CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO

ANA CLARA CUNHA LOPES
GUSTAVO DE ASSIS XAVIER

Compiladores

Trabalho parte 2 - Analisador Sintático

Belo Horizonte - MG
27 de junho de 2025

SUMÁRIO

Observação: As partes destacadas em vermelho são as referentes à etapa dois do trabalho.

1. INTRODUÇÃO

Este relatório apresenta o desenvolvimento de um analisador léxico e sintático de um compilador para uma linguagem de programação específica, implementado como parte de um trabalho acadêmico. O objetivo é descrever como o analisador sintático funciona, a abordagem utilizada na implementação, destacando as principais classes e seus propósitos, e os resultados dos testes realizados, incluindo os programas-fontes analisados e as saídas do compilador.

2. ANALISADOR LÉXICO

2.1. FORMA DE USO DO COMPILADOR

O analisador sintático do compilador foi desenvolvido em Java e executado em um ambiente de desenvolvimento integrado (VS-Code). A seguir, descrevemos os passos para executá-lo:

1. **Preparação do programa-fonte:** o usuário deve criar um arquivo de texto (testeX.txt) contendo o código-fonte da linguagem especificada. Este arquivo pode ser salvo na pasta testes, junto aos demais arquivos.

2. **Compilação do projeto:** Com o terminal dentro da pasta **compilador,** faça:

   javac analisadorlexico/*.java interpretador/value/*.java *.java

3. **Execução do compilador:** após compilar, para executar o programa basta ir com o terminal para pasta COMPILADORES e executar:

   java compilador.exe

   Há o passo a passo também no README, nos arquivos.

4. **Interpretação da saída:** a saída do compilador lista os tokens identificados no formato Token(TIPO, "valor", linha=N), onde TIPO é o tipo do token (ex.: PROGRAM, IDENTIFIER), valor é o lexema identificado e N é a linha onde o lexema foi encontrado. Linhas vazias são ignoradas na contagem lógica. Se houver erro, uma mensagem acima do problema.

2.2. ABORDAGEM UTILIZADA NA IMPLEMENTAÇÃO

A implementação do compilador foi realizada em Java, focando inicialmente na etapa de análise léxica, que é responsável por transformar o código-fonte em uma sequência de tokens. A abordagem utilizada é baseada em um analisador léxico determinístico, que processa o código caractere por caractere, identificando tokens com base em regras predefinidas para a linguagem.

2.3. PRINCIPAIS CLASSES DA APLICAÇÃO

A aplicação é composta por três “classes”, cada uma com um propósito específico:

- **exe:** esta classe contém o método main, sendo o ponto de entrada da aplicação. É responsável por iniciar o compilador e coordenar a execução do processo de análise léxica. Ela instancia a classe AnalisadorLexico, passa o conteúdo do arquivo de entrada e exibe a saída gerada (tokens ou mensagens de erro) no console.

- **LexicalAnalysis:** esta classe é o núcleo do compilador, responsável pela análise léxica do código-fonte. Ela lê o arquivo de entrada, processa o texto caractere por caractere e gera uma sequência de tokens. Possui métodos para:
    - Pular espaços em branco e quebras de linha, ajustando a contagem de linhas (ignorando linhas vazias na contagem lógica).
    - Identificar palavras-chave, identificadores, constantes, literais, operadores e símbolos.
    - Gerar tokens com informações sobre tipo, valor e linha.
    - Reportar erros léxicos, como caracteres inválidos ou literais mal formados.

A classe utiliza um mapa de palavras reservadas para associar palavras-chave a tipos de tokens específicos.

- **Token:** esta classe representa um token gerado pelo analisador léxico. Cada token possui quatro atributos: o tipo (um enum que define categorias como PROGRAM, IDENTIFIER, SEMICOLON, etc.), o valor (que é representado por uma determinada classe derivada da classe Value, pois isso ajudará nas próximas implementações) o lexema (o “texto” correspondente no código-fonte) e a linha onde foi encontrado. A classe é usada para estruturar a saída do analisador léxico e facilita a integração com futuras etapas do compilador (como a análise sintática).

- **Value<?> e seus filhos:** Considerando a disciplina de LP, essa classe pode nos ajudar a padronizar valores e fazer conversões, facilitando a implementação dos demais analisadores.

