UNIVERSIDADE FEDERAL DE VIÇOSA - *CAMPUS* FLORESTAL

Fábio Trindade Ramos - mat.3869
Gabriel Vitor da Fonseca Miranda - mat.3857
Lázaro Izidoro Bodevan Maia - mat.3861
Pedro Augusto Maia Silva - mat.3878
Caio Augusto Campos - mat.3042

Trabalho Prático 3
Compiladores

Trabalho prático da disciplina Compiladores - CCF 441, do curso de Ciência da Computação da Universidade Federal de Viçosa - Campus Florestal.

Professor(a): Daniel Mendes

Florestal
2022

Sumário

1. Introdução

Este trabalho tem como principal objetivo criar um analisador léxico para um determinada linguagem criada pelo grupo. A linguagem escolhida para fazer este trabalho foi uma linguagem a qual faz alusão aos “memes”, cuja definição mais formal é: “No contexto da internet, meme é uma mensagem quase sempre de irônico que pode ou não ser acompanhada por uma imagem ou vídeo e que é intensamente compartilhada por usuários nas mídias sociais.”. Dessa forma, cada palavra reservada da linguagem C foi transformada em uma palavra que faz referência aos memes. Para este primeiro trabalho, foi necessário criar um analisador léxico para que todo o fluxo de caracteres da linguagem para que a mesma fosse reconhecido pelo analisador. Diante disso, foi usada a ferramenta lex para fazer este analisador léxico reconhecer a linguagem como entrada e transformá-la em um fluxo de tokens.

O nome da linguagem sobre a qual trataremos neste trabalho foi batizada de “MemeAllusion” a linguagem dos memes. Este nome foi escolhido pensando na tradução literal da palavra em inglês “Allusion” que significa “alusão”, sendo no contexto global, “alusão a memes” referindo-se de forma conceitual o fato de que a linguagem fará alusão a memes que contém os seus respectivos repertórios de palavras mais comuns. Vale salientar que as palavras utilizadas em nossa linguagem possuem alguma relação com as palavras reservadas, mas só fazem sentido quando se trata de memes e linguagem informal.

Esta linguagem contém todas as características necessárias para despertar o tom humorístico do usuário na hora de programar. Isso permite ao programador uma maior proximidade e conforto com a sua própria codificação, a fim de minimizar o estresse com relação a sintaxe de uma linguagem de programação. Algumas tomadas de decisões da linguagem nos impediram de que algumas palavras reservadas tivessem esta denotação, mas mantendo o foco principal que é resgatar o humor do programador, utilizando de uma metalinguagem, onde as palavras formam um meme com relação a linguagem inspiradora (linguagem C).

2. Análise Léxica

A análise léxica tem como objetivo converter um fluxo de caracteres em um fluxo de tokens para que possa ser posteriormente realizada a análise sintática. Para que o analisador léxico se tornasse mais completo, foi implementada uma métrica de identificação de erros definidos pelo grupo, onde o analisador léxico indica a linha onde um token possui um erro, indicando qual caractere está causando este erro, apesar de reconhecer os caracteres posteriores.

A proposta é que um erro léxico aconteça quando se usa caracteres inválidos definidos pela linguagem em qualquer palavra. Dessa forma, auxiliando o programador a reconhecer mais facilmente onde e como um erro ocorreu durante a compilação do programa.

Além disso, foi definido um tipo comentário que serve para que quando um código em "MemeAllusion" tenha comentários, o analisador léxico o ignore e reporte o comentário ignorado pelo mesmo. Isso serve apenas para visualização do funcionamento do analisador léxico, sendo que tais impressões não serão realizadas durante a compilação completa do programa.

2.1 Definições regulares

Para a construção do analisador léxico, foram criadas algumas definições regulares na gramática da linguagem "MemeAllusion" para que sejam reconhecidos lexemas da linguagem.

- *delim* → tabulações e espaços em branco
- *ws* → Uma ou mais tabulações e/ou espaços em branco
- *digito* → dígito entre 0 e 9.
- *numero* → um ou mais dígitos, podendo ser negativos ou positivos
- *letra* → caractere entre A e Z , podendo ser maiúsculas ou minúsculas
- *booleano* →valores booleanos realmente(true) e mentiraa(false).
- *operadoresLogic* → sendo eles "MenorQue", "MaiorQue", "gemeas", "notIgual", "MenorIgualQue", "MaiorIgualQue", "ii, e "ou".
- *id* → lexema que encontra um identificador da linguagem.
- operadores →operadores aritméticos +, -, *, /, **.

2.2 Regras da linguagem

Esta seção contém as regras da linguagem de programação MemesAllusion, ou seja, o que a linguagem reconhece como um lexema e como ela trata determinados caracteres na leitura de um fluxo de caracteres. Levando em consideração e aprofundando um pouco mais nas definições regulares da seção anterior temos que:

- id: É um padrão que é reconhecido por pelo menos uma letra seguida de uma ou mais letra ou digito.
- numero: É um padrão de números inteiros que são reconhecidos por um ou mais digite(números positivos) ou um sinal de negativo seguido por um ou mais digito(números negativos).
- decimal: É um padrão de números decimais que são reconhecidos por um ou mais digito seguidos de um ponto(.) seguido de um ou mais digito.
- ws: Faz a leitura de espaços em branco e tabulações a fim de ignorá-los na leitura do fluxo de caracteres.

Em resumo do que foi dito acima, podemos dizer que a nossa linguagem reconhece variáveis, números inteiros e números decimais de acordo com as especificações citadas acima, e ignora espaços em branco e tabulações. Além disso, ele reconhece outros lexemas, como, palavras-chave e palavras reservadas, as quais são apresentadas na seção 2.4. Outras regras da nossa linguagem são o reconhecimento de comentários, ponto e vírgula e dois pontos. Vale ressaltar que na nossa linguagem há a existência de caracteres que não são permitidos na nossa linguagem.

Os comentários devem ser inicializados e finalizados com "//", e dentro desses podem existir qualquer tipo de caractere, os quais serão ignorados pela linguagem no momento da compilação.

Os caracteres que não são permitidos na nossa linguagem são:ponto final(exceto para números decimais), ^, ~ e $.

2.3 Executando o analisador léxico

Assim que todas as definições regulares e as regras da gramática estiverem prontas no arquivo Flex, é necessário criar o executável para o analisador léxico para poder reconhecer o fluxo de caracteres e transformá-los em fluxo de tokens. Para isto, basta executar o seguinte comando: "*flex lex.l*", "*gcc lex.yy.c"*, e " *./a.out* < *nomeArquivo*".

2.4 Fluxo de tokens

Dada a especificação da linguagem em questão, descrevemos o fluxo de tokens da mesma. Conforme acordado, os delimitadores de blocos e expressões são reconhecidos da mesma forma como na linguagem C, sendo "{ }" e "( )" respectivamente. Essas e as demais definições das palavras-chave e palavras reservadas são mostradas na tabela abaixo.

print	NuncaNemvi | seliga

for	paraZe
scanf	leiame
while	uaiou
if	hipotese
else	quediafoisso
case	cazemito
switch	trocar
void	nadaNaoSS
int	Daniel
long	Glaucia
float	Thais
double	Fabricio
char	Nacif
struct	fusao
funcao()	encargo
ponteiro	_x9
multiplicação	veis
divisão	Socialismo
soma	mais
subtração	menos
break	parecomisso
malloc	alok
sizeof	saizonof
=(atribuição)	receba
<	MenorQue
>	MaiorQue
==	gemeas
!=	notIgual
<=	MenorIqualQue

>=	MaiorIqualQue
return	leiNewton
do	faz
;	;
True	realmente
False	mentiraa
typedef	tipoIsso
goto	vai
continue	toBeContinua
&&	ii
||	ou
default	padraozinho

Além disso, foram desenvolvidas duas formas de apresentação dos lexemas identificados, sendo um seguindo a forma “Foi encontrado um …. Lexema: …” e a segunda sendo apenas “->”. Por questões de melhora de visualização, esta segunda forma foi adotada.

