RAFAEL DIAS CAMPOS

COMPILADORES - TRABALHO FINAL

Belo Horizonte-MG
2022

RAFAEL DIAS CAMPOS

COMPILADORES - TRABALHO FINAL

Relatório do compilador implementado no trabalho final da disciplina de Compiladores, ministrada pela professora Kecia Marques no primeiro semestre do ano letivo de 2022.

Belo Horizonte-MG
2022

Sumário

Resumo

Para este trabalho, foi definida pela professora a especificação de uma linguagem de programação. Em seguida, foi realizada a implementação de um compilador para esta linguagem.

Este trabalho retrata o desenvolvimento e validação dos módulos do compilador: Analisador Léxico, Analisador Sintático, Analisador Semântico e Gerador de Código.

Palavras-chave: Compilador, Léxico, Sintático, Semântico, Linguagem de Programação.

1 Introdução

Antes de se dar início a este trabalho, a professora Kecia Marques definiu a especificação de uma linguagem de programação para ser utilizada. A linguagem escolhida foi criada para este trabalho, e não retrata uma linguagem pré-existente.

A partir dessa especificação, foi primeiramente desenvolvido um Analisador Léxico para a linguagem, que é capaz de ler um arquivo fonte e emitir uma sequência de tokens.

Em seguida, foi desenvolvido um Analisador Sintático, que receberá os tokens provenientes do Analisador Léxico e irá produzir uma árvore de derivação para o arquivo fonte, com base na gramática definida para a linguagem.

Na próxima etapa, foi feita a criação de um Analisador Semântico, responsável por validar a árvore de derivação produzida pelo Analisador Sintático.

Finalmente, foi realizado um Gerador de Código, que recebe a árvore de derivação produzida pelo Analisador Sintático e validada pelo Analisador Semântico e produz seu código Assembly correspondente.

Além do processo de desenvolvimento dos módulos do compilador, esse trabalho também apresenta os resultados dos testes de validação realizados em cada etapa.

1.1 Motivação

Este trabalho foi realizado para colocar em prática os conhecimentos aprendidos em sala durante a disciplina de Compiladores.

Além disso, ele oferece uma introdução ao desenvolvimento de um Compilador para uma linguagem arbitrária e pode ser útil para o desenvolvimento de projetos similares no futuro.

1.2 Objetivos

O objetivo principal deste trabalho é produzir um compilador capaz de reconhecer a linguagem definida pela professora, e gerar o código correspondente.

Como objetivo secundário, foi escolhida a implementação de recuperação de erro, que possibilite a exibição de todos os erros presentes no código fonte com uma única execução do compilador.

2 Utilização

2.1 Download do Código

O código desenvolvido para o compilador encontra-se no repositório do GitHub:
https://github.com/RafaelDiasCampos/Compiler

2.2 Compilação

2.2.1 Dependências

Para realizar a compilação, primeiro é necessário instalar as seguintes dependências:

```
g++
cmake
```

2.2.2 Comando de Compilação

Compile com o seguintes comandos:

```
$ git clone https://github.com/RafaelDiasCampos/Compiler
$ cd Compiler
$ cmake .
$ cmake --build .
```

2.3 Execução

```
$ ./Compiler file
```

3 Descrição da Linguagem

3.1 Gramática

A linguagem foi definida pela seguinte Gramática Livre de Contexto (em notação BNF):

⟨*program*⟩ ::= ‘`routine`’ body

⟨*body*⟩ ::= [⟨*decl-list*⟩] ‘`begin`’ ⟨*stmt-list*⟩ ‘`end`’

⟨*decl-list*⟩ ::= ‘`declare`’ ⟨*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`’ ⟨*stmt-list*⟩ ‘`end`’
| ‘`if`’ ⟨*condition*⟩ ‘`then`’ ⟨*stmt-list*⟩ ‘`else`’ ⟨*stmt-list*⟩ ‘`end`’

⟨*condition*⟩ : ⟨*expression*⟩

⟨*repeat-stmt*⟩ ::= ‘`repeat`’ ⟨*stmt-list*⟩ ⟨*stmt-suffix*⟩

⟨*stmt-suffix*⟩ ::= ‘`until`’ ⟨*condition*⟩

⟨*while-stmt*⟩ ::= ⟨*stmt-prefix*⟩ ⟨*stmt-list*⟩ ‘`end`’

⟨*stmt-prefix*⟩ ::= ‘`while`’ ⟨*condition*⟩ ‘`do`’

⟨*read-stmt*⟩ ::= ‘`read`’ ‘`(`’ ⟨*identifier*⟩ ‘`)`’

⟨*write-stmt*⟩ ::= ‘`write`’ ‘`(`‘ <writable> ‘`)`’

⟨*writable*⟩ ::= ⟨*simple-expr*⟩ | ⟨*literal*⟩

⟨*expression*⟩ ::= ⟨*simple-expr*⟩ | ⟨*simple-expr*⟩ ⟨*relop*⟩ ⟨*simple-expr*⟩

⟨*simple-expr*⟩ ::= ⟨*term*⟩ | ⟨*simple-expr*⟩ ⟨*addop*⟩ ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor-a*⟩ | ⟨*term*⟩ ⟨*mulop*⟩ ⟨*factor-a*⟩

⟨*factor-a*⟩ ::= ⟨*factor*⟩ | ‘`not`’ ⟨*factor*⟩ | ‘−’ ⟨*factor*⟩

⟨*factor*⟩ ::= ⟨*identifier*⟩ | ⟨*constant*⟩ | ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*relop*⟩ ::= ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’

⟨*addop*⟩ ::= ‘+’ | ‘−’ | ‘`or`’

⟨*mulop*⟩ ::= ‘*’ | ‘/’ | ‘`and`’

⟨*constant*⟩ ::= ⟨*integer_const*⟩ | ⟨*float_const*⟩ | ⟨*char_const*⟩

3.2 Tokens

Os tokens foram definidos a partir das seguintes expressões regulares:

⟨*integer_const*⟩ ::= ⟨*digit*⟩+

⟨*float_const*⟩ ::= ⟨*digit*⟩+ ‘`.`’ ⟨*digit*⟩+

⟨*char_const*⟩ ::= ‘`'`’ ⟨*carac*⟩ ‘`'`’

⟨*literal*⟩ ::= ‘`"`’ ⟨*caractere*⟩* ‘`"`’

⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)*

⟨*letter*⟩ ::= [A-za-z]

⟨*digit*⟩ ::= [0-9]

⟨*carac*⟩ ::= .*

3.3 Outras características

A linguagem apresenta algumas outras características, descritas a seguir:

- Palavras-chave são reservadas
- Variáveis devem ser declaradas antes de seu uso
- Comentários se iniciam por ’%’ e terminal com ’%’
- Valores do tipo inteiro podem ser atribuídos a variáveis do tipo float, mas o inverso não é permitido
- O resultado de uma divisão é sempre um float
- A linguagem é case-sensitive

4 Analisador Léxico

Neste capítulo será descrita a implementação do Analisador Léxico. Esse componente do compilador é responsável por ler o arquivo fonte e transformá-lo em uma sequência de tokens.