2.4. DETALHES DA IMPLEMENTAÇÃO

A análise léxica foi implementada utilizando uma abordagem baseada em máquina de estados implícita, onde o método nextToken() da classe LexicalAnalysis lê caracteres consecutivamente e decide qual token gerar com base em padrões predefinidos:

- Palavras-chave e identificadores são identificados verificando se o caractere inicial é uma letra ou underline, seguido por letras, dígitos ou underline.
- Constantes numéricas são processadas verificando dígitos e, no caso de números reais, a presença de um ponto decimal seguido por mais dígitos.
- Constantes de caracteres e literais são identificados por aspas simples e duplas, respectivamente.
- Símbolos e operadores (como: +, *, - , etc.) são tratados por um switch que verifica o caractere atual e, em alguns casos, o próximo caractere, como por exemplo: ==, >=, etc.).

Os casos no geral são dados pelo diagrama (feito do Draw.io):

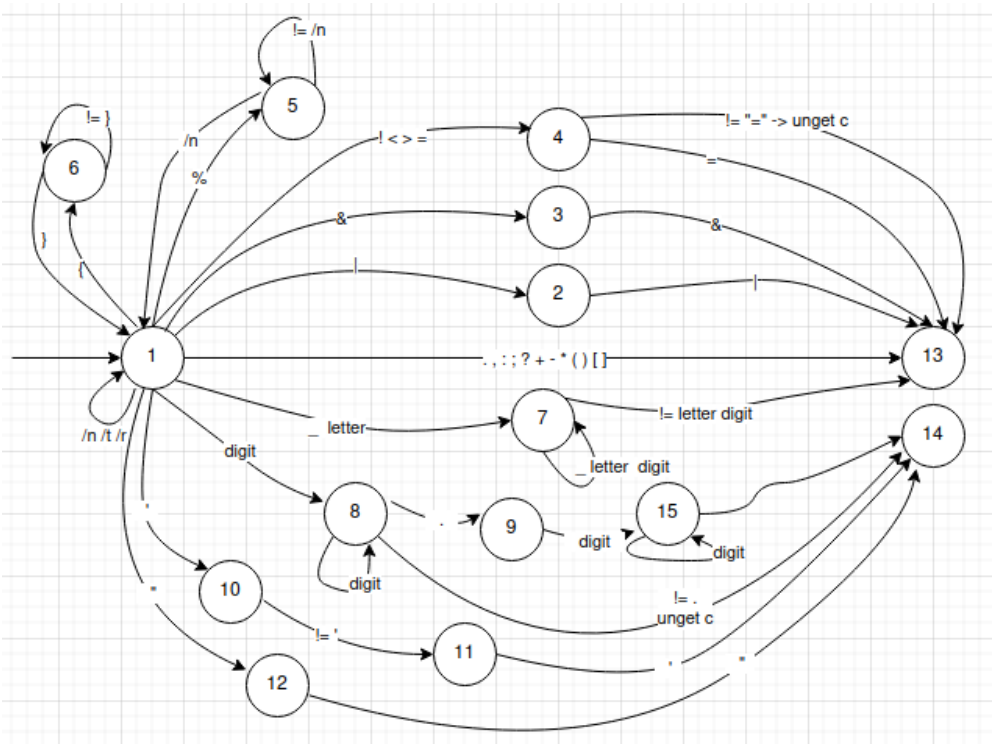

Diagrama de Estados do Analisador Léxico.

2.5. RESULTADOS DOS TESTES

Foram realizados testes com diferentes programas-fonte para verificar o funcionamento do compilador. Todos os testes que deram erro foram corrigidos adequando esses testes para a análise sintática. Um teste pequeno é colocado abaixo para exemplificar o funcionamento do analisador léxico:

```
program

     float: raio, area$ = 0.0;

begin
repeat
   in(raio);
   char: resposta;
   if (raio > 0.0) then
     area =  3. * raio * raio;
     out (area);
   end;
   out ("Deseja continuar?);
   in (resp);

until (resp == 'N' || resp == 'n');

end
```

```
Analisando: teste2.txt
Token: PROGRAM      Lexema: program  Linha: 1
Token: FLOAT_CONST  Lexema: float    Linha: 2
Token: COLON        Lexema: :        Linha: 2
Token: IDENTIFIER   Lexema: raio     Linha: 2
Token: COMMA        Lexema: ,        Linha: 2
Token: IDENTIFIER   Lexema: area     Linha: 2
Erro ao analisar léxico: Na linha 2->Caracter não permitido.
```

