Centro Federal de Educação Tecnológica de Minas Gerais

Departamento de Computação

DISCIPLINA DE COMPILADORES

Prof. Kecia Aline Marques Ferreira

Trabalho Prática 2 - Analisador Sintático e LL1

João Victor Dias Gomes

Pedro Henrique Maia Duarte

Thales Henrique Bastos Neves

Belo Horizonte

16 de outubro de 2023

INTRODUÇÃO

Neste trabalho prático 2, abordaremos o desenvolvimento de um analisador sintático, seguindo a utilização do trabalho prático 1. O analisador sintático desempenha um papel fundamental na compreensão e verificação da estrutura de um programa, ajudando a identificar erros de sintaxe e a construir a árvore de análise necessária para a posterior geração de código. Este projeto constitui uma etapa crítica no processo de construção de um compilador, onde transformamos o código fonte em uma forma que pode ser facilmente processada. Neste contexto, exploraremos a gramática fornecida e utilizaremos os conjuntos First e Follow previamente definidos para guiar a implementação do analisador sintático. O objetivo é criar uma ferramenta capaz de analisar programas escritos na linguagem especificada e garantir que sigam a sintaxe correta de acordo com as regras estabelecidas na gramática.

O código fonte do trabalho se encontra no github a seguir: https://github.com/DiasGomes/Compilador

Comandos de execução:

Para executar o programa basta entrar no diretório principal do programa e executar o comando abaixo. Vale destacar que para o funcionamento ideal do programa é necessário passar o nome do arquivo teste em linha de comando.

java -jar compilador.jar Testes/[nome do arquivo teste]
ou
java CompiladorTP/Compilador Testes/[nome do arquivo teste]

OBJETIVOS

Para a 2° parte tem como objetivo implementar um código que seja capaz de realizar análise sintática em um código fonte da linguagem criada.

ANALISADOR SINTÁTICO

Um analisador sintático, também conhecido como parser, é uma componente essencial em um compilador que verifica a estrutura gramatical de um programa fonte. Ele analisa a sequência de tokens gerados pelo analisador léxico e determina se ela segue as regras da gramática da linguagem de programação. O analisador sintático pode ser descrito por meio de um autômato finito determinístico (AFD) ou por uma gramática livre de contexto (GLC). A gramática é frequentemente utilizada para representar a sintaxe da linguagem, enquanto o analisador sintático implementa as regras dessa gramática para determinar a estrutura do programa. Isso permite que o compilador identifique erros sintáticos e construa uma árvore de análise que servirá de base para a geração de código intermediário ou tradução para a linguagem de máquina.

MUDANÇAS NA GRAMÁTICA

O analisador léxico empregado nesse trabalho é do tipo LL(1), ou seja, um analisador preditivo que processa a cadeia da esquerda para a direita com derivação mais à esquerda e que só precisa conhecer um token a cada passo para decidir à ação a ser tomada. Abaixo está a tabela de first e follow da gramática atual. Esta é de grande importância para entendermos que é necessário fazer mudanças em algumas regras da gramática de forma que ele fique de acordo com as regras LL(1).

Figura 1 - Tabela First Follow

	First	Follow
program	class	$
decl-list	int, string, float	"{"
decl	int, string, float	";"
ident-list	identifer	";"
type	int, string, float	identifier
body	{	$
stmt-list	indentifier, if, do, read e write	"}"
stmt	indentifier, if, do, read e write	";"
assign-stmt	identifer	";"
if-stmt	if	";"
condition	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("	")"
do-stmt	do	";"
do-sufix	while	";"
read-stmt	read	";"
write-stmt	write	";"
writable	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("	")"
expression	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("	")"
simple-expr	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("	")", ">", ">=" , "<", "<=", "!=", "==", "+", "-" e "||"
term	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("	")", "*", "/" , "&&", "+", "-" e "||"
factor-a	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("	")", "*", "/" e "&&"
factor	identifier, INT_const, FLOAT_const, String_const e "("	")", "*", "/" e "&&"
relop	">", ">=" , "<", "<=", "!=" e "=="	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("
addop	"+", "-" e "||"	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("
mulop	"*", "/" e "&&"	identifier, INT_const, FLOAT_const, String_const, "!", "-" e "("

Agora iremos explicar as mudanças que fizemos na gramática, de forma que resolva alguns problemas encontrados. Foram identificados quatro regras que não estão de acordo com as regras de uma gramática LL(1), sendo elas: "*if-stmt*", "*expression*", "*simple-expr*" e "*term*".