2.5 Resultados

Para o teste da nossa linguagem, foram utilizados os diversos lexemas mapeados e informados na seção anterior. Buscamos a elaboração de cenários que nos trouxessem erros léxicos e que mostrassem o reconhecimento dos variados tokens. Foi elaborada uma “listinha de programação” para exemplificar os cenários. Vale ressaltar que os tokens reconhecidos são impressos da forma: “-> + token”, já os que não foram reconhecidos são impressos com o erro seguido da linha que identificou-se o token não reconhecido.

1) Realizar um programa que lê o valor de um inteiro do terminal e procura o múltiplo mais próximo do passado por parâmetro.

```
// Programa que lê o valor de uma entrada e acha o multiplo de 7 mais próximo//

Daniel ilaVamosNos(Daniel mult){
    Daniel valor receba leiame;
    Daniel i receba 0;
    faz{
        hipotese(valor gemeas 0){
            lei3Newton 0;
        }
        Daniel div receba (valor - i)/mult;
        hipotese((valor-i - div*mult) gemeas 0){
            lei3Newton valor-i;
        }
    }uaiou(valor MenorQue 0)
}
```

Figura 1.

```
Comentario ignorado::  Programa que lê o valor de uma entrada e acha o multiplo de 7 mais próximo
-> Daniel
-> ilaVamosNos
-> (
-> Daniel
-> mult
-> )
-> {
-> Daniel
-> valor
-> receba
-> leiame
-> ;
-> Daniel
-> i
-> receba
-> 0
-> ;
-> faz
-> {
-> hipotese
-> (
-> valor
-> gemeas
-> 0
-> )
-> {
-> lei3Newton
-> 0
-> ;
-> }
-> Daniel
-> div
```

Figura 2.

Neste exemplo, foram utilizados os tokens referentes aos comentários, definição de variável, condicionais, leitura pelo terminal e laço de repetição. A saída reconhece esses tokens bem como a definição dos escopos.

2) Desenvolva uma função que calcule o fibonacci de um número informado pelo usuário.

```
Daniel ilaVamosNos()
{
   seliga(Insere ai um numero mane);
   Daniel n = leiame();
   seliga("Ti liga no fibonacci: ");
   seliga(fibonacci(n))
   lei3Newton 0;
}
Daniel fibonacci(Daniel num)
{
   hipotese(num gemeas 1 ou num gemeas 2)
      lei3Newton 1;
   quediafoisso
      lei3Newton fibonacci(num-1) + fibonacci(num-2);
}
```

Figura 3.

```
-> Daniel
-> ilaVamosNos
-> (
-> )
-> {
-> seliga
-> (
-> Insere
-> ai
-> um
-> numero
-> mane
-> )
-> ;
-> Daniel
-> n
:: Erro Léxico :: Linha -> 4 Erro de lexema: =
-> leiame
-> (
-> )
```

Figura 4.

Neste cenário foram desenvolvidas 2 funções (main e fibonacci). A Fibonacci foi desenvolvida de forma recursiva e é invocada pela "ilaVamosNos", função main. Nota-se que o usuário utilizou o token "=" para atribuir o resultado do leitor à variável "n" contudo tal token não é reconhecido pela linguagem. Logo, nota-se uma mensagem de erro.

3) Programa que lê um caractere e converte em uma das opções válidas. O programa só pode encerrar se alguma opção for escolhida.

```
Nacif[] ilaVamosNos(){

    Nacif letra receba tipoIsso(Nacif)leiame();
    trocar(letra){
        cazemito "a":
            lei3Newton "Opcao 1";
            parecomisso;
        cazemito "b":
            lei3Newton "Opcao 2";
            parecomisso;
    }
    vai 3;
}
```

Figura 5.

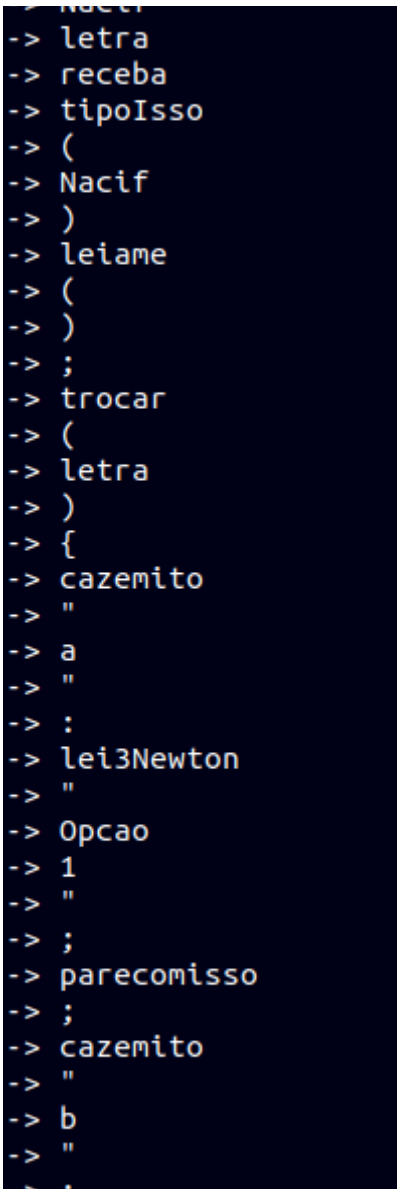


Figura 6.

Nesse exemplo foram explorados os tokens de redirecionamento do fluxo (goto), conversão de tipos e escolha entre 2 opções. Os Lexemas foram reconhecidos e apresentados no terminal de saída.

3. Análise Sintática

Análise Sintática é um processo de um compilador (de uma linguagem de programação), é a segunda fase da compilação onde se analisa uma sequência que foi dada entrada (via um arquivo de computador ou via teclado, por exemplo) para verificar sua estrutura gramatical segundo uma determinada gramática formal. Este processo trabalha em conjunto com a análise lexical (primeira etapa, onde se verifica de acordo com determinado alfabeto) e análise semântica (terceira etapa, onde verificam-se os erros semânticos).

A análise sintática transforma um texto na entrada em uma estrutura de dados, em geral uma árvore, o que é conveniente para processamento posterior e captura a hierarquia implícita desta

entrada. Através da análise lexical é obtido um grupo de tokens, para que o analisador sintático use um conjunto de regras para construir uma árvore sintática da estrutura.

Em termos práticos, por exemplo, pode também ser usada para decompor um texto em unidades estruturais para serem organizadas dentro de um bloco.

A vasta maioria dos analisadores sintáticos implementados em compiladores aceitam alguma linguagem livre de contexto para fazer a análise. Estes analisadores podem ser de vários tipos, como o LL, LR e SLR.

3.1 Alterações - Parte 2

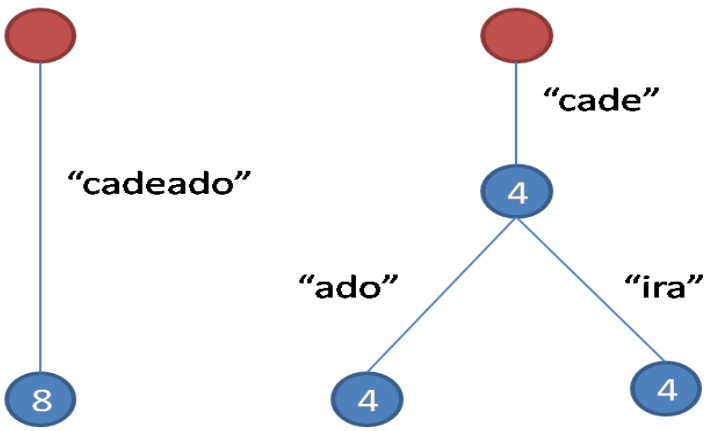
Para esta segunda parte do trabalho, retiramos as expressões regulares dos booleanos e operadores lógicos com o objetivo de facilitar a utilização dos mesmos no yacc. Em contrapartida, foi adicionado o suporte a constantes para construção dos códigos.

Soma-se a isso, uma definição realizada pelo grupo como uma condição de parada no lex, a descoberta de um erro léxico. Isso permite ao programador da linguagem saber onde o erro foi cometido e, além disso, evita muitas mensagens de erro na tela ao executar o programa.