3. ANALISADOR SINTÁTICO

3.1. GRAMÁTICA

Para a implementação da linguagem, algumas alterações e definições na gramática foram feitas, são elas:

program ::= program [decl-list] begin stmt-list end **“;”**

decl-list ::= decl {decl}

decl ::= type “:” ident-list **“;”**

ident-list ::= identifier {"," identifier}

type ::= int | float | char

stmt-list ::= stmt {stmt}

stmt ::= assign-stmt **“;”** | if-stmt **“;”** | while-stmt **“;”** | repeat-stmt **“;”** | read-stmt **“;”** | write-stmt **“;”**

assign-stmt ::= identifier "=" simple_expr

if-stmt ::= if condition then [decl-list] stmt-list **[else declaration stmt- list]** end

while-stmt ::= stmt-prefix [decl-list] stmt-list end

stmt-prefix ::= while condition do

repeat-stmt ::= repeat [decl-list] stmt-list stmt-suffix

stmt-suffix ::= until condition

read-stmt ::= in "(" identifier ")"

write-stmt ::= out "(" writable ")"

writable ::= simple-expr | literal

condition ::= expression

expression ::= simple-expr [relop simple-expr]

simple-expr ::= term {addop term}

term ::= factor-a {mulop factor-a}

fator-a ::= = ["!" | "-"] factor

factor ::= identifier | constant | "(" expression ")"

relop ::= "==" | ">" | ">=" | "<" | "<=" | "!="

addop ::= "+" | "-" | ||

mulop ::= "*" | "/" | &&

constant ::= integer_const | float_const | char_const

Com isso, nossa linguagem traz alguns destaques, são eles:

- Todos os comandos terminam com “;”. Sendo que por comando se entende toda a estrutura do comando. Por exemplo, o comando if é dado por:

  if condition **then** [decl-list] stmt-list **end**

  Sendo assim, após o **end** haverá um “;”:

  if condition **then** [decl-list] stmt-list **end;**

  No formato de quebra de linha:

  if condition **then**
  [decl-list]
  stmt-list
  end;

  Isso inclui o último “end” do programa , por motivos de padronização.

- Em cada bloco, todas declarações devem ser sempre feitas antes dos demais comandos, ou seja, podemos ter:

  if condition **then**
  int : a, b;
  a = 2;
  b = 2;
  end;

  Mas não podemos ter:

  if condition **then**
  int : a;
  a = 2;
  int : b;
  b = 2;
  end;

- Comparações aninhadas devem ser organizadas com “( )”. Por exemplo, a seguinte comparação gera erros:

  if (a > b && a > 2)

  Ela deve ser organizados de forma:

  if ((a > b) && (a > 2))

3.2. IMPLEMENTAÇÃO

Para a implementação da gramática foi usado um parser LL(1). Por isso, as ambiguidades e recursão à esquerda foram corrigidas. Por exemplo, a representação correta do if seria:

if-stmt ::= if condition then [decl-list] stmt-list [else-part] end |

else-part ::= else [decl-list] stmt-list

Que pode ser simplificada por:

if-stmt ::= if condition then [decl-list] stmt-list [else declaration stmt- list] end

Já com relação a recursão a esquerda, temos que uma regra como:

simple-expr ::= term | simple-expr addop term

Se torna:

simple-expr ::= term {addop term}

Através da transformação dada por:

A ::= A α | β

Onde β = termo e A = simple-expr e α = addop term, que inserindo uma variável de "ajuda" se torna:

A ::= β A'

A' ::= α A' | ε

Que se trata de β seguido por um número indeterminado de α. Isso pode ser simplificada por:

A ::= β {α}

Da mesma forma os demais casos.

3.3. INSTRUÇÕES DE USO DO COMPILADOR SINTÁTICO

O analisador sintático do compilador foi desenvolvido em Java e executado em um ambiente de desenvolvimento integrado (VS-Code). A seguir, descrevemos os passos para executá-lo:

- **Preparação do programa-fonte:** o usuário deve criar um arquivo de texto (testeX.txt) contendo o código-fonte da linguagem especificada. Este arquivo pode ser salvo na pasta testes, junto aos demais arquivos.

- **Compilação do projeto:** Com o terminal dentro da pasta **compiler (**que possui o README**),** faça:

  javac -d . $(find . -name "*.java")

- **Execução do compilador:** após compilar, para executar o programa, ainda no mesmo terminal, digite:

  java compiler.exe

  Há o passo a passo também no README, nos arquivos.

- **Interpretação da saída:** Na saída, cada "found (token)" encontrado significa que esse token estava sendo esperado, e foi devidamente encontrado, caso não fosse esperado, é gerado um erro. Nos testes esse comportamento é melhor explicado.