4.1 Definição dos Tokens

Inicialmente, foi necessário definir quais serão os tokens da linguagem nessa implementação. Com base, na gramática apresentada na seção 3.1, foi definida a seguinte lista de tokens:

Tokens Básicos: ROUTINE, BEGIN, END, DECLARE, INT, FLOAT, CHAR, IF, THEN, ELSE, REPEAT, UNTIL, WHILE, DO, READ, WRITE, NOT, OR, AND, ASSIGN, ADD, SUB, MUL, DIV, COMP_EQ, COMP_NE, COMP_GT, COMP_GE, COMP_LT, COMP_LE, OPEN_BRACES, CLOSE_BRACES, SEMICOLON, COMMA

Constantes: CONST_INT, CONST_FLOAT, CONST_CHAR, CONST_STRING

Identificadores: ID

Outros Tokens: END_OF_FILE, INVALID_TOKEN, ERROR

4.2 Estrutura do programa

Foi criada uma classe Token para representar os tokens da linguagem. Essa classe possui um enum TokenType utilizado para armazenar o tipo de token que o objeto representa, com base na lista apresentada na seção 4.1. Para representar constantes, foi definida uma classe derivada de Token, chamada de ValueType. Dessa classe foram derivadas as classes TokenConstInt, TokenConstFloat, TokenConstChar e TokenConstString, com cada objeto armazenando seu respectivo valor. Finalmente, para Identificadores, foi criada a classe TokenId, derivada de Token, que armazena seu id e uma referência para o ValueType com seu respectivo valor.

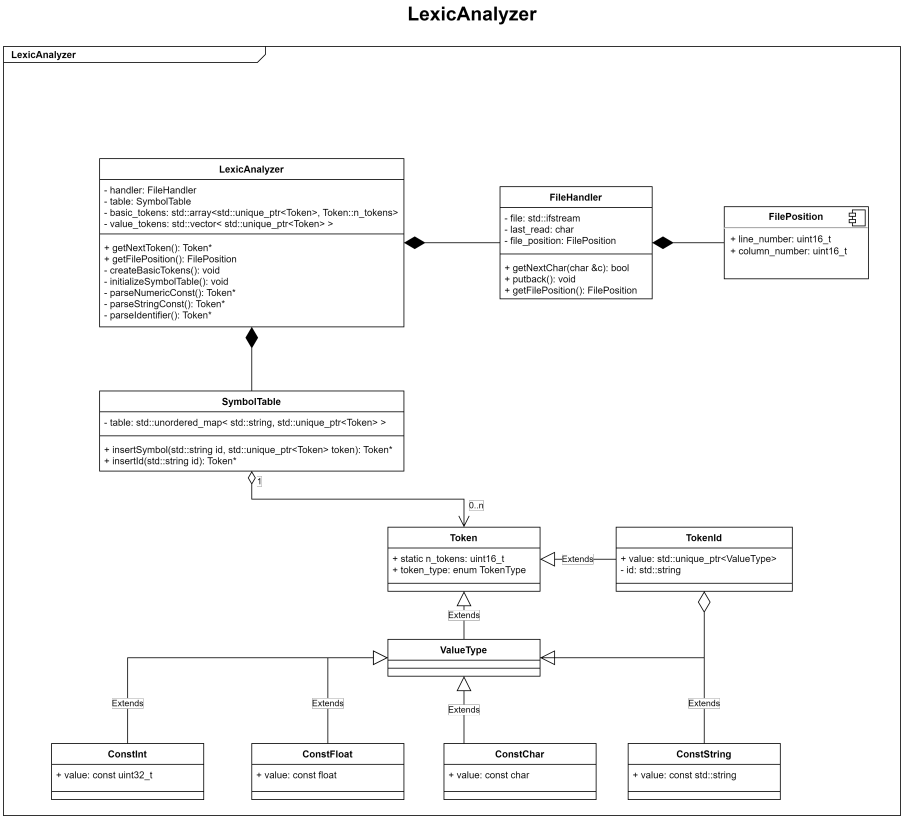
Para navegar no arquivo fonte, foi criada uma classe FileHandler, que é responsável por abrir e ler o arquivo. Essa classe também armazena um objeto do tipo struct chamado FilePosition, que indica a linha e a coluna atual do arquivo e também oferece uma função putback que retorna o último caractere lido para o buffer.

A implementação da tabela de símbolos foi realizada por meio da classe SymbolTable. Ela armazena um unordered_map (correspondente a um hash map) que relaciona o nome de um símbolo com seu respectivo objeto do tipo Token. Ela implementa dois métodos, insertSymbol() e insertId(), que podem ser utilizados para inserir um novo token na tabela, ou para retornar o token já existente.

O Analisador Léxico em si foi implementado por meio de um parser recursivo descendente. Ele possui um método principal, getNextToken(), que retorna um ponteiro para o próximo Token

lido. Para evitar criar tokens de forma desnecessária, ele armazena um array com os tokens básicos, já que eles são imutáveis. Ele também armazena um vetor com os ValueTokens que são criados. Essa parte é necessária pois eles são alocados de forma dinâmica na memória e portanto é preciso armazenar uma referência para posteriormente liberar a memória, e, no caso dos ValueTypes, não se pode saber a tempo de compilação até em que momento eles serão utilizados. O Analisador Léxico também armazena o FileHandler e a SymbolTable, para seu uso. Ao se iniciar a execução, a tabela de símbolos é populada com as palavras reservadas da linguagem, pois dessa forma elas podem ser tratadas pelo Léxico como identificadores durante o processo de parsing do arquivo.

A seguir encontra-se o diagrama de classes do programa, no estilo UML:



4.3 Testes de validação

4.3.1 Teste 01

4.3.1.1 Código Original

```
routine

declare

int a, b;
int resul;
```

```
float a,x;

begin

a := 12a;
x := 12;
read (a);
read (b);
read (c)
result := (a*b + 1) / (c+2);
write "Resultado: ";
write (result);

end
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: a
Token ,
Token id. Name: b
Token ;
Token int
Token id. Name: resul
Token ;
Token float
Token id. Name: a
Token ,
Token id. Name: x
Token ;
Token begin
Token id. Name: a
Token :=
Token const int. Value: 12
Token id. Name: a
Token ;
Token id. Name: x
Token :=
```

```
Token const int. Value: 12
Token ;
Token read
Token (
Token id. Name: a
Token )
Token ;
Token read
Token (
Token id. Name: b
Token )
Token ;
Token read
Token (
Token id. Name: c
Token )
Token id. Name: result
Token :=
Token (
Token id. Name: a
Token *
Token id. Name: b
Token +
Token const int. Value: 1
Token )
Token /
Token (
Token id. Name: c
Token +
Token const int. Value: 2
Token )
Token ;
Token write
Token const string. Value: "Resultado: "
Token ;
Token write
Token (
Token id. Name: result
Token )
```

```
Token ;
Token end

LexicAnalyzer:
SymbolTable:
result: Token id. Name: result
x: Token id. Name: x
and: Token and
c: Token id. Name: c
not: Token not
resul: Token id. Name: resul
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

4.3.2 Teste 02

4.3.2.1 Código Original