Além disso, permite que seja mais simples a correção de erros, sabendo que cada erro pode ocasionar outros erros. Pensando nisso, além de facilitar a compilação, facilita ao programador corrigir os seus erros.

3.2 Tabela de Símbolos

Para a construção da tabela de símbolos, foi implementada uma árvore PATRICIA (*Practical Algorithm To Retrieve Information Coded In Alphanumeric*). Isso se deve ao fato de que a tabela de símbolos tem por objetivo ser consultada. Dessa forma, utilizando da estrutura de árvore a PATRÍCIA, é possível obter um melhor desempenho no que diz respeito à inserção e busca de palavras, o que se encaixa perfeitamente no contexto atual, visto que ela durante a inserção, impede que palavras repetidas sejam inseridas, evitando ter que percorrer toda uma estrutura como acontece em uma lista encadeada.



A tabela de símbolos é composta pelos campos “Símbolo”, “Tipo do Símbolo”, “Tipo” e “Nº Linha”, conforme mostrado na figura abaixo.

```
printf("_____________________________________________________________________________\n");
printf("|                          Tabela de símbolos                                 |\n");
printf("_____________________________________________________________________________\n");
printf("|          Simbolo          |    Tipo do Simbolo   |          Tipo          | Nº Linha  |\n");
imprimeArvore(no);
printf("_____________________________________________________________________________\n");
```

Isso permite verificar após a análise sintática todos os símbolos presentes na construção do algoritmo, sem que os mesmos estejam de forma repetida. Com os campos de tipo e tipo do símbolo, dá para perceber como a análise sintática está classificando cada um dos símbolos inseridos. Com o campo da linha é possível perceber que os símbolos são inseridos apenas na sua declaração ou quando aparece pela primeira vez no algoritmo construído em “MemeAllusion”.

Os tipos de símbolos que são inseridos na tabela são:

- Operadores aritméticos → + , - , * , /
- Strings → Que são classificadas como um vetor também como tipo do símbolo
- Operadores lógicos →

MenorQue
MaiorQue
gemeas
notIgual
MenorIqualQue
MaiorIqualQue
ii
ou

- Variáveis → São elementos presentes no código que são criados durante a execução do mesmo, os quais recebem alguma classificação de tipos.

- Constantes → São números que aparecem declarados no código, de forma que o seu valor não pode ser alterado, visto que não possui uma variável que o contém para ser acessado posteriormente.

3.3 Translate.y

Para a construção do arquivo Translate.y , o qual vai fornecer os comandos para a construção do analisador sintático Yacc, foi necessário a realização dos retornos dos tokens criados pelo analisador léxico. No trabalho anterior, como os tokens obtidos eram apenas impressos na tela, a mudança a ser realizada foi a troca dos “printf’s” para “return”.

Além disso, é necessário que o analisador sintático reconheça os tokens, portanto tendo que ser declarados no escopo do arquivo. Assim, os tokens reconhecidos na análise léxica, podem ser identificados na análise sintática, sabendo o que cada token representa na gramática da linguagem.

Soma-se a isso a construção das gramáticas da linguagem e, como cada escopo, palavra reservada e caracteres devem se encaixar na construção do algoritmo em MemeAllusion.

Isso permite ao analisador sintático, ter as ferramentas necessárias para dizer se o algoritmo construído está sintaticamente correto. Assim, caso o programador utilize de forma errada uma declaração de variável ou esqueça um ‘;’ , o analisador sintático irá retornar uma mensagem de erro na linha cuja infração foi acometida.

3.4 Executando o analisador sintático

Para usar o analisador léxico em conjunto do analisador sintático, basta apenas digitar o comando “make”, e em seguida digitar o arquivo de entrada que será usado, exemplo: “make”, “./a.out < arquivo.txt”. Se quiser gerar um arquivo que mostre a saída, basta digitar: “./a.out < arquivo.txt > saida.txt”.

3.5 Resultados

Nesta parte serão apresentados quatro exemplos de programas sintaticamente corretos e com suas devidas tabelas de símbolos, todavia também serão apresentados alguns exemplos sintaticamente incorretos:

Programas sintaticamente corretos:

- **Exemplo 1: arquivo: “LpMemes1.txt”**

```
1       Daniel ilaVamosNos(){
2           Daniel valor1;
3           valor1 receba 0;
4           paraZe(Daniel i receba 0; i MenorQue 0; i receba i + 1){
5               valor1 receba valor1 + i;
6           }
7           seliga("%d", valor1);
8           leiNewton 0;
9       }

Programa Sintaticamente Correto
```

Figura 1. Exemplo 1 sintaticamente correto.

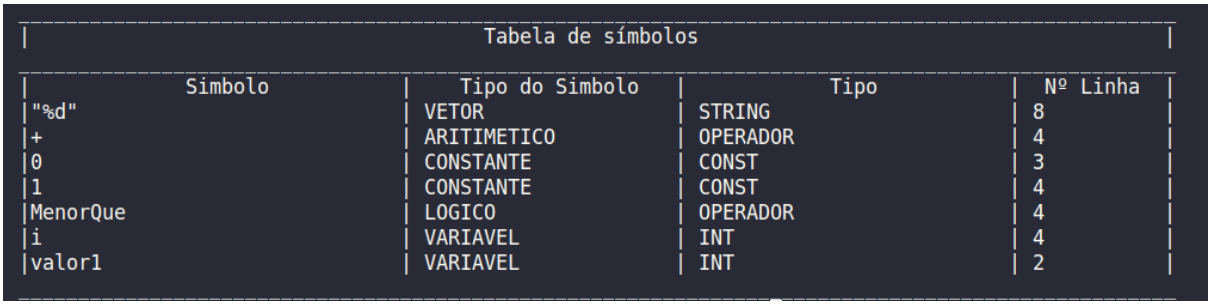
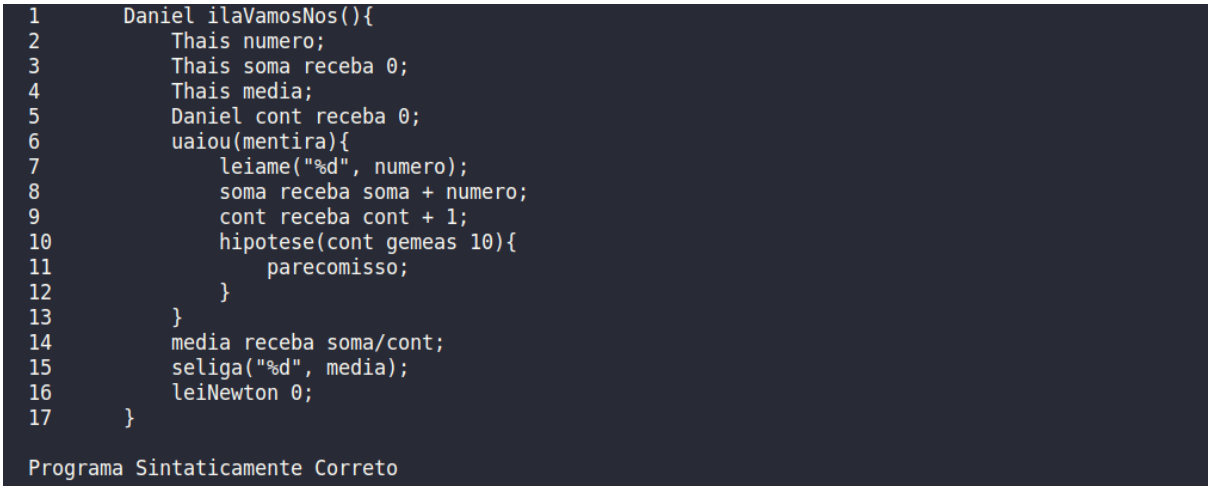


Tabela de símbolos

Simbolo	Tipo do Simbolo	Tipo	Nº Linha
"%d"	VETOR	STRING	8
+	ARITIMETICO	OPERADOR	4
0	CONSTANTE	CONST	3
1	CONSTANTE	CONST	4
MenorQue	LOGICO	OPERADOR	4
i	VARIAVEL	INT	4
valor1	VARIAVEL	INT	2

Figura 2. Tabela de símbolos da figura 1.