3.4. TESTES

3.4.1. TESTE 1

No input, destacamos em vermelho os erros e mudanças feitas no código para mostrar as correções feitas para serem aceitos no analisador sintático. No output, imprimimos a saída da análise léxica.

INPUT (erros declarados em vermelho):

INPUT ANTES DA CORREÇÃO.	INPUT CORRIGIDO:

program
int: a,b,c;
float: result;
char: ch;
begin
out("Digite o valor de a:");
in (a);
out("Digite o valor de c:");
read (ch);
b = 10;
result = (a * ch)/**(b 5 - 345.27)**;
out("O resultado e: ");
out(result);
result = result + ch;
end
	program
	int: a,b,c;
	float: result;
	char: ch;
	begin
	out("Digite o valor de a:");
	in (a);
	out("Digite o valor de c:");
	in(ch);
	b = 10;
	result = (a * ch)/**(b - 345.27)**;
	out("O resultado e: ");
	out(result);
	result = result + ch;
	end**;**

OUTPUT do código sem correção:

```
Analisando: teste1.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 09: Esperado 'END', encontrado 'READ'.  / Token atual: read (linha: 9).
[RESULTADO] Analise sintatica falhou.
```

O erro dado é devido a utilização da palavra “read”, que não é uma palavra pertencente à gramática, com isso, o analisador pensa que havia acabado o stmt-list e tenta consumir “end”, mas obtém “read”. A correta utilização seria “in(ch);”. Ajustando o código obtemos:

```
Analisando: teste1.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 10: Esperado 'CLOSE_PAR', encontrado 'INTEGER_CONST'.  / Token atual: 5 (linha: 10).
[RESULTADO] Analise sintatica falhou.
```

Esse erro se dá pela expressão mal formada (b 5 - 345.27). Corrigindo ela para (b - 345.27), obtemos:

```
Analisando: teste1.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 14: Fim de arquivo inesperado.  / Token atual:  (linha: 14).
[RESULTADO] Analise sintatica falhou.
```

Esse erro se dá, porque na atualização da gramática que fizemos, precisa ter um “;” depois do end, corrigindo essa parte, obtemos:

```
Analisando: teste1.txt
[INFO] Analise sintatica iniciada.
[INFO] Analise sintatica finalizada.
[RESULTADO] Analise sintatica realizada com sucesso.
```

Como podemos ver, a análise foi concluída com sucesso (sem erros).

3.4.2. TESTE 2

INPUT:

Correções: declaração com comando de atribuição, declaração não vindo antes dos demais comandos em um bloco, e comparações sem a utilização de “( )”.

INPUT ANTES DA CORREÇÃO.	INPUT CORRIGIDO:

program
float: raio, area **= 0.0**;
begin
repeat
in(raio);
char: resposta;
if (raio > 0.0) then
	program
	float: raio, **area;**
	begin
	repeat
	char: resposta;
	in(raio);
	if (raio > 0.0) then

area = 3.0 * raio * raio;
out (area);
end;
out ("Deseja continuar?");
in (resp);
until (resp == 'N' || resp == 'n');
end
	area = 3.0 * raio * raio;
	out (area);
	end;
	out ("Deseja continuar?");
	in (resp);
	until ((resp == 'N') || (resp == 'n'));
	end**;**

OUTPUT:

```
Analisando: teste2.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 02: Esperado 'SEMICOLON', encontrado 'ASSIGN'.  / Token atual: = (linha: 2).
[RESULTADO] Analise sintatica falhou.
```

Esse erro se deve a atribuição de valor a variável “area” de forma indevida, pois nossa gramática não permite que atribua valor nessa parte do programa. Corrigindo, obtemos:

```
Analisando: teste2.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 06: Esperado 'UNTIL', encontrado 'CHAR'.  / Token atual: char (linha: 6).
[RESULTADO] Analise sintatica falhou.
```

Esse erro se deve a declaração de variável “char:resposta;” nessa etapa do código. Nossa linguagem não permite que variáveis sejam declaradas nessa parte do fluxo, logo, o analisador sintático identificou esse problema e seguindo o fluxo de execução do “repeat-stmt”, ele esperava pelo “until”. Corrigindo obtemos:

```
Analisando: teste2.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 13: Esperado 'CLOSE_PAR', encontrado 'EQUALS'.  / Token atual: == (linha: 13).
[RESULTADO] Analise sintatica falhou.
```

Esse erro é devido a expressão “until (resp == 'N' || resp == 'n');”, precisar conter parênteses em cada parte da operação, ou seja, “until ((resp == 'N') ||(resp == 'n'));”, sendo assim, obtemos:

```
Analisando: teste2.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 14: Fim de arquivo inesperado.  / Token atual:  (linha: 14).
[RESULTADO] Analise sintatica falhou.
```

Esse erro é devido a falta de “;” no final do código. Ajustando, obtemos:

```
Analisando: teste2.txt
[INFO] Analise sintatica iniciada.
[INFO] Analise sintatica finalizada.
[RESULTADO] Analise sintatica realizada com sucesso.
```

Com isso, obtemos um código sem erros.