A primeira mudança é em relação a regra do if-stmt que apresenta duas cadeias que começam com o mesmo símbolo terminal "if". Dessa forma, é impossível distinguir qual cadeia será usada. Para resolver esse problema basta manter a parte comum a ambas as partes e criar uma nova regra. No caso, esta nova regra serve para identificar se há um else após um if.

Tabela 1 - Regra if-stmt

Antiga	if-stmt ::= if "(" condition ")" "{" stmt-list "}" | if "(" condition ")" "{" stmt-list "}" else "{" stmt-list "}"
Nova	if-stmt ::= **if** "(" condition ")" "{" stmt-list "}" else-stmt else-stm ::== **else** "{" stmt-list "}" | λ

A regra do “*expression*” também apresenta uma ambiguidade, visto que ambas as suas cadeias começam com a regra ”*simple-expr*”. Para contornar isso, reescrevemos a “*expression*” como um “*simple-expr*” seguido de uma nova regra chamada ”*expression’* ”. Esta nova regra nada mais é do que lâmbda, ou seja, vazio ou a regra “*relop*” seguida do “*expression*”. É possível perceber que a regra antiga pode ser escrita a partir da nova regra, com a diferença que a mais recente pode ser implementada como LL(1).

Tabela 1 - Regra expression

Antiga	expression ::= simple-expr | simple-expr relop simple-expr
Nova	expression ::= simple-expr expression’ expression’ ::= relop expression | λ

A regra do “*simple-expr*” apesar de começar com símbolos não terminais diferentes, ambos os símbolos apresentam os mesmos símbolos não terminais como possibilidades de início da cadeia. Isso pode ser confirmado pela tabela de first. O problema foi solucionado criando uma nova regra e trocando a ordem dos termos não terminais.

Tabela 1 - Regra simple-expr

Antiga	simple-expr ::= term | simple-expr addop term
Nova	simple-expr ::= term simple-expr’ simple-expr’ ::= addop simple-expr | λ

O mesmo que foi feito acima também se aplica para a regra “*term*” como pode ser visto na tabela abaixo.

Tabela 1 - Regra term

Antiga	term ::= factor-a | term mulop factor-a
Nova	term ::= factor-a term’ term’ ::= mulop term| λ

Gramática corrigida da linguagem:

program ::= **class** identifier [decl-list] body
decl-list ::= decl ";" { decl ";"}
decl ::= type ident-list
ident-list ::= identifier {"," identifier}
type ::= **int** | **string** | **float**
body ::= "{" stmt-list "}"
stmt-list ::= stmt ";" { stmt ";" }
stmt ::= assign-stmt | if-stmt | do-stmt
| read-stmt | write-stmt
assign-stmt ::= identifier "=" simple_expr
if-stmt ::= **if** "(" condition ")" "{" stmt-list "}" else-stmt
else-stm ::== **else** "{" stmt-list "}" | λ

condition ::= expression
do-stmt ::= **do** "{" stmt-list "}" do-suffix
do-suffix ::= **while** "(" condition ")"
read-stmt ::= **read** "(" identifier ")"
write-stmt ::= **write** "(" writable ")"
writable ::= simple-expr
expression ::= simple-expr expression’
expression’ ::= relop expression | λ
simple-expr ::= term simple-expr’
simple-expr’ ::= addop simple-expr | λ
term ::= factor-a term’
term’ ::= mulop term| λ
factor-a ::= factor | "!" factor | "-" factor
factor ::= identifier | constant | "(" expression ")"
relop ::= ">" | ">=" | "<" | "<=" | "!=" | "=="
addop ::= "+" | "-" | "||"
mulop ::= "*" | "/" | "&&”

TESTES

Os testes usados na primeira etapa do trabalho tiveram seus erros léxicos corrigidos e agora será testado o nosso analisador sintático. A saída do nosso compilador são as possíveis mensagens de erro sintático que o código pode apresentar como obteve o token X, mas esperava o token Y.

1. Teste 1

O teste 1 é um código que apresenta apenas um erro na linha onze onde após o identificador “b” apareceu uma constante inteira de valor cinco. Vale mencionar que apesar de apenas um erro de fato, o compilador acabou identificando outros, visto que esse primeiro erro acabou se alastrando pela sua análise do código. Após a correção deste erro a saída do código passa a não apresentar mais erros.