- **Exemplo 2: arquivo: “LpMemes2.txt”**



```
1       Daniel ilaVamosNos(){
2           Thais numero;
3           Thais soma receba 0;
4           Thais media;
5           Daniel cont receba 0;
6           uaiou(mentira){
7               leiame("%d", numero);
8               soma receba soma + numero;
9               cont receba cont + 1;
10              hipotese(cont gemeas 10){
11                  parecomisso;
12              }
13          }
14          media receba soma/cont;
15          seliga("%d", media);
16          leiNewton 0;
17      }

Programa Sintaticamente Correto
```

Figura 3. Exemplo 2 sintaticamente correto.

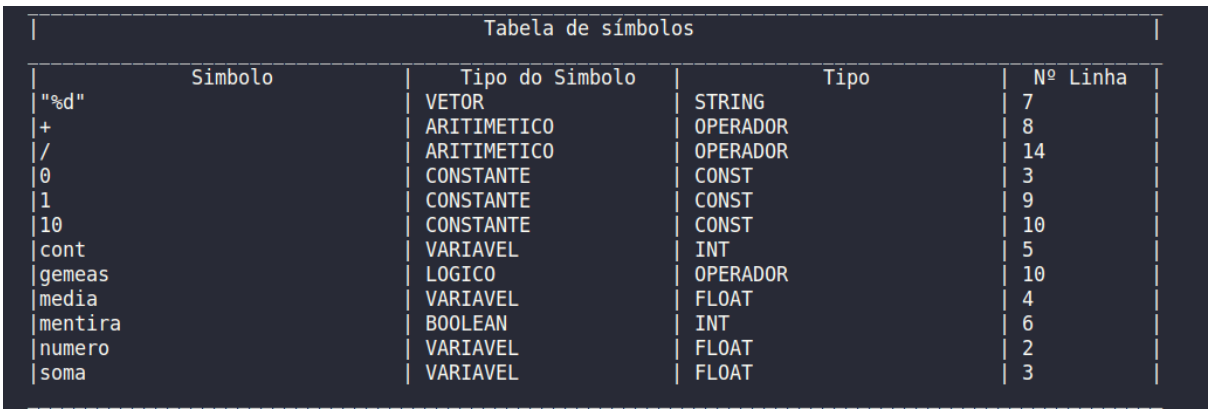


Tabela de símbolos

Simbolo	Tipo do Simbolo	Tipo	Nº Linha
"%d"	VETOR	STRING	7
+	ARITIMETICO	OPERADOR	8
/	ARITIMETICO	OPERADOR	14
0	CONSTANTE	CONST	3
1	CONSTANTE	CONST	9
10	CONSTANTE	CONST	10
cont	VARIAVEL	INT	5
gemeas	LOGICO	OPERADOR	10
media	VARIAVEL	FLOAT	4
mentira	BOOLEAN	INT	6
numero	VARIAVEL	FLOAT	2
soma	VARIAVEL	FLOAT	3

Figura 4. Tabela de símbolos da figura 3.

- **Exemplo 3: arquivo: “LpMemes4.txt”**

```
1       Daniel ilaVamosNos(){
3           Daniel n receba 10;
4           Thais vet(n);
5           Thais menor, maior;
6           paraZe(Daniel i receba 0; i MenorQue 10; i receba i + 1){
7               leiame("%d", vet(i));
8               hipotese(i gemeas 0){
9                   maior receba vet(i);
10                  menor receba vet(i);
11              }
12              hipotese(maior MenorQue vet(i)){
13                  maior receba vet(i);
14              }
15              hipotese(menor MaiorQue vet(i)){
16                  menor receba vet(i);
17              }
18          }
19          seliga("%d",menor);
20          seliga("%d",maior);
21          leiNewton 0;
22      }

Programa Sintaticamente Correto
```

Figura 5. Exemplo 3 sintaticamente correto.

```
|                                    Tabela de símbolos                                    |
|------------------------------------------------------------------------------------------|
|           Simbolo          |    Tipo do Simbolo    |           Tipo          |  Nº Linha  |
|"%d"                        | VETOR                 | STRING                  | 7          |
|+                           | ARITIMETICO           | OPERADOR                | 6          |
|0                           | CONSTANTE             | CONST                   | 6          |
|1                           | CONSTANTE             | CONST                   | 6          |
|10                          | CONSTANTE             | CONST                   | 3          |
|MaiorQue                    | LOGICO                | OPERADOR                | 15         |
|MenorQue                    | LOGICO                | OPERADOR                | 6          |
|gemeas                      | LOGICO                | OPERADOR                | 8          |
|i                           | VARIAVEL              | INT                     | 6          |
|maior                       | VARIAVEL              | FLOAT                   | 5          |
|menor                       | VARIAVEL              | FLOAT                   | 5          |
|n                           | VARIAVEL              | INT                     | 3          |
|vet                         | VARIAVEL              | FLOAT                   | 4          |
```

Figura 6. Tabela de simbolos da figura 5.

- **Exemplo 4: arquivo: “LpMemes5.txt”**

```
1       Daniel ilaVamosNos(){
2           Daniel n, i, fib1 receba 1, fib2 receba 1, soma;
3           leiame("%d", n);
4           paraZe(Daniel i receba 3; i MenorIgualQue n; i receba i + 1){
5               soma receba fib1 + fib2;
6               fib1 receba fib2;
7               fib2 receba soma;
8           }
9           seliga("%d",fib2);
10          leiNewton 0;
11      }

Programa Sintaticamente Correto
```

Figura 7. Exemplo 4 sintaticamente correto.

Tabela de símbolos

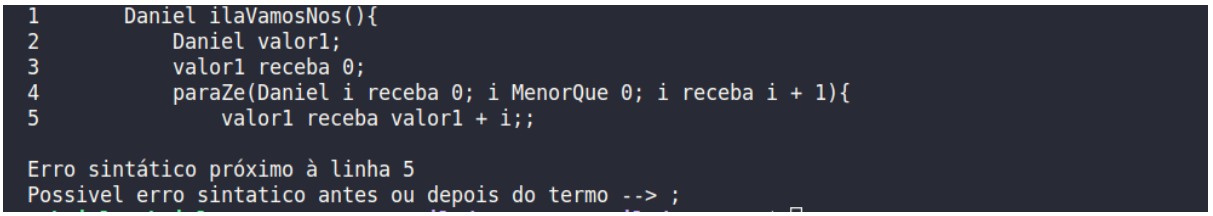
Simbolo	Tipo do Simbolo	Tipo	Nº Linha
"%d"	VETOR	STRING	3
+	ARITIMETICO	OPERADOR	4
0	CONSTANTE	CONST	10
1	CONSTANTE	CONST	2
3	CONSTANTE	CONST	4
MenorIgualQue	LOGICO	OPERADOR	4
fib1	VARIAVEL	INT	2
fib2	VARIAVEL	INT	2
i	VARIAVEL	INT	2
n	VARIAVEL	INT	2
soma	VARIAVEL	INT	2

Figura 8. Tabela de símbolos da figura 7.

Agora serão apresentados alguns erros sintáticos, e a partir desses erros uma mensagem informando onde pode estar o erro sintático.