3.4.3. TESTE 3

INPUT:

INPUT ANTES DA CORREÇÃO.	INPUT CORRIGIDO:

program
int: a, b, aux;
begin
in (a);
in(b);
if (a>b) then
int aux;
aux = b;
b = a;
a = aux
end;
out(a;
out(b)
end;
	program
	int: a, b, aux;
	begin
	in (a);
	in(b);
	if (a>b) then
	int: aux;
	aux = b;
	b = a;
	a = aux;
	end;
	out(a);
	out(b);
	end**;**

OUTPUT:

```
Analisando: teste3.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 07: Esperado 'COLON', encontrado 'IDENTIFIER'.  / Token atual: aux (linha: 7).
[RESULTADO] Analise sintatica falhou.
```

Esse erro é devido a falta de “:” em “int aux;”. Corrigindo obtemos:

```
Analisando: teste3.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 11: Esperado 'SEMICOLON', encontrado 'END'.  / Token atual: end (linha: 11).
[RESULTADO] Analise sintatica falhou.
```

Esse erro é devido a falta de “;” na expressão “a = aux”, ajustando obtemos:

```
Analisando: teste3.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 12: Esperado 'CLOSE_PAR', encontrado 'SEMICOLON'.  / Token atual: ; (linha: 12).
[RESULTADO] Analise sintatica falhou.
```

Esse erro é devido a falta de “)” na linha de código “out(a;”, ajustando obtemos:

```
Analisando: teste3.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 14: Esperado 'SEMICOLON', encontrado 'END'.  / Token atual: end (linha: 14).
[RESULTADO] Analise sintatica falhou.
```

Esse erro é devido a falta de “;” na linha de código: “out(b)”, corrigindo obtemos:

```
Analisando: teste2.txt
[INFO] Analise sintatica iniciada.
[INFO] Analise sintatica finalizada.
[RESULTADO] Analise sintatica realizada com sucesso.
```

Como podemos ver, o código foi compilado com sucesso.

3.4.4. TESTE 4

Nestes próximos testes, mostraremos como cada token é recebido.

INPUT:

Correções: Declaração errada. if sem stmt-list, usando “and” e sem “then”. Falta de vários “;”.

INPUT ANTES DA CORREÇÃO.

```
program
        a, b, c, maior, outro: int;
begin
        repeat
                out("A");
                in(a);
                out("B");
                in(b);
                out("C");
                in(c);
                %Verifica o maior
                if ( (a>b) and (a>c) ) end
                        maior = a
                else
                        if (b>c) then
                                maior = b;
                            else
                                maior = c
                        end;
                end;
                out("Maior valor:");
                out (maior);
                out ("Outro ");
                in(outro);
        until (outro == 0)
end;
```

INPUT CORRIGIDO:

```
program
        int: a, b, c, maior, outro;
begin
        repeat
                out("A");
                in(a);
                out("B");
                in(b);
                out("C");
                in(c);
                %Verifica o maior
                if ( (a>b) && (a>c) ) then
                  in(c);
                else
                        maior = a;
                        if (b>c) then
                                maior = b;
                            else
                                maior = c;
                        end;
                end;
                out("Maior valor:");
                out (maior);
                out ("Outro ");
                in(outro);
        until (outro == 0);
end;
```

OUTPUT:

3.4.5. TESTE 5

INPUT:

Correções: Diretiva “declare” e “programa” inexistentes. int declarado como “inteiro”. Sem end no final. “;” faltantes.