Figura 2 - Código teste 1 - Código teste 1 corrigido

```
class Teste1
int a,b,c;
float result;

{
    write("Digite o valor de a:");
    read (a);
    write("Digite o valor de c:");
    read (c);
    b = 10;
    result = (a * c)/(b 5 - 345);
    write("O Resultado e: ");
    write(result);
}
```

```
class Teste1
int a,b,c;
float result;

{
    write("Digite o valor de a:");
    read (a);
    write("Digite o valor de c:");
    read (c);
    b = 10;
    result = (a * c)/(b - 345);
    write("O Resultado e: ");
    write(result);
}
```

Figura 3 - Saída do teste 1

```
Compilando o arquivo .\Testes\teste1.txt

ERRO: Token esperado: <)>, mas obteve <INTEGER_CONST> (linha 11)
ERRO: Token esperado: <;>, mas obteve <)> (linha 11)
ERRO: Token esperado: <}>, mas obteve <;> (linha 11)
ERRO: Token esperado: <EOF>, mas obteve <WRITE> (linha 12)
========== Fim de analise! ==========
```

Figura 4 - Saída do teste 1 com código corrigido

```
Compilando o arquivo .\Testes\teste1.txt

========== Fim de analise! ==========
```

2. Teste 2

O teste 2 apresenta alguns erros, sendo o primeiro deles a não especificação do tipo das variáveis na declaração na linha quatro. Ainda nesse trecho temos uma constante inteira de valor nove antes de uma identificador e por fim uma variável nomeada como uma palavra reservada “int”. O último erro deste código está na linha treze onde foi escrito de forma incorreta a função “write” sendo reconhecida como um identificador e esperando uma atribuição ao invés de um abre parênteses.

Figura 5 - Código teste 2 - Código teste 2 corrigido

```
class Teste2
/* Teste de comentário
com mais de uma linha */
a, 9valor, b_1, b_2, int;
{
    write("Entre com o valor de a: ");
    read (a);
    b_1 = a * a;
    write("O valor de b1 e: ");
    write (b_1);
    b_2 = b + a/2 * (a + 5);
    write("O valor de b2 e: ");
    Write (b2);
}
```

```
class Teste2
/* Teste de comentário
com mais de uma linha */
int a, valor, b_1, b_2, inta;
{
    write("Entre com o valor de a: ");
    read (a);
    b_1 = a * a;
    write("O valor de b1 e: ");
    write (b_1);
    b_2 = b + a/2 * (a + 5);
    write("O valor de b2 e: ");
    write (b2);
}
```

Figura 6 - Saída do teste 2

```
Compilando o arquivo .\Testes\teste2.txt

ERRO: Token esperado: <INT, STRING ou FLOAT>, mas obteve <ID> (linha 4)
ERRO: Token esperado: <ID>, mas obteve <INTEGER_CONST> (linha 4)
ERRO: Token esperado: <;>, mas obteve <ID> (linha 4)
ERRO: Token esperado: <{>, mas obteve <,> (linha 4)
ERRO: Token esperado: <=>, mas obteve <,> (linha 4)
ERRO: Token esperado: <;>, mas obteve <,> (linha 4)
ERRO: Token esperado: <}>, mas obteve <INT> (linha 4)
ERRO: Token esperado: <EOF>, mas obteve <;> (linha 4)
============ Fim de analise! ==========
```

3. Teste 3

O teste 3 contém três erros. O primeiro erro é na primeira linha com a escrita incorreta da palavra “class”, já o segundo erro é na linha doze com a escrita incorreta da palavra reservada “if”. Por fim, o último erro é na linha 28 onde se esqueceu do ponto e vírgula após o if-stmt.

Figura 7 - Código teste 3 - Código teste 3 corrigido

```
classe Teste3
/** Verificando fluxo de controle
Programa com if e while aninhados **/

int i;
int media, soma;

{
    soma = 0;
    write("Quantos dados deseja informar?" );
    read (qtd);
    IF (qtd>=2){
        i=0;
        do{
            write("Altura: ");
            read (altura);
            soma = soma+altura;
            i = i + 1;
        }while( i < qtd);

        media = soma / qtd;
        write("Media: ");
        write (media);
    }
    else{
        write("Quantidade inválida.");
    }
}
```

```
class Teste3
/** Verificando fluxo de controle
Programa com if e while aninhados **/

int i;
int media, soma;

{
    soma = 0;
    write("Quantos dados deseja informar?" );
    read (qtd);
    if (qtd>=2){
        i=0;
        do{
            write("Altura: ");
            read (altura);
            soma = soma+altura;
            i = i + 1;
        }while( i < qtd);

        media = soma / qtd;
        write("Media: ");
        write (media);
    }
    else{
        write("Quantidade inválida.");
    };
}
```