```
routine

int a, b, c;
float d, _var
```

```
begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula
write c;
val := 34.2
c = val + 2. + a;
write (val)

end
```

Saida do Compilador:

```
Token routine
Token int
Token id. Name: a
Token ,
Token id. Name: b
Token ,
Token id. Name: c
Token ;
Token float
Token id. Name: d
Token ,
Invalid token at position (4, 11)
Token id. Name: var
Token begin
Token read
Token (
Token id. Name: a
Token )
Token ;
Token id. Name: b
Token :=
Token id. Name: a
Token *
Token id. Name: a
Token ;
Token id. Name: c
```

```
Token :=
Token id. Name: b
Token +
Token id. Name: a
Token /
Token const int. Value: 2
Token *
Token (
Token const int. Value: 35
Token /
Token id. Name: b
Token )
Token ;
Invalid token at position (16, 4)

LexicAnalyzer:
SymbolTable:
var: Token id. Name: var
d: Token id. Name: d
and: Token and
c: Token id. Name: c
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
```

```
do: Token do
write: Token write
```

4.3.2.2 Código com Comentário Corrigido

```
routine

int a, b, c;
float d, _var

begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write c;
val := 34.2
c = val + 2. + a;
write (val)

end
```

Saida do Compilador:

4.3.2.3 Código Corrigido

```
routine

int a, b, c;
float d, var

begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write c;
val := 34.2
c = val + 2.0 + a;
write (val)
```

```
end
```

Saida do Compilador:

```
Token routine
Token int
Token id. Name: a
Token ,
Token id. Name: b
Token ,
Token id. Name: c
Token ;
Token float
Token id. Name: d
Token ,
Token id. Name: var
Token begin
Token read
Token (
Token id. Name: a
Token )
Token ;
Token id. Name: b
Token :=
Token id. Name: a
Token *
Token id. Name: a
Token ;
Token id. Name: c
Token :=
Token id. Name: b
Token +
Token id. Name: a
Token /
Token const int. Value: 2
Token *
Token (
Token const int. Value: 35
Token /
Token id. Name: b
```

```
Token )
Token ;
Token write
Token id. Name: c
Token ;
Token id. Name: val
Token :=
Token const float. Value: 34.200000
Token id. Name: c
Token =
Token id. Name: val
Token +
Token const float. Value: 2.000000
Token +
Token id. Name: a
Token ;
Token write
Token (
Token id. Name: val
Token )
Token end

LexicAnalyzer:
SymbolTable:
val: Token id. Name: val
var: Token id. Name: var
d: Token id. Name: d
and: Token and
c: Token id. Name: c
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
```

```
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

4.3.3 Teste 03

4.3.3.1 Código Original

```
routine

declare

int a, aux;
float b;

begin

B := 0;
read (a);
read(b);
if (a<>) b then
begin
    if (a>b) then
        aux := b;
        b := a;
        a := aux;
    end;
    write(a;
    write(b)
end
else
    write("Numeros iguais.");
```

```
end
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: a
Token ,
Token id. Name: aux
Token ;
Token float
Token id. Name: b
Token ;
Token begin
Token id. Name: B
Token :=
Token const int. Value: 0
Token ;
Token read
Token (
Token id. Name: a
Token )
Token ;
Token read
Token (
Token id. Name: b
Token )
Token ;
Token if
Token (
Token id. Name: a
Token <>
Token )
Token id. Name: b
Token then
Token begin
Token if
Token (
Token id. Name: a
```

```
Token >
Token id. Name: b
Token )
Token then
Token id. Name: aux
Token :=
Token id. Name: b
Token ;
Token id. Name: b
Token :=
Token id. Name: a
Token ;
Token id. Name: a
Token :=
Token id. Name: aux
Token ;
Token end
Token ;
Token write
Token (
Token id. Name: a
Token ;
Token write
Token (
Token id. Name: b
Token )
Token end
Token else
Token write
Token (
Token const string. Value: "Numeros iguais."
Token )
Token ;
Token end

LexicAnalyzer:
SymbolTable:
aux: Token id. Name: aux
and: Token and
```

```
not: Token not
B: Token id. Name: B
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

4.3.4 Teste 04

4.3.4.1 Código Original

```
routine

declare

int pontuacao, pontuacaoMaxina, disponibilidade;
char pontuacaoMinima;

begin

pontuacaoMinima = 50;
pontuacaoMaxima = 100;
write("Pontuacao do candidato: );
read(pontuacao);
write("Disponibilidade do candidato: ");
```

```
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima) do
    if ((pontuação > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end
    write("Pontuacao do candidato: ");
    read(pontuacao);
    write("Disponibilidade do candidato: ");
    read(disponibilidade);
end;

end
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: pontuacao
Token ,
Token id. Name: pontuacaoMaxina
Token ,
Token id. Name: disponibilidade
Token ;
Token char
Token id. Name: pontuacaoMinima
Token ;
Token begin
Token id. Name: pontuacaoMinima
Token =
Token const int. Value: 50
Token ;
Token id. Name: pontuacaoMaxima
```

```
Token =
Token const int. Value: 100
Token ;
Token write
Token (
Token const string. Value: "Pontuacao do candidato: );
read(pontuacao);
write("
Token id. Name: Disponibilidade
Token do
Token id. Name: candidato
Invalid token at position (14, 37)
Token const string. Value: ");
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima) do
    if ((pontuacão > pontuacaoMinima) and (disponibilidade=1)) then
        write("
Token id. Name: Candidato
Token id. Name: aprovado
Invalid token at position (23, 35)
Token const string. Value: ")
    else
        write("
Token id. Name: Candidato
Token id. Name: reprovado
Invalid token at position (25, 36)
Token const string. Value: ")
    end
    write("
Token id. Name: Pontuacao
Token do
Token id. Name: candidato
Invalid token at position (27, 35)
Token const string. Value: ");
```

```
    read(pontuacao);
    write("
Token id. Name: Disponibilidade
Token do
Token id. Name: candidato
Invalid token at position (29, 41)
Token const string. Value: ");
    read(disponibilidade);
end;

end"

LexicAnalyzer:
SymbolTable:
Candidato: Token id. Name: Candidato
Pontuacao: Token id. Name: Pontuacao
candidato: Token id. Name: candidato
pontuacaoMinima: Token id. Name: pontuacaoMinima
disponibilidade: Token id. Name: disponibilidade
pontuacaoMaxina: Token id. Name: pontuacaoMaxina
and: Token and
not: Token not
Disponibilidade: Token id. Name: Disponibilidade
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
aprovado: Token id. Name: aprovado
or: Token or
end: Token end
else: Token else
pontuacaoMaxima: Token id. Name: pontuacaoMaxima
begin: Token begin
repeat: Token repeat
```