INPUT ANTES DA CORREÇÃO.

```
programa
        declare
        inteiro: pontuacao, pontuacaoMaxina,
disponibilidade;
        char: pontuacaoMinima
begin
        disponibilidade = 'S';
        pontuacaoMinima = 50;
        pontuacaoMaxima = 100;
        out("Pontuacao Candidato: ");
        in(pontuacao);
        out("Disponibilidade Candidato: ");
        in(disponibilidade);

{ Comentario
grande
while (pontuacao>0 &&
(pontuação<=pontuacaoMaxima) do
int: cont;
cont = cont + 1;
if ((pontuação > pontuacaoMinima) &&
(disponibilidade==1)) then
out("Candidato aprovado")
else
out("Candidato reprovado")
end
out("Pontuacao Candidato: ");
in(pontuacao);
out("Disponibilidade
Candidato: ");
in(disponibilidade);
end
out (cont);
end}
```

INPUT CORRIGIDO:

```
program
        int: pontuacao, pontuacaoMaxina,
disponibilidade;
        char: pontuacaoMinima;
begin
        disponibilidade = 'S';
        pontuacaoMinima = 50;
        pontuacaoMaxima = 100;
        out("Pontuacao Candidato: ");
        in(pontuacao);
        out("Disponibilidade Candidato: ");
        in(disponibilidade);

{ Comentario
grande
while (pontuacao>0 &&
(pontuação<=pontuacaoMaxima) do
int: cont;
cont = cont + 1;
if ((pontuação > pontuacaoMinima) &&
(disponibilidade==1)) then
out("Candidato aprovado")
else
out("Candidato reprovado")
end
out("Pontuacao Candidato: ");
in(pontuacao);
out("Disponibilidade
Candidato: ");
in(disponibilidade);
end
out (cont);
end}
end
```

OUTPUT:

```
Found ("program", PROGRAM, 1, null)
Found ("int", INT, 2, null)
Found (":", COLON, 2, null)
Found ("pontuacao", IDENTIFIER, 2, null)
Found (",", COMMA, 2, null)
Found ("pontuacaoMaxina", IDENTIFIER, 2, null)
Found ("out", OUT, 8, null)
Found ("(", OPEN_PAR, 8, null)
Found ("Pontuacao Candidato: ", LITERAL, 8,
Pontuacao Candidato: )
Found (")", CLOSE_PAR, 8, null)
Found (";", SEMICOLON, 8, null)
```

```
Found (",", COMMA, 2, null)
Found ("disponibilidade", IDENTIFIER, 2, null)
Found (";", SEMICOLON, 2, null)
Found ("char", CHAR, 3, null)
Found (":", COLON, 3, null)
Found ("pontuacaoMinima", IDENTIFIER, 3, null)
Found (";", SEMICOLON, 3, null)
Found ("begin", BEGIN, 4, null)
Found ("disponibilidade", IDENTIFIER, 5, null)
Found ("=", ASSIGN, 5, null)
Found ("S", CHAR_CONST, 5, S)
Found (";", SEMICOLON, 5, null)
Found ("pontuacaoMinima", IDENTIFIER, 6, null)
Found ("=", ASSIGN, 6, null)
Found ("50", INTEGER_CONST, 6, 50)
Found (";", SEMICOLON, 6, null)
Found ("pontuacaoMaxima", IDENTIFIER, 7, null)
Found ("=", ASSIGN, 7, null)
Found ("100", INTEGER_CONST, 7, 100)
Found (";", SEMICOLON, 7, null)
```

```
Found ("in", IN, 9, null)
Found ("(", OPEN_PAR, 9, null)
Found ("pontuacao", IDENTIFIER, 9, null)
Found (")", CLOSE_PAR, 9, null)
Found (";", SEMICOLON, 9, null)
Found ("out", OUT, 10, null)
Found ("(", OPEN_PAR, 10, null)
Found ("Disponibilidade Candidato: ", LITERAL,
10, Disponibilidade Candidato: )
Found (")", CLOSE_PAR, 10, null)
Found (";", SEMICOLON, 10, null)
Found ("in", IN, 11, null)
Found ("(", OPEN_PAR, 11, null)
Found ("disponibilidade", IDENTIFIER, 11, null)
Found (")", CLOSE_PAR, 11, null)
Found (";", SEMICOLON, 11, null)
Found ("end", END, 31, null)
Found (";", SEMICOLON, 31, null)
Found ("", END_OF_FILE, 32, null)
```

3.4.6. TESTE 6

INPUT:

Correções: Símbolos não pertencentes à gramática “:=”, comparações sem o uso de “( )” e “;” no final

INPUT ANTES DA CORREÇÃO.

```
program
        int: x, y;
        float: media;
        char: opcao;
% Teste de comentario de linha inicial

begin
{ Teste de comentario em bloco
sobre o programa
}
        out("Digite o valor de x: ");
        in(x);
        out("Digite o valor de y: ");
        in(y);

        media := (x + y) / 2.0;
        out("Media dos valores: ");
        out(media);

        if (media >= 5.0) then
                out("Media suficiente!");
        else
                out("Media insuficiente!");
        end;

        out("Digite a opcao (S/N): ");
        in(opcao);
```