Programas sintaticamente incorretos:

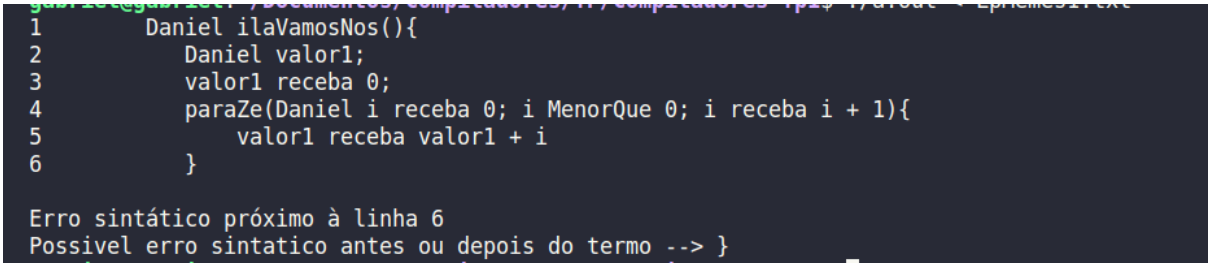
- **Exemplos de erros sintáticos no arquivo “LpMemes1.txt”:**



```
1       Daniel ilaVamosNos(){
2           Daniel valor1;
3           valor1 receba 0;
4           paraZe(Daniel i receba 0; i MenorQue 0; i receba i + 1){
5               valor1 receba valor1 + i;;

Erro sintático próximo à linha 5
Possivel erro sintatico antes ou depois do termo --> ;
```

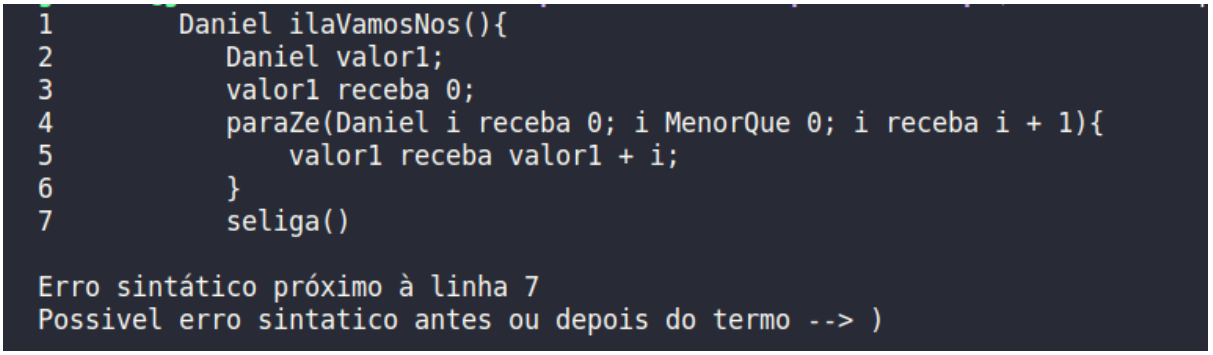
Figura 9. Erro sintático com 2 ‘;’.



```
1       Daniel ilaVamosNos(){
2           Daniel valor1;
3           valor1 receba 0;
4           paraZe(Daniel i receba 0; i MenorQue 0; i receba i + 1){
5               valor1 receba valor1 + i
6           }

Erro sintático próximo à linha 6
Possivel erro sintatico antes ou depois do termo --> }
```

Figura 10. Erro sintático, falta de um ‘;’.



```
1       Daniel ilaVamosNos(){
2           Daniel valor1;
3           valor1 receba 0;
4           paraZe(Daniel i receba 0; i MenorQue 0; i receba i + 1){
5               valor1 receba valor1 + i;
6           }
7           seliga()

Erro sintático próximo à linha 7
Possivel erro sintatico antes ou depois do termo --> )
```

Figura 11. ausência de corpo no seliga.

```
1        ilaVamosNos

Erro sintático próximo à linha 1
Possivel erro sintatico antes ou depois do termo --> ilaVamosNos
```

Figura 12. Erro sintático, falta do tipo da função.

- **Exemplos de erros sintáticos no arquivo "LpMemes2.txt":**

```
1       Daniel ilaVamosNos(){
2           Thais numero;
3           Thais receba

Erro sintático próximo à linha 3
Possivel erro sintatico antes ou depois do termo --> receba
```

Figura 13. Erro sintático, atribuição depois de um tipo'.

```
1       Daniel ilaVamosNos(){
2           Thais numero;
3           Thais soma receba 0;
4           Thais media;
5           Daniel cont receba 0;
6           uaiou(mentira{

Erro sintático próximo à linha 6
Possivel erro sintatico antes ou depois do termo --> {
```

Figura 14. Erro sintático, ausência de um ')'.

```
1       Daniel ilaVamosNos(){
2           Thais numero;
3           Thais soma receba 0;
4           Thais media;
5           Daniel cont receba 0;
6           uaiou(mentira){
7               leiame("%d", numero);
8               soma receba soma + numero;
9               cont receba cont + 1;
10              hipotese(cont gemeas 10){
11                  parecomisso;
12
13          }
14          media receba soma/cont;
15          seliga("%d", media);
16          leiNewton

Erro sintático próximo à linha 16
Possivel erro sintatico antes ou depois do termo --> leiNewton
```

Figura 15. Erro sintático, ausência de um '}'.

```
1       Daniel ilaVamosNos(){
2           Thais valor numero

Erro sintático próximo à linha 2
Possivel erro sintatico antes ou depois do termo --> numero
```

Figura 16: Erro sintático, ausência de um ';'.

Exemplos de erros sintáticos no arquivo “LpMemes4.txt”:

```
1       Daniel ilaVamosNos(){
2           Daniel n receba 10;
3           Thais vet(n)-

Erro sintático próximo à linha 3
Possivel erro sintatico antes ou depois do termo --> -
```

Figura 17: Erro sintático, ‘-’ depois de um ID.

```
1       Daniel ilaVamosNos(){
2           Daniel n receba 10;
3           Thais vet(n);
4           Thais menor, maior;
5           paraZe(Daniel i receba 0; i MenorQue 10; i receba i + 1){
6               leiame("%d", vet(i);

Erro sintático próximo à linha 6
Possivel erro sintatico antes ou depois do termo --> ;
```

Figura 18: Erro sintático, ausência de um ‘)’.

```
1       Daniel ilaVamosNos(){
2           Daniel n receba 10;
3           Thais vet(n);
4           Thais menor, maior;
5           paraZe(Daniel i receba 0; i MenorQue 10; i receba i + 1){
6               leiame("%d", vet(i));
7               hipotese(i gemeas 0)
8                   maior

Erro sintático próximo à linha 8
Possivel erro sintatico antes ou depois do termo --> maior
```

Figura 19: Erro sintático, ausência de um ‘{’.

```
1       Daniel ilaVamosNos(){
2           Daniel n receba 10;
3           Thais vet(n);
4           Thais menor, maior;
5           paraZe(Daniel i receba 0; i MenorQue 10; i receba i + 1){
6               leiame("%d", vet()

Erro sintático próximo à linha 6
Possivel erro sintatico antes ou depois do termo --> )
```

Figura 20: Erro sintático, ausência de um ID no vet().

3.6 Planejamentos futuros

Ao decorrer da implementação do trabalho, percebemos que as implementações seguiam uma estrutura similar a da linguagem de programação C. Então, com o intuito de melhorar a originalidade acerca da linguagem criada pelo grupo, foi decidido que para a entrega 3, as definições da gramática,

expressões regulares, tokens, palavras-reservadas e demais aspectos sobre a linguagem MemeAllusion, sofrerá algumas alterações, as quais serão mencionadas também na próxima entrega. Ou seja, tanto a análise léxica quanto a análise sintática apresentarão algumas mudanças para que o trabalho esteja apresentando características de originalidade e criatividade.

4. Análise Semântica

A análise semântica é um processo de um compilador (de uma linguagem de programação) na qual são verificados os erros semânticos (por exemplo, divisão de um número inteiro por outro número real(float) no padrão ANSI) no código fonte é coletada as informações necessárias para a próxima fase da compilação, que é a geração de código objeto.

A análise semântica verifica e aponta as expressões que quebram qualquer regra determinada pela gramática. Mas o nível de complexidade aumenta quando tratamos de linguagens dependentes de contexto. A análise semântica busca apontar(não resolver) este tipo de erros(dependentes de contexto)

O objetivo da análise semântica é, assim, criar, a partir de um texto-fonte, uma interpretação expressa em alguma notação adequada - geralmente uma linguagem intermediária do compilador. Isto é feito com base nas informações das tabelas e nas saídas dos outros analisadores. Denomina-se semântica de uma sentença o significado por ela assumido dentro do contexto em que se encontra. Semântica de uma linguagem é a interpretação que se pode atribuir ao conjunto de todas as suas sentenças.

4.1 Alterações - Parte 3

Visando trazer maior identidade à nossa linguagem, bem como melhor interpretação dos memes selecionados foram realizadas algumas alterações, tanto nas definições regulares quanto nas etapas da análise sintática.