```
pontuacao: Token id. Name: pontuacao
while: Token while
do: Token do
reprovado: Token id. Name: reprovado
write: Token write
```

4.3.4.2 Código com String Corrigida

```
routine

declare

int pontuacao, pontuacaoMaxina, disponibilidade;
char pontuacaoMinima;

begin

pontuacaoMinima = 50;
pontuacaoMaxima = 100;
write("Pontuacao do candidato: ");
read(pontuacao);
write("Disponibilidade do candidato: ");
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima) do
    if ((pontuação > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end
    write("Pontuacao do candidato: ");
    read(pontuacao);
    write("Disponibilidade do candidato: ");
    read(disponibilidade);
end;
```

```
end
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: pontuacao
Token ,
Token id. Name: pontuacaoMaxina
Token ,
Token id. Name: disponibilidade
Token ;
Token char
Token id. Name: pontuacaoMinima
Token ;
Token begin
Token id. Name: pontuacaoMinima
Token =
Token const int. Value: 50
Token ;
Token id. Name: pontuacaoMaxima
Token =
Token const int. Value: 100
Token ;
Token write
Token (
Token const string. Value: "Pontuacao do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: pontuacao
Token )
Token ;
Token write
Token (
Token const string. Value: "Disponibilidade do candidato: "
Token )
Token ;
```

```
Token read
Token (
Token id. Name: disponibilidade
Token )
Token ;
Token while
Token (
Token id. Name: pontuacao
Token >
Token const int. Value: 0
Token and
Token (
Token id. Name: pontuacao
Token <=
Token id. Name: pontuacaoMaxima
Token )
Token do
Token if
Token (
Token (
Token id. Name: pontua
Invalid token at position (22, 17)
Invalid token at position (22, 18)
Invalid token at position (22, 19)
Invalid token at position (22, 20)
Token id. Name: o
Token >
Token id. Name: pontuacaoMinima
Token )
Token and
Token (
Token id. Name: disponibilidade
Token =
Token const int. Value: 1
Token )
Token )
Token then
Token write
Token (
```

```
Token const string. Value: "Candidato aprovado."
Token )
Token else
Token write
Token (
Token const string. Value: "Candidato reprovado."
Token )
Token end
Token write
Token (
Token const string. Value: "Pontuacao do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: pontuacao
Token )
Token ;
Token write
Token (
Token const string. Value: "Disponibilidade do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: disponibilidade
Token )
Token ;
Token end
Token ;
Token end

LexicAnalyzer:
SymbolTable:
o: Token id. Name: o
pontua: Token id. Name: pontua
pontuacaoMinima: Token id. Name: pontuacaoMinima
disponibilidade: Token id. Name: disponibilidade
pontuacaoMaxina: Token id. Name: pontuacaoMaxina
```

```
and: Token and
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
pontuacaoMaxima: Token id. Name: pontuacaoMaxima
begin: Token begin
repeat: Token repeat
pontuacao: Token id. Name: pontuacao
while: Token while
do: Token do
write: Token write
```

4.3.4.3 Código Corrigido

```
routine

declare

int pontuacao, pontuacaoMaxina, disponibilidade;
char pontuacaoMinima;

begin

pontuacaoMinima = 50;
pontuacaoMaxima = 100;
write("Pontuacao do candidato: ");
read(pontuacao);
write("Disponibilidade do candidato: ");
read(disponibilidade);
```

```
%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima) do
    if ((pontuacao > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end
    write("Pontuacao do candidato: ");
    read(pontuacao);
    write("Disponibilidade do candidato: ");
    read(disponibilidade);
end;

end
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: pontuacao
Token ,
Token id. Name: pontuacaoMaxina
Token ,
Token id. Name: disponibilidade
Token ;
Token char
Token id. Name: pontuacaoMinima
Token ;
Token begin
Token id. Name: pontuacaoMinima
Token =
Token const int. Value: 50
Token ;
Token id. Name: pontuacaoMaxima
Token =
Token const int. Value: 100
```

```
Token ;
Token write
Token (
Token const string. Value: "Pontuacao do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: pontuacao
Token )
Token ;
Token write
Token (
Token const string. Value: "Disponibilidade do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: disponibilidade
Token )
Token ;
Token while
Token (
Token id. Name: pontuacao
Token >
Token const int. Value: 0
Token and
Token (
Token id. Name: pontuacao
Token <=
Token id. Name: pontuacaoMaxima
Token )
Token do
Token if
Token (
Token (
Token id. Name: pontuacao
Token >
Token id. Name: pontuacaoMinima
```

```
Token )
Token and
Token (
Token id. Name: disponibilidade
Token =
Token const int. Value: 1
Token )
Token )
Token then
Token write
Token (
Token const string. Value: "Candidato aprovado."
Token )
Token else
Token write
Token (
Token const string. Value: "Candidato reprovado."
Token )
Token end
Token write
Token (
Token const string. Value: "Pontuacao do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: pontuacao
Token )
Token ;
Token write
Token (
Token const string. Value: "Disponibilidade do candidato: "
Token )
Token ;
Token read
Token (
Token id. Name: disponibilidade
Token )
Token ;
```

```
Token end
Token ;
Token end

LexicAnalyzer:
SymbolTable:
pontuacaoMinima: Token id. Name: pontuacaoMinima
disponibilidade: Token id. Name: disponibilidade
pontuacaoMaxina: Token id. Name: pontuacaoMaxina
and: Token and
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
pontuacaoMaxima: Token id. Name: pontuacaoMaxima
begin: Token begin
repeat: Token repeat
pontuacao: Token id. Name: pontuacao
while: Token while
do: Token do
write: Token write
```

4.3.5 Teste 05

4.3.5.1 Código Original

```
declare

integer a, b, c, maior;
char outro;
```

```
begin

repeat
    write("A: ");
    read(a);
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) end
        maior := a
    else
        if (b>c) then
            maior := b;
        else
            maior := c
        end
    end
    write("Maior valor:"");
    write (maior);
    write ("Outro? (S/N)");
    read(outro);
until (outro = 'N' || outro = 'n)

end
```

Saida do Compilador:

```
Token declare
Token id. Name: integer
Token id. Name: a
Token ,
Token id. Name: b
Token ,
Token id. Name: c
Token ,
Token id. Name: maior
Token ;
Token char
Token id. Name: outro
```

```
Token ;
Token begin
Token repeat
Token write
Token (
Token const string. Value: "A: "
Token )
Token ;
Token read
Token (
Token id. Name: a
Token )
Token ;
Token write
Token (
Token const string. Value: "B: "
Token )
Token ;
Token read
Token (
Token id. Name: b
Token )
Token ;
Token write
Token (
Token const string. Value: "C: "
Token )
Token ;
Token read
Token (
Token id. Name: c
Token )
Token ;
Token if
Token (
Token (
Token id. Name: a
Token >
Token id. Name: b
```