INPUT CORRIGIDO:

```
program
        int: x, y;
        float: media;
        char: opcao;
% Teste de comentario de linha inicial

begin
{ Teste de comentario em bloco
sobre o programa
}
        out("Digite o valor de x: ");
        in(x);
        out("Digite o valor de y: ");
        in(y);

        media = (x + y) / 2.0;
        out("Media dos valores: ");
        out(media);

        if (media >= 5.0) then
                out("Media suficiente!");
        else
                out("Media insuficiente!");
        end;

        out("Digite a opcao (S/N): ");
        in(opcao);
```

```
        while (opcao != 'N' && opcao != 'n') do
        out("Digite novos valores para x e y: ");
        in(x);
        in(y);
        media := (x + y) / 2.0;
        out("Nova media: ");
        out(media);
        out("Digite a opcao (S/N): ");
        in(opcao);
        end;
end;
```

```
        while ((opcao != 'N') && (opcao != 'n')) do
        out("Digite novos valores para x e y: ");
        in(x);
        in(y);
        media = (x + y) / 2.0;
        out("Nova media: ");
        out(media);
        out("Digite a opcao (S/N): ");
        in(opcao);
        end;
end;
```

OUTPUT:

```
Found ("program", PROGRAM, 1, null)
Found ("int", INT, 2, null)
Found (":", COLON, 2, null)
Found ("x", IDENTIFIER, 2, null)
Found (",", COMMA, 2, null)
Found ("y", IDENTIFIER, 2, null)
Found (";", SEMICOLON, 2, null)
Found ("float", FLOAT, 3, null)
Found (":", COLON, 3, null)
Found ("media", IDENTIFIER, 3, null)
Found (";", SEMICOLON, 3, null)
Found ("char", CHAR, 4, null)
Found (":", COLON, 4, null)
Found ("opcao", IDENTIFIER, 4, null)
Found (";", SEMICOLON, 4, null)
Found ("begin", BEGIN, 7, null)
Found ("out", OUT, 11, null)
Found ("(", OPEN_PAR, 11, null)
Found ("Digite o valor de x: ", LITERAL, 11, Digite
o valor de x: )
Found (")", CLOSE_PAR, 11, null)
Found (";", SEMICOLON, 11, null)
Found ("in", IN, 12, null)
Found ("(", OPEN_PAR, 12, null)
Found ("x", IDENTIFIER, 12, null)
Found (")", CLOSE_PAR, 12, null)
Found (";", SEMICOLON, 12, null)
Found ("out", OUT, 13, null)
Found ("(", OPEN_PAR, 13, null)
Found ("Digite o valor de y: ", LITERAL, 13, Digite
o valor de y: )
Found (")", CLOSE_PAR, 13, null)
Found (";", SEMICOLON, 13, null)
Found ("in", IN, 14, null)
Found ("(", OPEN_PAR, 14, null)
Found ("y", IDENTIFIER, 14, null)
Found (")", CLOSE_PAR, 14, null)
Found (";", SEMICOLON, 14, null)
Found ("media", IDENTIFIER, 16, null)
Found ("=", ASSIGN, 16, null)
Found ("(", OPEN_PAR, 16, null)
Found ("x", IDENTIFIER, 16, null)
Found ("+", ADD, 16, null)
```

```
Found (";", SEMICOLON, 24, null)
Found ("out", OUT, 26, null)
Found ("(", OPEN_PAR, 26, null)
Found ("Digite a opcao (S/N): ", LITERAL, 26,
Digite a opcao (S/N): )
Found (")", CLOSE_PAR, 26, null)
Found (";", SEMICOLON, 26, null)
Found ("in", IN, 27, null)
Found ("(", OPEN_PAR, 27, null)
Found ("opcao", IDENTIFIER, 27, null)
Found (")", CLOSE_PAR, 27, null)
Found (";", SEMICOLON, 27, null)
Found ("while", WHILE, 28, null)
Found ("(", OPEN_PAR, 28, null)
Found ("(", OPEN_PAR, 28, null)
Found ("opcao", IDENTIFIER, 28, null)
Found ("!=", NOT_EQUALS, 28, null)
Found ("N", CHAR_CONST, 28, N)
Found (")", CLOSE_PAR, 28, null)
Found ("&&", AND, 28, null)
Found ("(", OPEN_PAR, 28, null)
Found ("opcao", IDENTIFIER, 28, null)
Found ("!=", NOT_EQUALS, 28, null)
Found ("n", CHAR_CONST, 28, n)
Found (")", CLOSE_PAR, 28, null)
Found (")", CLOSE_PAR, 28, null)
Found ("do", DO, 28, null)
Found ("out", OUT, 29, null)
Found ("(", OPEN_PAR, 29, null)
Found ("Digite novos valores para x e y: ",
LITERAL, 29, Digite novos valores para x e y: )
Found (")", CLOSE_PAR, 29, null)
Found (";", SEMICOLON, 29, null)
Found ("in", IN, 30, null)
Found ("(", OPEN_PAR, 30, null)
Found ("x", IDENTIFIER, 30, null)
Found (")", CLOSE_PAR, 30, null)
Found (";", SEMICOLON, 30, null)
Found ("in", IN, 31, null)
Found ("(", OPEN_PAR, 31, null)
Found ("y", IDENTIFIER, 31, null)
Found (")", CLOSE_PAR, 31, null)
Found (";", SEMICOLON, 31, null)
```