Figura 8 - Saída do teste 3

```
Compilando o arquivo .\Testes\teste3.txt

ERRO: Token esperado: <CLASS>, mas obteve <ID> (linha 1)
ERRO: Token esperado: <=>, mas obteve <(> (linha 12)
ERRO: Token esperado: <;>, mas obteve <>=> (linha 12)
ERRO: Token esperado: <}>, mas obteve <INTEGER_CONST> (linha 12)
ERRO: Token esperado: <EOF>, mas obteve <)> (linha 12)
========== Fim de analise! ==========
```

Figura 9 - Saída do teste 3 com “class” e “if” corrigidos

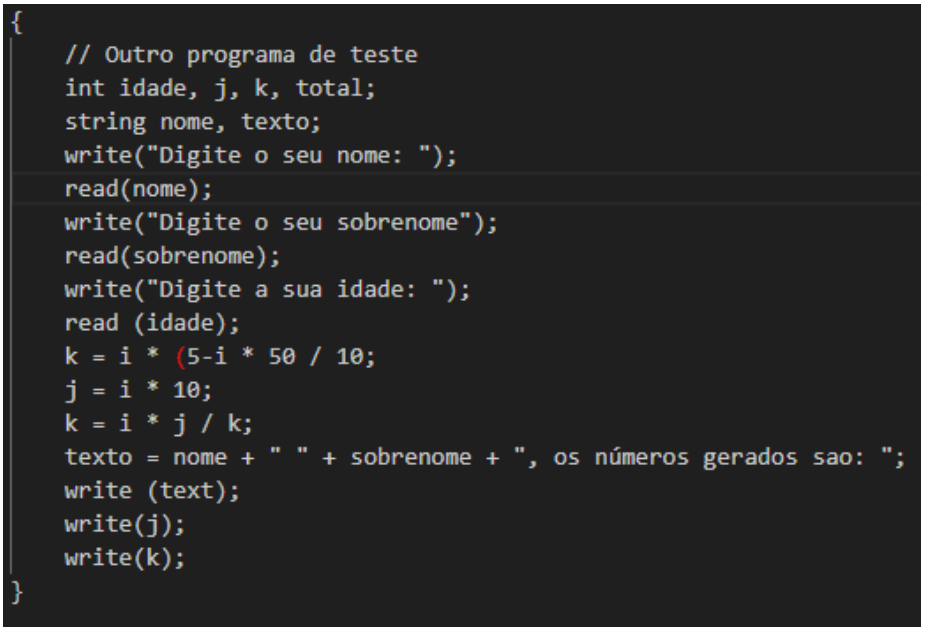
```
Compilando o arquivo .\Testes\teste3.txt

ERRO: Token esperado: <;>, mas obteve <}> (linha 28)
ERRO: Token esperado: <}>, mas obteve <EOF> (linha 28)
========== Fim de analise! ==========
```

4. Teste 4

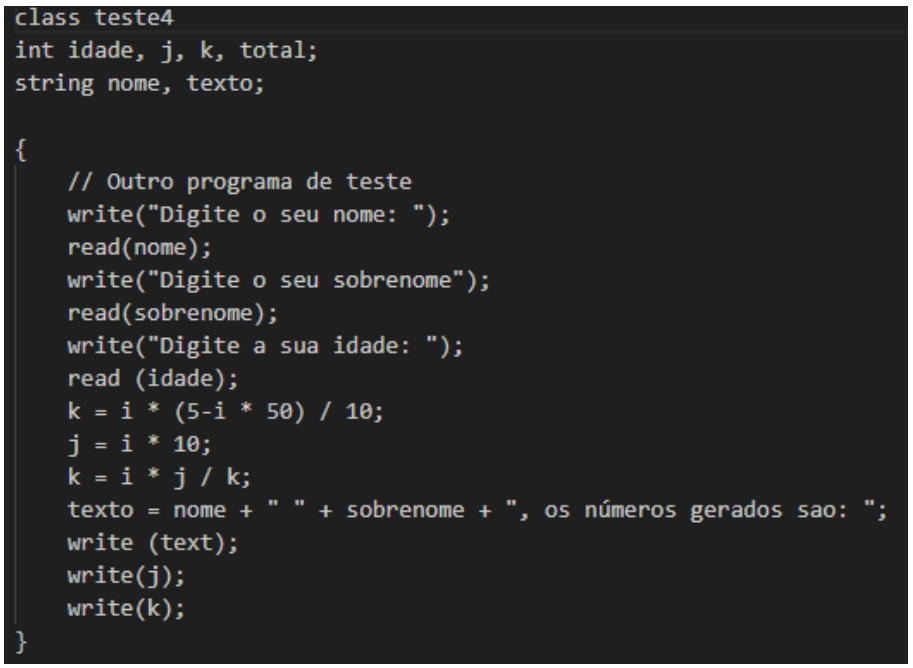
O código do teste 4 apresenta alguns erros no início como não apresentar o “class” e seu identificador, além das declarações de variáveis terem sido feitas dentro do body e não antes dele. Outro erro no código foi na linha 13 em que se esqueceu de fechar o parênteses.