```
Token )
Token and
Token (
Token id. Name: a
Token >
Token id. Name: c
Token )
Token )
Token end
Token id. Name: maior
Token :=
Token id. Name: a
Token else
Token if
Token (
Token id. Name: b
Token >
Token id. Name: c
Token )
Token then
Token id. Name: maior
Token :=
Token id. Name: b
Token ;
Token else
Token id. Name: maior
Token :=
Token id. Name: c
Token end
Token end
Token write
Token (
Token const string. Value: "Maior valor:"
Token const string. Value: ");
    write (maior);
    write ("
Token id. Name: Outro
Invalid token at position (26, 19)
Token (
```

```
Token id. Name: S
Token /
Token id. Name: N
Token )
Token const string. Value: ");
    read(outro);
until (outro = 'N' || outro = 'n)

end"

LexicAnalyzer:
SymbolTable:
Outro: Token id. Name: Outro
N: Token id. Name: N
and: Token and
S: Token id. Name: S
c: Token id. Name: c
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
char: Token char
outro: Token id. Name: outro
if: Token if
maior: Token id. Name: maior
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
integer: Token id. Name: integer
do: Token do
```

```
write: Token write
```

4.3.5.2 Código com String Corrigida

```
declare

integer a, b, c, maior;
char outro;

begin

repeat
    write("A: ");
    read(a);
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) end
        maior := a
    else
        if (b>c) then
            maior := b;
        else
            maior := c
        end
    end
    write("Maior valor:");
    write (maior);
    write ("Outro? (S/N)");
    read(outro);
until (outro = 'N' || outro = 'n)

end
```

Saida do Compilador:

```
Token declare
Token id. Name: integer
Token id. Name: a
Token ,
```

```
Token id. Name: b
Token ,
Token id. Name: c
Token ,
Token id. Name: maior
Token ;
Token char
Token id. Name: outro
Token ;
Token begin
Token repeat
Token write
Token (
Token const string. Value: "A: "
Token )
Token ;
Token read
Token (
Token id. Name: a
Token )
Token ;
Token write
Token (
Token const string. Value: "B: "
Token )
Token ;
Token read
Token (
Token id. Name: b
Token )
Token ;
Token write
Token (
Token const string. Value: "C: "
Token )
Token ;
Token read
Token (
Token id. Name: c
```

```
Token )
Token ;
Token if
Token (
Token (
Token id. Name: a
Token >
Token id. Name: b
Token )
Token and
Token (
Token id. Name: a
Token >
Token id. Name: c
Token )
Token )
Token end
Token id. Name: maior
Token :=
Token id. Name: a
Token else
Token if
Token (
Token id. Name: b
Token >
Token id. Name: c
Token )
Token then
Token id. Name: maior
Token :=
Token id. Name: b
Token ;
Token else
Token id. Name: maior
Token :=
Token id. Name: c
Token end
Token end
Token write
```

```
Token (
Token const string. Value: "Maior valor:"
Token )
Token ;
Token write
Token (
Token id. Name: maior
Token )
Token ;
Token write
Token (
Token const string. Value: "Outro? (S/N)"
Token )
Token ;
Token read
Token (
Token id. Name: outro
Token )
Token ;
Token until
Token (
Token id. Name: outro
Token =
Token const char. Value: 'N'
Invalid token at position (28, 21)
Invalid token at position (28, 22)
Token id. Name: outro
Token =
Invalid token at position (28, 33)
Token )
Token end

LexicAnalyzer:
SymbolTable:
and: Token and
c: Token id. Name: c
not: Token not
int: Token int
read: Token read
```

```
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
char: Token char
outro: Token id. Name: outro
if: Token if
maior: Token id. Name: maior
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
integer: Token id. Name: integer
do: Token do
write: Token write
```

4.3.5.3 Código Corrigido

```
declare

integer a, b, c, maior;
char outro;

begin

repeat
    write("A: ");
    read(a);
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) end
        maior := a
    else
```

```
        if (b>c) then
            maior := b;
        else
            maior := c
        end
    end
    write("Maior valor:");
    write (maior);
    write ("Outro? (S/N)");
    read(outro);
until (outro = 'N' or outro = 'n')

end
```

Saida do Compilador:

```
Token declare
Token id. Name: integer
Token id. Name: a
Token ,
Token id. Name: b
Token ,
Token id. Name: c
Token ,
Token id. Name: maior
Token ;
Token char
Token id. Name: outro
Token ;
Token begin
Token repeat
Token write
Token (
Token const string. Value: "A: "
Token )
Token ;
Token read
Token (
Token id. Name: a
Token )
```

```
Token ;
Token write
Token (
Token const string. Value: "B: "
Token )
Token ;
Token read
Token (
Token id. Name: b
Token )
Token ;
Token write
Token (
Token const string. Value: "C: "
Token )
Token ;
Token read
Token (
Token id. Name: c
Token )
Token ;
Token if
Token (
Token (
Token id. Name: a
Token >
Token id. Name: b
Token )
Token and
Token (
Token id. Name: a
Token >
Token id. Name: c
Token )
Token )
Token end
Token id. Name: maior
Token :=
Token id. Name: a
```

```
Token else
Token if
Token (
Token id. Name: b
Token >
Token id. Name: c
Token )
Token then
Token id. Name: maior
Token :=
Token id. Name: b
Token ;
Token else
Token id. Name: maior
Token :=
Token id. Name: c
Token end
Token end
Token write
Token (
Token const string. Value: "Maior valor:"
Token )
Token ;
Token write
Token (
Token id. Name: maior
Token )
Token ;
Token write
Token (
Token const string. Value: "Outro? (S/N)"
Token )
Token ;
Token read
Token (
Token id. Name: outro
Token )
Token ;
Token until
```

```
Token (
Token id. Name: outro
Token =
Token const char. Value: 'N'
Token or
Token id. Name: outro
Token =
Token const char. Value: 'n'
Token )
Token end

LexicAnalyzer:
SymbolTable:
and: Token and
c: Token id. Name: c
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
b: Token id. Name: b
a: Token id. Name: a
char: Token char
outro: Token id. Name: outro
if: Token if
maior: Token id. Name: maior
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
integer: Token id. Name: integer
do: Token do
write: Token write
```

4.3.6 Teste 06

4.3.6.1 Código Original

```
routine

declare

int qtd, currentNum, currentDiv, divisorFound;
double divResul;

begin

write("Quantos numeros deseja imprimir?")
read(qtd);

write("Numeros primos: ");

currentNum := 2;
while (qtd > 0) do
    currentNum := currentNum + 1;
    currentDiv := 2;
    divisorFound := 0;

    repeat
        divResul := currentNum / currentDiv;

        while ((divResul - 0.00.01) > 0) do
            divResul := divResul - 1;
        if (divResul >= -0.00.01) then
            divisorFound =  1

        currentDiv := currentDiv + 1;

    until ( (currentDiv > currentNum / 2) and (divisorFound = 1))

    if (divisorFound == 0) then
        qtd : qtd - 1;
        write(currentNum);
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: qtd
Token ,
Token id. Name: currentNum
Token ,
Token id. Name: currentDiv
Token ,
Token id. Name: divisorFound
Token ;
Token id. Name: double
Token id. Name: divResul
Token ;
Token begin
Token write
Token (
Token const string. Value: "Quantos numeros deseja imprimir?"
Token )
Token read
Token (
Token id. Name: qtd
Token )
Token ;
Token write
Token (
Token const string. Value: "Numeros primos: "
Token )
Token ;
Token id. Name: currentNum
Token :=
Token const int. Value: 2
Token ;
Token while
Token (
Token id. Name: qtd
Token >
Token const int. Value: 0
Token )
```