4.1.1 - Definição de escopo

Para se construir um código nesta linguagem, novos passos devem ser tomados para se definir corretamente cada funcionalidade em seu respectivo escopo. A estrutura que se segue abaixo ilustra a nova definição.

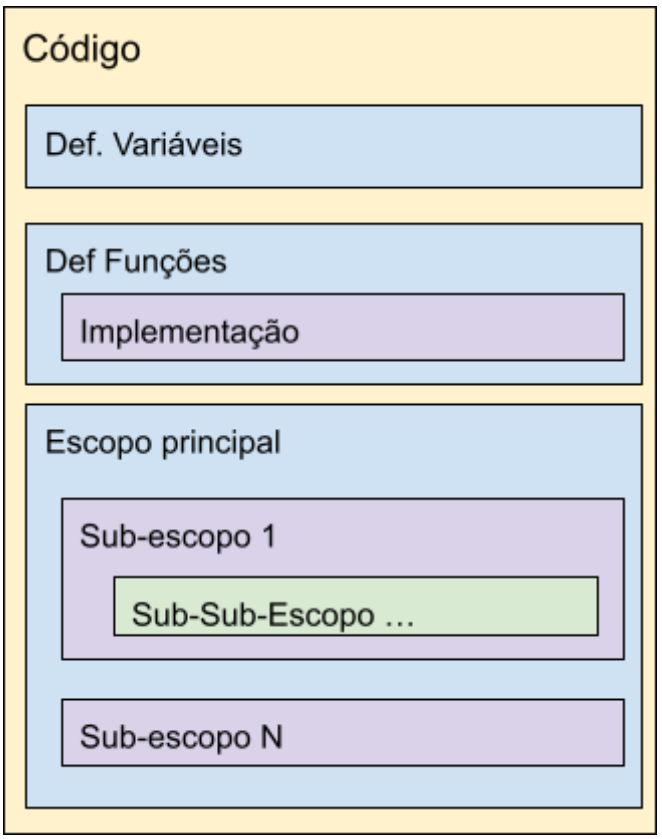


Figura 4.1.2 - Figura representativa da construção de escopos na linguagem

Como veremos a seguir, os escopos da linguagem são definidos por identificadores singulares que podem ser diferentes dependendo de onde se encontram no código e particulares à funções específicas.

Vale ressaltar que não se faz necessário a definição de escopos vazios, ou seja, aqueles que não possuem nenhuma implementação podendo ser omissos sem que isso prejudique a semântica do código.

4.1.2 - Tabela de Tokens

Para os tipos primitivos da linguagem, foram definidos os seguintes identificadores.

Valor	Identificador
int	natashacalderao
float	cazemito
wanessa	double
marinho	char
juliete	struct
irineu	void

Com relação aos operadores aritméticos, foram mantidos os mesmos para poupar complexidade alterando somente os operadores lógicos. Foram realizadas as seguintes definições:

Valor	Identificador
+,-,*,/,**	+,-,*,/,**
true	opaitaon
false	opaitaoff
=	receba
<	mopaz
>	umpoucomais
==	dorime
!=	deixadeserfalsa
<=	pioral
>=	maioral
AND	ihmane
OR	tapassada

Para as demais definições da linguagem foram substituídas as antigas por reais alusões a memes, como se segue na tabela abaixo.

Valor	Identificador
continue	tobecontinua
typedef	tipoisso
scanf	soprusmenino
return	acertoumizeravi
default	padraozinho
printf	seligaein
for	paraze
while	uaiso
if	masoque isso

else	quediafoiisso
case	botaumchopped
switch	reage
malloc	aloka
sizeof	mariacururu
break	parecomisso
do	ameno

Novos delimitadores de escopos também foram inseridos na linguagem. Agora existem 3 escopos principais: definição de variáveis, definição/implementação de funções e corpo principal. Na definição das variáveis se insere todas as variáveis que serão utilizadas no sistema, presentes no escopo principal. Elas são definidas no formato “tipo de dado” + “nome”. Para o escopo de definição de funções é necessário identificar quais funções poderão ser acessíveis ao sistema. Neste caso elas devem estar no formato “retorno” + “nome” + "parâmetros" dentro do referido escopo. Além disso, devem ser fornecidas as devidas implementações. Por fim, no corpo principal são implementadas as lógicas e funcionalidades do sistema. A tabela abaixo descreve a representação final desta nova característica da linguagem.

identificador	Descrição	Exemplo
celokocachoeira	escopo para declaração de variável	celokocachoeira: natashacaldeirao var1
memear	Executar uma função (run)	soprusmenino.memear<var1>
querocafee	Escopo para implementação de funções	querocafee: <cazemito> funcao1 <cazemito var1>: var1 receba 1
ameno	função main	ameno.memear: funcao1<varAux>

Por fim, para trazer uma identidade mais marcante à nossa linguagem, foram redefinidos os delimitadores de início e fim de escopo e funções. Segue as modificações na tabela abaixo.

Representação padrão	Identificador	Descrição
;	:)	Fim da função main
;	;(	Fim de código
{	:	inicio de escopo
}	<-	Fim de escopo

struct { corpo} nome;	->	Nomear Struct
|	|	separador/quebra de linha, usado na definição de variáveis (opcional)
,	<>	separador de identificadores
.	.	acesso à sub métodos ou atributos.

4.2 Implementações

A fim de ilustrar as alterações realizadas para esta 3ª parte, foram convertidos os exemplos descritos anteriormente para a nova estrutura da linguagem. Os exemplos convertidos estão em acordo às novas regras sintáticas/semânticas estabelecidas nesta alteração bem como com os devidos lexemas adaptados.

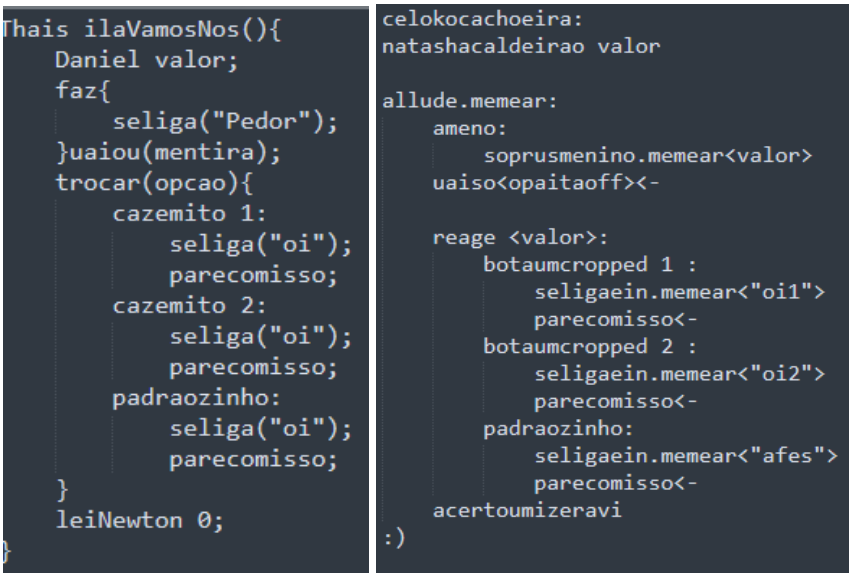
```
Daniel ilaVamosNos(){
    Daniel valor1;
    valor1 receba 0;
    paraZe(Daniel i receba 0; i MenorQue 0; i receba i + 1){
        valor1 receba valor1 + i;
    }
    seliga("%d", valor1);
    leiNewton 0;
}
```

```
celokocachoeira:
natashacaldeirao valor1
natashacaldeirao i

allude.memear :
   valor1 receba 0
   paraze <i receba 0 ; i mopaz 10 ; i receba i + 1 > :
      valor1 receba valor1+i
   <-
   seligaein.memear(valor1)
   acertoumizeravi 0
:)
```