Figura 10 - Código teste 4



```
{
    // Outro programa de teste
    int idade, j, k, total;
    string nome, texto;
    write("Digite o seu nome: ");
    read(nome);
    write("Digite o seu sobrenome");
    read(sobrenome);
    write("Digite a sua idade: ");
    read (idade);
    k = i * (5-i * 50 / 10;
    j = i * 10;
    k = i * j / k;
    texto = nome + " " + sobrenome + ", os números gerados sao: ";
    write (text);
    write(j);
    write(k);
}
```

Figura 11 - Código teste 4 corrigido



```
class teste4
int idade, j, k, total;
string nome, texto;

{
    // Outro programa de teste
    write("Digite o seu nome: ");
    read(nome);
    write("Digite o seu sobrenome");
    read(sobrenome);
    write("Digite a sua idade: ");
    read (idade);
    k = i * (5-i * 50) / 10;
    j = i * 10;
    k = i * j / k;
    texto = nome + " " + sobrenome + ", os números gerados sao: ";
    write (text);
    write(j);
    write(k);
}
```

Figura 12 - Saída do teste 4

```
Compilando o arquivo .\Testes\teste4.txt

ERRO: Token esperado: <CLASS>, mas obteve <{> (linha 1)
ERRO: Token esperado: <ID>, mas obteve <INT> (linha 3)
ERRO: Token esperado: <INT, STRING ou FLOAT>, mas obteve <ID> (linha 3)
ERRO: Token esperado: <{>, mas obteve <WRITE> (linha 5)
ERRO: Token esperado: <ID, IF, DO, READ ou WRITE>, mas obteve <(> (linha 5)
ERRO: Token esperado: <;>, mas obteve <(> (linha 5)
ERRO: Token esperado: <}>, mas obteve <STRING_CONST> (linha 5)
ERRO: Token esperado: <EOF>, mas obteve <)> (linha 5)
============ Fim de analise! ==========
```

Figura 13 - Saída do teste 4 após primeira correção

```
Compilando o arquivo .\Testes\teste4.txt

ERRO: Token esperado: <)>, mas obteve <;> (linha 13)
ERRO: Token esperado: <;>, mas obteve <ID> (linha 14)
ERRO: Token esperado: <}>, mas obteve <=> (linha 14)
ERRO: Token esperado: <EOF>, mas obteve <ID> (linha 14)
============ Fim de analise! ==========
```

5. Teste 5

O teste 5 apresenta em um primeiro momento dois erros, sendo eles: declaração sendo feita dentro do body na linha três e o parênteses não fechado na expressão read na linha dez. Após essas correções o compilador irá retornar outros erros em razão do não uso de abre e fecha chaves para as expressões condicionais de “if” e “else”. Além disso, também há a questão do ponto e vírgula após essas expressões.

Figura 14 - Código teste 5 - Código teste 5 corrigido

```
class MinhaClasse
{
    float a, b, c;
    write("Digite um número");
    read(a);
    write("Digite outro número: ");
    read(b);
    write("Digite mais um número: ");
    read(c;
    maior = 0;
    if ( a>b && a>c )
        maior = a;
    else
        if (b>c)
            maior = b;
        else
            maior = c;
    write("O maior número é: ");
    write(maior);
```

```
class MinhaClasse
float a, b, c;

{
    write("Digite um número");
    read(a);
    write("Digite outro número: ");
    read(b);
    write("Digite mais um número: ");
    read(c);
    maior = 0;
    if ( a>b && a>c ){
        maior = a;
    }else{
        if (b>c){
            maior = b;
        }
        else{
            maior = c;
        };
    };
    write("O maior número é: ");
    write(maior);
}
```