```
Token do
Token id. Name: currentNum
Token :=
Token id. Name: currentNum
Token +
Token const int. Value: 1
Token ;
Token id. Name: currentDiv
Token :=
Token const int. Value: 2
Token ;
Token id. Name: divisorFound
Token :=
Token const int. Value: 0
Token ;
Token repeat
Token id. Name: divResul
Token :=
Token id. Name: currentNum
Token /
Token id. Name: currentDiv
Token ;
Token while
Token (
Token (
Token id. Name: divResul
Token -
Token const float. Value: 0.000000
Invalid token at position (24, 33)
Token const int. Value: 1
Token )
Token >
Token const int. Value: 0
Token )
Token do
Token id. Name: divResul
Token :=
Token id. Name: divResul
Token -
```

```
Token const int. Value: 1
Token ;
Token if
Token (
Token id. Name: divResul
Token >=
Token -
Token const float. Value: 0.000000
Invalid token at position (26, 31)
Token const int. Value: 1
Token )
Token then
Token id. Name: divisorFound
Token =
Token const int. Value: 1
Token id. Name: currentDiv
Token :=
Token id. Name: currentDiv
Token +
Token const int. Value: 1
Token ;
Token until
Token (
Token (
Token id. Name: currentDiv
Token >
Token id. Name: currentNum
Token /
Token const int. Value: 2
Token )
Token and
Token (
Token id. Name: divisorFound
Token =
Token const int. Value: 1
Token )
Token )
Token if
Token (
```

```
Token id. Name: divisorFound
Token =
Token =
Token const int. Value: 0
Token )
Token then
Token id. Name: qtd
Invalid token at position (34, 14)
Token id. Name: qtd
Token -
Token const int. Value: 1
Token ;
Token write
Token (
Token id. Name: currentNum
Token )
Token ;

LexicAnalyzer:
SymbolTable:
divResul: Token id. Name: divResul
double: Token id. Name: double
currentDiv: Token id. Name: currentDiv
currentNum: Token id. Name: currentNum
qtd: Token id. Name: qtd
and: Token and
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
divisorFound: Token id. Name: divisorFound
or: Token or
end: Token end
```

```
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

4.3.6.2 Código Corrigido

```
routine

declare

int qtd, currentNum, currentDiv, divisorFound;
double divResul;

begin

write("Quantos numeros deseja imprimir?")
read(qtd);

write("Numeros primos: ");

currentNum := 2;
while (qtd > 0) do
    currentNum := currentNum + 1;
    currentDiv := 2;
    divisorFound := 0;

    repeat
        divResul := currentNum / currentDiv;

        while ((divResul - 0.0001) > 0) do
            divResul := divResul - 1;
        if (divResul >= -0.0001) then
            divisorFound =  1

        currentDiv := currentDiv + 1;

    until ( (currentDiv > currentNum / 2) and (divisorFound = 1))
```

```
    if (divisorFound == 0) then
        qtd := qtd - 1;
        write(currentNum);
```

Saida do Compilador:

```
Token routine
Token declare
Token int
Token id. Name: qtd
Token ,
Token id. Name: currentNum
Token ,
Token id. Name: currentDiv
Token ,
Token id. Name: divisorFound
Token ;
Token id. Name: double
Token id. Name: divResul
Token ;
Token begin
Token write
Token (
Token const string. Value: "Quantos numeros deseja imprimir?"
Token )
Token read
Token (
Token id. Name: qtd
Token )
Token ;
Token write
Token (
Token const string. Value: "Numeros primos: "
Token )
Token ;
Token id. Name: currentNum
Token :=
Token const int. Value: 2
Token ;
```

```
Token while
Token (
Token id. Name: qtd
Token >
Token const int. Value: 0
Token )
Token do
Token id. Name: currentNum
Token :=
Token id. Name: currentNum
Token +
Token const int. Value: 1
Token ;
Token id. Name: currentDiv
Token :=
Token const int. Value: 2
Token ;
Token id. Name: divisorFound
Token :=
Token const int. Value: 0
Token ;
Token repeat
Token id. Name: divResul
Token :=
Token id. Name: currentNum
Token /
Token id. Name: currentDiv
Token ;
Token while
Token (
Token (
Token id. Name: divResul
Token -
Token const float. Value: 0.000100
Token )
Token >
Token const int. Value: 0
Token )
Token do
```

```
Token id. Name: divResul
Token :=
Token id. Name: divResul
Token -
Token const int. Value: 1
Token ;
Token if
Token (
Token id. Name: divResul
Token >=
Token -
Token const float. Value: 0.000100
Token )
Token then
Token id. Name: divisorFound
Token =
Token const int. Value: 1
Token id. Name: currentDiv
Token :=
Token id. Name: currentDiv
Token +
Token const int. Value: 1
Token ;
Token until
Token (
Token (
Token id. Name: currentDiv
Token >
Token id. Name: currentNum
Token /
Token const int. Value: 2
Token )
Token and
Token (
Token id. Name: divisorFound
Token =
Token const int. Value: 1
Token )
Token )
```

```
Token if
Token (
Token id. Name: divisorFound
Token =
Token =
Token const int. Value: 0
Token )
Token then
Token id. Name: qtd
Token :=
Token id. Name: qtd
Token -
Token const int. Value: 1
Token ;
Token write
Token (
Token id. Name: currentNum
Token )
Token ;

LexicAnalyzer:
SymbolTable:
divResul: Token id. Name: divResul
double: Token id. Name: double
currentDiv: Token id. Name: currentDiv
currentNum: Token id. Name: currentNum
qtd: Token id. Name: qtd
and: Token and
not: Token not
int: Token int
read: Token read
float: Token float
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
divisorFound: Token id. Name: divisorFound
```

```
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

4.3.7 Teste 07

4.3.7.1 Código Original

```
routine

int qtd, currentNum
double lastNum;

bein

write("Quantos numeros deseja imprimir?")
read(qtd);

write("Sequencia de Fibonacci: ");

currentNum : 1;
lastNum := 1;

while (qtd > 0) d
    qtd := qtd - 2;
    write(lastNum);
    wrtie(currentNum);
    lastNum := currentNum + lastNum;
    currentNum := lastNum + currentNum;
```

Saida do Compilador:

```
Token routine
Token int
Token id. Name: qtd
```

```
Token ,
Token id. Name: currentNum
Token id. Name: double
Token id. Name: lastNum
Token ;
Token id. Name: bein
Token write
Token (
Token const string. Value: "Quantos numeros deseja imprimir?"
Token )
Token read
Token (
Token id. Name: qtd
Token )
Token ;
Token write
Token (
Token const string. Value: "Sequencia de Fibonacci: "
Token )
Token ;
Token id. Name: currentNum
Invalid token at position (13, 13)
Token const int. Value: 1
Token ;
Token id. Name: lastNum
Token :=
Token const int. Value: 1
Token ;
Token while
Token (
Token id. Name: qtd
Token >
Token const int. Value: 0
Token )
Token id. Name: d
Token id. Name: qtd
Token :=
Token id. Name: qtd
Token -
```

```
Token const int. Value: 2
Token ;
Token write
Token (
Token id. Name: lastNum
Token )
Token ;
Token id. Name: wrtie
Token (
Token id. Name: currentNum
Token )
Token ;
Token id. Name: lastNum
Token :=
Token id. Name: currentNum
Token +
Token id. Name: lastNum
Token ;
Token id. Name: currentNum
Token :=
Token id. Name: lastNum
Token +
Token id. Name: currentNum
Token ;

LexicAnalyzer:
SymbolTable:
d: Token id. Name: d
lastNum: Token id. Name: lastNum
double: Token id. Name: double
currentNum: Token id. Name: currentNum
qtd: Token id. Name: qtd
and: Token and
not: Token not
int: Token int
read: Token read
bein: Token id. Name: bein
float: Token float
wrtie: Token id. Name: wrtie
```

```
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

4.3.7.2 Código Corrigido

```
routine

int qtd, currentNum
double lastNum;

bein

write("Quantos numeros deseja imprimir?")
read(qtd);

write("Sequencia de Fibonacci: ");

currentNum := 1;
lastNum := 1;

while (qtd > 0) d
    qtd := qtd - 2;
    write(lastNum);
    wrtie(currentNum);
    lastNum := currentNum + lastNum;
    currentNum := lastNum + currentNum;
```

Saida do Compilador:

```
Token routine
Token int
Token id. Name: qtd
Token ,
Token id. Name: currentNum
Token id. Name: double
Token id. Name: lastNum
Token ;
Token id. Name: bein
Token write
Token (
Token const string. Value: "Quantos numeros deseja imprimir?"
Token )
Token read
Token (
Token id. Name: qtd
Token )
Token ;
Token write
Token (
Token const string. Value: "Sequencia de Fibonacci: "
Token )
Token ;
Token id. Name: currentNum
Token :=
Token const int. Value: 1
Token ;
Token id. Name: lastNum
Token :=
Token const int. Value: 1
Token ;
Token while
Token (
Token id. Name: qtd
Token >
Token const int. Value: 0
Token )
Token id. Name: d
```

```
Token id. Name: qtd
Token :=
Token id. Name: qtd
Token -
Token const int. Value: 2
Token ;
Token write
Token (
Token id. Name: lastNum
Token )
Token ;
Token id. Name: wrtie
Token (
Token id. Name: currentNum
Token )
Token ;
Token id. Name: lastNum
Token :=
Token id. Name: currentNum
Token +
Token id. Name: lastNum
Token ;
Token id. Name: currentNum
Token :=
Token id. Name: lastNum
Token +
Token id. Name: currentNum
Token ;

LexicAnalyzer:
SymbolTable:
d: Token id. Name: d
lastNum: Token id. Name: lastNum
double: Token id. Name: double
currentNum: Token id. Name: currentNum
qtd: Token id. Name: qtd
and: Token and
not: Token not
int: Token int
```

```
read: Token read
bein: Token id. Name: bein
float: Token float
wrtie: Token id. Name: wrtie
until: Token until
routine: Token routine
declare: Token declare
then: Token then
char: Token char
if: Token if
or: Token or
end: Token end
else: Token else
begin: Token begin
repeat: Token repeat
while: Token while
do: Token do
write: Token write
```

5 Analisador Sintático

5.1 Alteração na Gramática

A gramática original apresenta recursão a esquerda, o que impossibilita a construção de um parser recursivo descendente que a reconheça. Por isso, foi feita a seguinte modificação na gramática para eliminar essa recursão:

⟨*program*⟩ ::= ‘`routine`’ body

⟨*body*⟩ ::= [⟨*decl-list*⟩] ‘`begin`’ ⟨*stmt-list*⟩ ‘`end`’

⟨*decl-list*⟩ ::= ‘`declare`’ ⟨*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-a*⟩

⟨*if-stmt*⟩ ::= ‘`if`’ ⟨*condition*⟩ ‘`then`’ ⟨*stmt-list*⟩ ‘`end`’
| ‘`if`’ ⟨*condition*⟩ ‘`then`’ ⟨*stmt-list*⟩ ‘`else`’ ⟨*stmt-list*⟩ ‘`end`’

⟨*condition*⟩ : ⟨*expression*⟩

⟨*repeat-stmt*⟩ ::= ‘`repeat`’ ⟨*stmt-list*⟩ ⟨*stmt-suffix*⟩

⟨*stmt-suffix*⟩ ::= ‘`until`’ ⟨*condition*⟩

⟨*while-stmt*⟩ ::= ⟨*stmt-prefix*⟩ ⟨*stmt-list*⟩ ‘`end`’

⟨*stmt-prefix*⟩ ::= ‘`while`’ ⟨*condition*⟩ ‘`do`’

⟨*read-stmt*⟩ ::= ‘`read`’ ‘`(`’ ⟨*identifier*⟩ ‘`)`’

⟨*write-stmt*⟩ ::= ‘`write`‘ ‘`(`‘ <writable> ‘`)`’

⟨*writable*⟩ ::= ⟨*simple-expr-a*⟩ | ⟨*literal*⟩

⟨*expression*⟩ ::= ⟨*simple-expr-a*⟩ | ⟨*simple-expr-a*⟩ ⟨*relop*⟩ ⟨*simple-expr-a*⟩

⟨*simple-expr-a*⟩ ::= ⟨*term-a*⟩ ⟨*simple-expr*⟩

⟨*simple-expr*⟩ ::= ⟨*addop*⟩ ⟨*term-a*⟩ ⟨*simple-expr*⟩ | lambda

⟨*term-a*⟩ ::= ⟨*factor-a*⟩ ⟨*term*⟩

⟨*term*⟩ ::= ⟨*mulop*⟩ ⟨*factor-a*⟩ ⟨*term*⟩ | lambda

⟨*factor-a*⟩ ::= ⟨*factor*⟩ | ‘`not`’ ⟨*factor*⟩ | ‘`-`’ ⟨*factor*⟩

⟨*factor*⟩ ::= ⟨*identifier*⟩ | ⟨*constant*⟩ | ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*relop*⟩ ::= ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’

⟨*addop*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`or`’

⟨*mulop*⟩ ::= ‘`*`’ | ‘`/`’ | ‘`and`’

⟨*constant*⟩ ::= ⟨*integer_const*⟩ | ⟨*float_const*⟩ | ⟨*char_const*⟩

5.2 Calculo de First e Follow

Para ajudar na implementação do compilador, foi realizad o cálculo do First e Follow para cada não-terminal da gramática:

