 29 - Saída

Execução de arquivos de entrada sintaticamente corretos:

- Teste de declarações, operadores, parâmetros de função, vetores, palavras reservadas, laço While, condicionais e expressões relacionais:

  Entrada:

```
int nomefuncao(string umastring, float umfloat){
    int vetor [3];
    vetor [1] = 30;
    if(1 < vetor [1]){
        return 1;
    }
    elsif(0 < vetor [1]){
        return -1;
    }
    else {
        return 10;
    }
    int i = 0;
    while(i<vetor[1]){
        i = i + 1;      /* Pablo ficou com medo desse arquivo de teste nao rodar */
        if((vetor[0]>100) and (umastring == "Pare")){
            break;
        }
        elsif ((i>101) or umfloat > 13.3){
            return;
        }
        else{
            continue;
        }
        vetor[0] = i;
    }
    return 0;
}
```

Figura 30- Entrada Correta

Saída:

```
Programa sintaticamente correto!
Tabela de Simbolos:
| N | ID | Tipo
-------------
| 1 | nomefuncao | int |
| 2 | umastring | string |
| 3 | umfloat | float |
| 4 | vetor | int |
| 5 | i | int |
-------------
```

Figura 31 - Saída

- Teste de balanceamento dos parênteses, operadores lógicos, operadores aritméticos, operadores relacionais, condicionais

Entrada:

```
int main(int ferrerinha, string thomas){
    int samuel [5];
    char c = 'b';
    int soma = ((ferrerinha / 2) + (5 * (ferrerinha - 5)));

    if(ferrerinha == 2){
        samuel[1] = ferrerinha;
    }
    elsif(((samuel[1] >= 3) or (c <> 'b')) and (1 <= 0)){
        return;
    }
    elsif(not (samuel[1] <= 3) and not ((c == 'b') or (1 >= 0))){
        return;
    }
}
```

Figura 32 - Entrada Correta

Saída:

```
Programa sintaticamente correto!
Tabela de Simbolos:
| N | ID | Tipo
-------------
| 1 | main | int |
| 2 | ferrerinha | int |
| 3 | thomas | string |
| 4 | samuel | int |
| 5 | c | char |
| 6 | soma | int |
-------------
```

Figura 33 - Saída

- Teste do uso de parênteses em operações aritméticas, além do operador “-” como inversor de sinal.

  Entrada:

```
int main(){
    int b=2;
    int a = 3;
    int c= 1.3;
    int d = (a+(b+c));
    int e = ((a+b)+c+d);
    int f = ((a+b+c)+(e+3));
    int g = (a * (b+c) / (4+1+1+1+-3+1+1+1) );
    int h = ((((((((((a+b - -c+d)*2)))+(2/3))))*2))));
    int i = 2 - 3;
    int j = -i;
    int k = (-main());
    c = 2 + (- d);
    return 2;
}
```

Figura 34 - Entrada Correta

Saída:

```
Programa sintaticamente correto!
Tabela de Simbolos:
| N | ID | Tipo
-------------
| 1 | main | int |
| 2 | b | int |
| 3 | a | int |
| 4 | c | int |
| 5 | d | int |
| 6 | e | int |
| 7 | f | int |
| 8 | g | int |
| 9 | h | int |
| 10 | i | int |
| 11 | j | int |
| 12 | k | int |
-------------
```

Figura 35 - Saída

Conclusão

Como conclusão para essa parte do trabalho, temos que ele foi de suma importância para entendimento de certas partes da disciplina (como uso da gramática e tokens). Tivemos grandes dificuldades em relação ao uso dos parênteses em nossa gramática, principalmente em situações onde o mesmo é opcional (onde o usuário pode colocar ou não) e com relação a comunicação entre o

Flex e o Bison. Apesar disso, durante a realização do trabalho começamos a sentir um gosto pela implementação, provavelmente causado pelo esclarecimento das dúvidas e funcionamento gradativo de acordo com as modificações realizadas.

Referências

DEGENER, Jutta. ANSI C Yacc grammar. 1995. Disponível em: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html. Acesso em 21 set. 2021.

GRAMÁTICAS: fundamentos e exemplos.TESES stmt FECHAPARE Disponível em: https://www.ime.usp.br/~fmario/mac122-15/gramatica.html. Acesso em: 15 set. 2021.

QUADROS, Everton Quadros Everton. **Software Developer © 2020 Implementando um analisador léxico usando o Flex**. Disponível em: https://eqdrs.github.io/compilers/2019/09/08/implementando-um-analisador-lexico-usando-o-flex.html. Acesso em: 15 set. 2021.

FULL Grammar specification. Disponível em: https://docs.python.org/3/reference/grammar.html. Acesso em: 16 set. 2021.

TAVARES, Tiago. 11 - Começo da Calculadora em Yacc/Bison. Youtube, 27 de mar. de 2020. Disponível em: https://www.youtube.com/watch?v=7aie74eD6Mg&t=170s&ab_channel=TiagoTavares
. Acesso em: data do acesso. 06 de out 2021.

TAVARES, Tiago. 12 - Completando calculadora no Yacc/Bison - fazendo a parte do Lex/Flex. Youtube, 27 de mar. de 2020. Disponível em: https://www.youtube.com/watch?v=waP8Wh3SE4k&ab_channel=TiagoTavares
. Acesso em: data do acesso. 06 de outubro 2021