Figura 4.2.1 - A. Implementação anterior. B. Nova implementação

Na imagem 4.2.1.B foi acrescido um escopo de definição de variáveis, iniciando-se com o delimitador ‘celokocachoeira’. O identificador ‘allude’ determina o início da função principal, já a função ‘memear’ é um atributo desta que a executa. Em linhas gerais este exemplo realiza um loop (for) que apenas incrementa o valor da variável ‘valor1’, do tipo ‘natashacaldeirao’ (inteiro). Ao final é impresso o valor da variável com a função ‘seligaein.memear’.



```
Thais ilaVamosNos(){
    Daniel valor;
    faz{
        seliga("Pedor");
    }uaiou(mentira);
    trocar(opcao){
        cazemito 1:
            seliga("oi");
            parecomisso;
        cazemito 2:
            seliga("oi");
            parecomisso;
        padraozinho:
            seliga("oi");
            parecomisso;
    }
    leiNewton 0;
}
```

```
celokocachoeira:
natashacaldeirao valor

allude.memear:
    ameno:
        soprusmenino.memear<valor>
    uaiso<opaitaoff><-

    reage <valor>:
        botaumcropped 1 :
            seligaein.memear<"oi1">
            parecomisso<-
        botaumcropped 2 :
            seligaein.memear<"oi2">
            parecomisso<-
        padraozinho:
            seligaein.memear<"afes">
            parecomisso<-
    acertoumizeravi
:)
```

Figura 4.2.2 - A. Implementação anterior. B. Nova implementação

A Figura 4.2.2 mostra um exemplo do uso dos condicionais, switch e laço de repetição (while). Neste exemplo é lido a variável ‘valor’ através da entrada do usuário e é realizado a escolha da opção referida pelos escopos ‘botaumcropped’ (case). Independente da escolha é impresso para o usuário os textos referentes à escolha.

4.3 Função TypeCorrect

A função TypeIsCorrect tem como objetivo explorar os tipos de cada variável para determinar erros semânticos existentes no código construído pelo programador. Dessa forma, para que tal verificação aconteça, é necessário que seja consultada a tabela de símbolos de forma a verificar se o símbolo foi declarado anteriormente (caso não tenha sido declarado o seu campo ‘Tipo’ vai conter uma string vazia) e essa é a importância de se usar uma árvore como estrutura de dados, visto que as consultas nessa estrutura possuem um custo computacional bem inferior à de uma estrutura lista ou fila encadeada, por exemplo. Dessa forma é possível retornar uma mensagem de erro para o programador indicando qual variável e em qual linha o erro foi acometido.

Vale ressaltar que a função tem como principal objetivo comparar dois símbolos que estejam sendo usados em alguma operação dentro do escopo do algoritmo em MemeAllusion. Assim, quando um tipo inteiro for usado em uma operação com um tipo string por exemplo, a função irá exibir uma mensagem de erro indicando que os tipos não são compatíveis e em qual linha esse erro foi cometido.

4.4 Erros semânticos

Exemplos típicos de erros semânticos são:

- uma variável não declarada
- operações entre tipos de dados diferentes
- atribuição de um literal para outro tipo, como um inteiro em um texto ou vice-versa.
- operações com tipos diferentes

4.5 Funcionalidade - Código de 3 endereços e Arvore Abstrata

Para expressões aritméticas, é considerada a ordem escrita, sabendo que a linguagem não permite "( )". Dessa forma, foi inserido no código dois algoritmos, sendo eles o gerador de código intermediário de 3 endereços para expressões aritméticas e uma árvore abstrata, também para expressões aritméticas, de forma a gerar um arquivo do código de 3 endereços chamado "tresEnderecos.txt" e uma saída de como foi a derivação da expressão na árvore e qual é o resultado obtido da expressão.

Vale ressaltar que para a árvore sintática abstrata implementada, a mesma consegue realizar a derivação de variáveis, mas não consegue realizar os cálculos para variáveis, somente para valores inteiros.

O funcionamento da geração de código de 3 endereços, utiliza como estrutura uma lista duplamente encadeada para poder analisar os valores a direita e à esquerda do operador que está sendo verificado, além disso, realiza a transformação do código utilizando a função "percorrerLista".

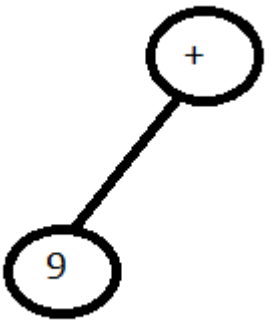
```
void percorrerLista(Lista* lista){
    Pointer aux = lista->inicio;
    while(aux != NULL){
        if(aux->isOperator == 1){
            char *instrucao = (char*)malloc(100*sizeof(char));
            atualizarVar(aux->esq,lista);
            atualizarVar(aux->dir,lista);
            strcpy(instrucao,concat(lista->autoIncremento));
            strcat(instrucao,": ");
            strcat(instrucao,aux->esq->var);
            strcat(instrucao," ");
            strcat(instrucao,aux->string);
            strcat(instrucao," ");
            strcat(instrucao,aux->dir->var);
            emiteInstrucao(instrucao);
            strcpy(aux->dir->var,concat(lista->autoIncremento));
            lista->autoIncremento++;
        }
        aux = aux->dir;
    }
}
```

Assim, ao realizar a função "emiteInstrução", é escrito no arquivo supracitado o código de 3 endereços para a operação. Ao final das instruções, o resultado fica contido na variável temporária que se encontra na última linha do arquivo.

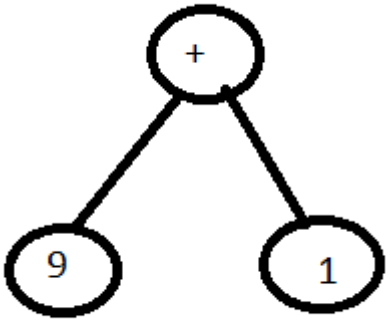
Para a implementação da árvore sintática abstrata, foi usada uma estrutura de nós os quais vão sendo remodelados de acordo com a inserção de operadores, como por exemplo, a operação "9+1", ao inserir o operador "+" na árvore, é obtido a seguinte construção:

Inserção do 9:

Inserção do +:



Inserção do 1:




```
Valor do nó: 9
Valor do nó: +
Valor do nó: 1

Resultado :: 10
```

```
No *arvore;
initNo( no: &arvore);
insertNo( no: &arvore, op: "9");
insertNo( no: &arvore, op: "+");
insertNo( no: &arvore, op: "1");
```

Como podemos observar na função “insertNo” a baixo, a raiz da árvore sempre é alterada quando é inserido um novo operador matemático:

```c
void insertNo(No **no,char* op){// depende da ordem com as quais as operações devem acontecer
    No *OP;
    initNo( no: &OP);
    criaNo( no: &OP, caracter: op, esq: NULL, dir: NULL);
    if(*no == NULL){
        *no = OP;
    }
    else{
        No *aux = *no;
        if(isOp( caractere: op)){
            OP->Esq = aux;
            *no = OP;
        }
        else if(isOp( caractere: (*no)->caracter)){
            if((*no)->Esq == NULL){
                (*no)->Esq = OP;
            }
            else{
                (*no)->Dir = OP;
            }
        }
    }
}
```

Executando as funcionalidades no código em “MemeAllusion”:

Entrada:

```
allude :
    valor1 receba 1 * 3 + 5
    seligaein.memear<"valor:",valor1>

:)
```

Resultados da compilação para árvore sintática abstrata:

```
Arvore sintática abstrata:
5 + 3 * 1
Resultado da conta: 8
```

Resultados da compilação para o código de 3 endereços:

```
T1: 5 + 3
T2: T1 * 1
```

Entrada 2:

```
allude :
   valor1 receba 1 * 3 + 5 * 20
   seligaein.memear<"valor:",valor1>

:)
```

Resultados da compilação para árvore sintática abstrata:

```
Arvore sintática abstrata:
20 * 5 + 3 * 1
Resultado da conta: 103
```

Resultados da compilação para o código de 3 endereços:

```
T1: 20 * 5
T2: T1 + 3
T3: T2 * 1
```

Lembrando que o que acontece no código é a leitura invertida da expressão. Dessa forma, deve ser inseridas as operações na ordem correta para que o resultado seja o desejado. Assim, observando a expressão acima, podemos notar que se o valor multiplicativo de 3 fosse 2 por exemplo, o resultado das operações ficariam incorretas.

```
Arvore sintática abstrata:
20 * 5 + 3 * 20
Resultado da conta: 2060
```

Isso acontece porque ele realiza a operação 20 * 5 = 100 soma com 3 = 103 e multiplica por 20, que resulta em 2060.