Found ("y", IDENTIFIER, 16, null)
Found (")", CLOSE_PAR, 16, null)
Found ("/", DIV, 16, null)
Found ("2.0", FLOAT_CONST, 16, 2)
Found (";", SEMICOLON, 16, null)
Found ("out", OUT, 17, null)
Found ("(", OPEN_PAR, 17, null)
Found ("Media dos valores: ", LITERAL, 17, Media dos valores:)
Found (")", CLOSE_PAR, 17, null)
Found (";", SEMICOLON, 17, null)
Found ("out", OUT, 18, null)
Found ("(", OPEN_PAR, 18, null)
Found ("media", IDENTIFIER, 18, null)
Found (")", CLOSE_PAR, 18, null)
Found (";", SEMICOLON, 18, null)
Found ("if", IF, 20, null)
Found ("(", OPEN_PAR, 20, null)
Found ("media", IDENTIFIER, 20, null)
Found (">=", GREATER_EQUAL, 20, null)
Found ("5.0", FLOAT_CONST, 20, 5)
Found (")", CLOSE_PAR, 20, null)
Found ("then", THEN, 20, null)
Found ("out", OUT, 21, null)
Found ("(", OPEN_PAR, 21, null)
Found ("Media suficiente!", LITERAL, 21, Media suficiente!)
Found (")", CLOSE_PAR, 21, null)
Found (";", SEMICOLON, 21, null)
Found ("else", ELSE, 22, null)
Found ("out", OUT, 23, null)
Found ("(", OPEN_PAR, 23, null)
Found ("Media insuficiente!", LITERAL, 23, Media insuficiente!)
Found (")", CLOSE_PAR, 23, null)
Found (";", SEMICOLON, 23, null)
Found ("end", END, 24, null)

Found ("media", IDENTIFIER, 32, null)
Found ("=", ASSIGN, 32, null)
Found ("(", OPEN_PAR, 32, null)
Found ("x", IDENTIFIER, 32, null)
Found ("+", ADD, 32, null)
Found ("y", IDENTIFIER, 32, null)
Found (")", CLOSE_PAR, 32, null)
Found ("/", DIV, 32, null)
Found ("2.0", FLOAT_CONST, 32, 2)
Found (";", SEMICOLON, 32, null)
Found ("out", OUT, 33, null)
Found ("(", OPEN_PAR, 33, null)
Found ("Nova media: ", LITERAL, 33, Nova media:)
Found (")", CLOSE_PAR, 33, null)
Found (";", SEMICOLON, 33, null)
Found ("out", OUT, 34, null)
Found ("(", OPEN_PAR, 34, null)
Found ("media", IDENTIFIER, 34, null)
Found (")", CLOSE_PAR, 34, null)
Found (";", SEMICOLON, 34, null)
Found ("out", OUT, 35, null)
Found ("(", OPEN_PAR, 35, null)
Found ("Digite a opcao (S/N): ", LITERAL, 35, Digite a opcao (S/N):)
Found (")", CLOSE_PAR, 35, null)
Found (";", SEMICOLON, 35, null)
Found ("in", IN, 36, null)
Found ("(", OPEN_PAR, 36, null)
Found ("opcao", IDENTIFIER, 36, null)
Found (")", CLOSE_PAR, 36, null)
Found (";", SEMICOLON, 36, null)
Found ("end", END, 37, null)
Found (";", SEMICOLON, 37, null)
Found ("end", END, 38, null)
Found (";", SEMICOLON, 38, null)
Found ("", END_OF_FILE, 38, null)