Figura 15 - Saída do teste 5

```
Compilando o arquivo .\Testes\teste5.txt

ERRO: Token esperado: <INT, STRING ou FLOAT>, mas obteve <{> (linha 2)
ERRO: Token esperado: <ID>, mas obteve <{> (linha 2)
ERRO: Token esperado: <;>, mas obteve <FLOAT> (linha 3)
ERRO: Token esperado: <{>, mas obteve <ID> (linha 3)
ERRO: Token esperado: <ID, IF, DO, READ ou WRITE>, mas obteve <,> (linha 3)
ERRO: Token esperado: <;>, mas obteve <,> (linha 3)
ERRO: Token esperado: <=>, mas obteve <,> (linha 3)
ERRO: Token esperado: <)>, mas obteve <;> (linha 9)
ERRO: Token esperado: <;>, mas obteve <ID> (linha 10)
ERRO: Token esperado: <}>, mas obteve <=> (linha 10)
ERRO: Token esperado: <EOF>, mas obteve <INTEGER_CONST> (linha 10)
============ Fim de analise! ==========
```

Figura 16 - Segunda saída do teste 5

```
Compilando o arquivo .\Testes\teste5.txt

ERRO: Token esperado: <{>, mas obteve <ID> (linha 12)
ERRO: Token esperado: <ID, IF, DO, READ ou WRITE>, mas obteve <=> (linha 12)
ERRO: Token esperado: <;>, mas obteve <=> (linha 12)
ERRO: Token esperado: <=>, mas obteve <;> (linha 12)
ERRO: Token esperado: <ID, CONSTANT ou '('>, mas obteve <ELSE> (linha 13)
ERRO: Token esperado: <;>, mas obteve <ELSE> (linha 13)
ERRO: Token esperado: <{>, mas obteve <ID> (linha 15)
ERRO: Token esperado: <ID, IF, DO, READ ou WRITE>, mas obteve <=> (linha 15)
ERRO: Token esperado: <;>, mas obteve <=> (linha 15)
ERRO: Token esperado: <=>, mas obteve <;> (linha 15)
ERRO: Token esperado: <ID, CONSTANT ou '('>, mas obteve <ELSE> (linha 16)
ERRO: Token esperado: <;>, mas obteve <ELSE> (linha 16)
ERRO: Token esperado: <}>, mas obteve <EOF> (linha 19)
ERRO: Token esperado: <;>, mas obteve <EOF> (linha 19)
ERRO: Token esperado: <}>, mas obteve <EOF> (linha 19)
ERRO: Token esperado: <;>, mas obteve <EOF> (linha 19)
ERRO: Token esperado: <}>, mas obteve <EOF> (linha 19)
============ Fim de analise! ==========
```

6. Teste 6

O teste 6 foi um teste criado pelo próprio grupo. Este apresenta somente um erro na linha 8 em que houve um caractere inserido de forma incorreta em uma atribuição de uma variável.

Figura 17 - Código teste 6

```
class TesteFloat
float a,b,c, result;

{
    write("Digite o valor de a:");
    read (a);
    b = 10.234;
    c = 0.25c;
    result = (a * b * c);
    write("O Resultado e: ");
    write(result);
}
```

```
class TesteFloat
float a,b,c, result;

{
    write("Digite o valor de a:");
    read (a);
    b = 10.234;
    c = 0.25;
    result = (a * b * c);
    write("O Resultado e: ");
    write(result);
}
```

Figura 18 - Saída do teste 6

```
Compilando o arquivo .\Testes\meuTeste6.txt

ERRO: Token esperado: <;>, mas obteve <ID> (linha 8)
ERRO: Token esperado: <}>, mas obteve <;> (linha 8)
ERRO: Token esperado: <EOF>, mas obteve <ID> (linha 9)
============ Fim de analise! ==========
```

7. Teste 7

Por fim, o último teste, é um código sem nenhum tipo de erro sintático. Ele serviu para mostrar o funcionamento do compilador em um caso ideal.

Figura 19 - Código teste 7

```
// testa comentarios
// de uma linha seguidos

class TesteDuploComentario
int a, result;

{
    write("Digite o valor de a:");
    read (a);
    result = (a * 2);
    write("O Resultado e: ");
    write(result);
}
```

Figura 20 - Saída do teste 7

```
Compilando o arquivo .\Testes\meuTeste7.txt

========== Fim de analise! ==========
```