4.6 Executando o analisador semântico

Para usar o analisador semântico em conjunto do analisador sintático e léxico, basta apenas digitar o comando “make”, e em seguida digitar o arquivo de entrada que será usado, exemplo: “make”, “./a.out < arquivo.txt”. Se quiser gerar um arquivo que mostre a saída, basta digitar: “./a.out < arquivo.txt > saida.txt”.

4.7 Resultados

Nesta parte serão apresentados alguns exemplos de códigos semanticamente incorretos, demonstrando que analisador semântico abrange as principais falhas semânticas.

Programas Semanticamente incorretos:

- **Variáveis não declaradas:**

Exemplo1

```
1       celokocachoeira:
2       natashacaldeirao valor1|
3       natashacaldeirao i |
4       cazemito sapiens |
5       natashacaldeirao soma1|
6       natashacaldeirao a|
7       natashacaldeirao b|
8
9
10      querocafeee:
11         <natashacaldeirao> somar<a - natashacaldeirao, b - natashacaldeirao>:
12         soma receba a + b
13         acertoumizeravi soma
14
15      allude :
16         valor2
Erro semântico na linha 16, termo 'valor2' não foi declarado
```

Figura 21.

Exemplo 2:

```
1       celokocachoeira:
2       natashacaldeirao valor1|
3       natashacaldeirao i |
4       cazemito sapiens |
5       natashacaldeirao soma1|
6       natashacaldeirao a|
7       natashacaldeirao b|
8
9
10      querocafeee:
11         <natashacaldeirao> somar<a - natashacaldeirao, b - natashacaldeirao>:
12         soma receba a + b
13         acertoumizeravi soma
14
15      allude :
16         valor1 receba valor1
17         paraze <i receba 0 ; i mopaz 10 ; i receba i + 1 > :
18            valor1 receba valor1+i
19         <-
20         somar.memear<valor1, valor2
Erro semântico na linha 20, termo 'valor2' não foi declarado
```

Figura 22.

Exemplo 3:

```
1       celokocachoeira:
2       natashacaldeirao valor1|
3       natashacaldeirao i |
4       cazemito sapiens |
5       natashacaldeirao soma1|
6       natashacaldeirao a|
7       natashacaldeirao b|
8
9
10      querocafeee:
11         <natashacaldeirao> somar<a - natashacaldeirao, b - natashacaldeirao>:
12         soma receba a + b
13         acertoumizeravi soma
14
15      allude :
16         valor1 receba valor1                 m
Erro semântico na linha 16, termo 'm' não foi declarado
```

Figura 23.

- **Operações com tipos diferentes**

Exemplo 1:

```
1       celokocachoeira:
2       natashacaldeirao numero|
3       cazemito valor|
4       natashacaldeirao soma|
5       cazemito media|
6       natashacaldeirao cont|
7       natashacaldeirao a|
8       natashacaldeirao b|
9
10
11      querocafeee:
12         <natashacaldeirao> somar<a - natashacaldeirao, b - natashacaldeirao>:
13         soma receba a + b
14         acertoumizeravi soma
15
16      allude:
17          soma receba 0
18          cont receba 0
19          valor receba numero
20          uaiso <opaitaon>:
21              soprusmenino.memear<"%d",numero>
22              soma receba soma+soma+valor
23              cont
Erro semântico próximo a linha 22, Os tipos dos termos 'soma' e 'valor' são incompatíveis
```

Figura 24.

Exemplo 2:

```
1       celokocachoeira:
2       natashacaldeirao numero|
3       cazemito valor|
4       natashacaldeirao soma|
5       cazemito media|
6       natashacaldeirao cont|
7       natashacaldeirao a|
8       natashacaldeirao b|
9
10
11      querocafeee:
12         <natashacaldeirao> somar<a - natashacaldeirao, b - natashacaldeirao>:
13         soma receba a + b
14         acertoumizeravi soma
15
16      allude:
17          soma receba 0
18          cont receba 0
19          valor receba numero
20          uaiso <opaitaon>:
21              soprusmenino.memear<"%d",numero>
22              soma receba soma+numero
23              cont receba cont/media
24              masoqueeisso
Erro semântico próximo a linha 23, Os tipos dos termos 'cont' e 'media' são incompatíveis
```

Figura 24.

- **Atribuições para tipos diferentes**

Exemplo 1:

```
1       celokocachoeira:
2       cazemito  i|
3       cazemito menor -> 1 <- |
4       cazemito maior->2<- |
5       cazemito media->3<- |
6       cazemito altura->4<- |
7       cazemito  qtd->i<- |
8       natashacaldeirao meno1 |
9
10      querocafeee:
11
12      allude:
13
14          soprusmenino.memear<"%d",altura>
15          masoqueeisso<i dorime 0.0>:
16              menor receba menor1
Erro semântico próximo a linha 16, termo 'menor1' não foi declarado
```

Figura 25.

5. Conclusão

Neste trabalho prático foram explorados três temas de grande importância para os estudos, principalmente na disciplina de Compiladores que foi o estudo sobre a Análise Léxica e a Análise Sintática, e a análise semântica. Na ciência da computação, análise léxica, lexing ou tokenização é o processo de converter uma sequência de caracteres em uma sequência de tokens. Um programa que realiza análise lexical pode ser denominado lexer, tokenizer, ou scanner, embora scanner também seja um termo para o primeiro estágio de um lexer. Já a análise sintática é um processo de um compilador , é a segunda fase da compilação onde se analisa uma sequência que foi dada entrada para verificar sua estrutura gramatical segundo uma determinada gramática formal. Todavia também existe a parte da análise semântica no compilador, que é um processo no qual são verificados os erros semânticos no código fonte e coletadas informações necessárias para a próxima fase da compilação.

Outrossim, a primeira etapa deste trabalho foi criar a nossa própria linguagem de programação usando a ferramenta Lex, que a partir de um fluxo de caracteres os transforma em fluxo de tokens. Inicialmente, transformamos todas as palavras reservadas na linguagem C, em palavras que venham a representar a nossa linguagem, a linguagem dos memes. Dessa forma, criando um analisador léxico que reconheça a linguagem “MemeAllusion”.

Ademais, a segunda parte foi pegar o texto que foi passado na análise léxica e transformá-lo em estrutura de dados, em geral uma árvore, o que é conveniente para processamento posterior e captura a hierarquia implícita desta entrada. Através da análise léxica é obtido um grupo de tokens, para que o analisador sintático use um conjunto de regras para construir uma árvore sintática da estrutura.

Destarte, a terceira parte foi a verificação de tipos para ver se o código fonte estava semanticamente correto, se tipos iguais estavam fazendo operações entre si, e quando o analisador léxico encontra tipos diferentes sendo usados em uma mesma operação ele informa um erro semântico.

Portanto, esta abordagem mais prática da implementação de um analisador léxico, do analisador sintático e do analisador semântico, melhorou ainda mais os conhecimentos adquiridos em sala de aula, pois fez com que conceitos sobre a análise léxica, sintática, e semântica de um compilador que por sua vez foram apresentados em sala fossem utilizados para a implementação desse analisador léxico, analisador sintático, e analisador semântico, para em conjunto formarem o front-end de um compilador.

6. Referências

[1] Análise Léxica, Wikipédia, 2020. Disponível em: <Link>. Acesso em: 20 de junho de 2022.

[2] Niemann, Thomas. A Compact guide to Lex & Yacc. Disponível em: <Link>. Acesso em: 20 de junho de 2022.

[3] Análise Sintática, Wikipédia, 2022. Disponível em: <Link>. Acesso em: 15 de Julho de 2022.

a

https://pt.wikipedia.ohttps://pt.wikipedia.org/wiki/An%C3%A1lise_sem%C3%A2nticahttps://pt.wikipedia.org/wiki/An%C3%A1lise_sem%C3%A2nticarg/wiki/An%C3%A1lise_sem%C3%A2ntica