- **First**:
  - <*program*>: ‘routine’
  - <*body*>: ‘begin’ | ‘declare’
  - <*decl-list*>: ‘declare’
  - <*decl*>: ‘int’ | ‘float’ | ‘char’
  - <*ident-list*>: <identifier>
  - <*type*>: ‘int’ | ‘float’ | ‘char’
  - <*stmt-list*>: <identifier> | ‘if’ | ‘while’ | ‘repeat’ | ‘read’ | ‘write’
  - <*stmt*>: <identifier> | ‘if’ | ‘while’ | ‘repeat’ | ‘read’ | ‘write’
  - <*assign-stmt*>: <identifier>
  - <*if-stmt*>: ‘if’
  - <*condition*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_-const>
  - <*repeat-stmt*>: ‘repeat’
  - <*stmt-suffix*>: ‘until’

  - <*while-stmt*>: ‘while’
  - <*stmt-prefix*>: ‘while’
  - <*read-stmt*>: ‘read’
  - <*write-stmt*>: ‘write’
  - <*writable*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <literal> | <integer_const> | <float_const> | <char_const>
  - <*expression*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_-const>
  - <*simple-expr-a*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_const>
  - <*simple-expr*>: ‘-’ | ‘+’ | ‘or’ | lambda
  - <*term-a*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_-const>
  - <*term*>: ‘*’ | ‘/’ | ‘and’ | lambda
  - <*factor-a*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_-const>
  - <*factor*>: <identifier> | ‘(’ | <integer_const> | <float_const> | <char_const>
  - <*relop*>: ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’
  - <*addop*>: ‘+’ | ‘-’ | ‘or’
  - <*mulop*>: ‘*’ | ‘/’ | ‘and’
  - <*constant*>: <integer_const> | <float_const> | <char_const>

- **Follow**:

  - <*program*>: $
  - <*body*>: $
  - <*decl-list*>: ‘begin’
  - <*decl*>: ‘;’
  - <*ident-list*>: ‘;’
  - <*type*>: <identifier>
  - <*stmt-list*>: ‘end’ | ‘else’ | ‘until’
  - <*stmt*>: ‘end’ | ‘else’ | ‘until’
  - <*assign-stmt*>: ‘end’ | ‘else’ | ‘until’
  - <*if-stmt*>: ‘end’ | ‘else’ | ‘until’

- <*condition*>: ‘end’ | ‘else’ | ‘until’ | ‘then’ | ‘do’
- <*repeat-stmt*>: ‘end’ | ‘else’ | ‘until’
- <*stmt-suffix*>: ‘end’ | ‘else’ | ‘until’
- <*while-stmt*>: ‘end’ | ‘else’ | ‘until’
- <*stmt-prefix*>: <identifier> | ‘if’ | ‘repeat’ | ‘while’ | ‘read’ | ‘write’
- <*read-stmt*>: ‘end’ | ‘else’ | ‘until’
- <*write-stmt*>: ‘end’ | ‘else’ | ‘until’
- <*writable*>: ‘)’
- <*expression*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’
- <*simple-expr-a*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’
- <*simple-expr*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’
- <*term-a*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘-’ | ‘+’ | ‘or’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’
- <*term*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘-’ | ‘+’ | ‘or’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’
- <*factor-a*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘-’ | ‘+’ | ‘or’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’ | ‘*’ | ‘/’ | ‘and’
- <*factor*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘-’ | ‘+’ | ‘or’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’ | ‘*’ | ‘/’ | ‘and’
- <*relop*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_const>
- <*addop*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_const>
- <*mulop*>: <identifier> | ‘(’ | ‘not’ | ‘-’ | <integer_const> | <float_const> | <char_const>
- <*constant*>: ‘end’ | ‘then’ | ‘else’ | ‘until’ | ‘do’ | ‘)’ | ‘-’ | ‘+’ | ‘or’ | ‘=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘<>’ | ‘*’ | ‘/’ | ‘and’

5.3 Estrutura do Programa

Foi feita implementação de um Parser Recursivo Descendente para reconhecer a gramática da linguagem tilizando o algoritmo de parsing LL(1) .

Cada regra da gramática foi representada por uma classe que herda de **Construct**. Essas regras recebem como parâmetros objetos do tipo **Token** ou do tipo **Construct** que representam a derivação escolhida para essa regra.

5.4 Recuperação de Erros

Para recuperar os erros de sintaxe, foi utilizado o método baseado na análise do Follow de cada regra da gramática. Ao encontrar um erro na construção de uma regra, o programa "come"novos Tokens até encontrar um que pertence ao Follow da regra que estava sendo construída. Dessa forma, é garantido que pelo menos o início da construção da regra seguinte será possível, o que permite um reconhecimento maior de erros.

5.5 Testes de validação

5.5.1 Teste 01

5.5.1.1 Código Original

```
routine

declare

int a, b;
int resul;
float a,x;

begin

a := 12a;
x := 12;
read (a);
read (b);
read (c)
result := (a*b + 1) / (c+2);
write "Resultado: ";
write (result);

end
```

Saida do Compilador:

```
Error: Unexpected token at position (11, 9)
```

5.5.1.2 Código Corrigido

```
routine
```

```
declare

int a, b;
int resul;
float a,x;

begin

a := 12;
x := 12;
read (a);
read (b);
read (c);
result := (a*b + 1) / (c+2);
write ("Resultado: ");
write (result)

end
```

Saida do Compilador:

```
Compilation successful!
```

5.5.2 Teste 02

5.5.2.1 Código Original

```
routine

int a, b, c;
float d, var

begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write c;
val := 34.2
c = val + 2.0 + a;
```

```
write (val)

end
```

Saida do Compilador:

```
Error: Unexpected token at position (3, 4)
Error: Unexpected token at position (11, 8)
```

5.5.2.2 Código Corrigido

```
routine

declare

int a, b, c;
float d, var;

begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write (c);
val := 34.2;
c := val + 2.0 + a;
write (val)

end
```

Saida do Compilador:

```
Compilation successful!
```

5.5.3 Teste 03

5.5.3.1 Código Original

```
routine

declare
```

```
int a, aux;
float b;

begin

B := 0;
read (a);
read(b);
if (a<>) b then
begin
    if (a>b) then
        aux := b;
        b := a;
        a := aux;
    end;
    write(a;
    write(b)
end
else
    write("Numeros iguais.");

end
```

Saida do Compilador:

```
Error: Unexpected token at position (13, 9)
Error: Unexpected token at position (13, 11)
```

5.5.4 Teste 04

5.5.4.1 Código Original

```
routine

declare

int pontuacao, pontuacaoMaxina, disponibilidade;
char pontuacaoMinima;

begin
```

```
pontuacaoMinima = 50;
pontuacaoMaxima = 100;
write("Pontuacao do candidato: ");
read(pontuacao);
write("Disponibilidade do candidato: ");
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima) do
    if ((pontuacao > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end
    write("Pontuacao do candidato: ");
    read(pontuacao);
    write("Disponibilidade do candidato: ");
    read(disponibilidade);
end;

end
```

Saida do Compilador:

```
Error: Unexpected token at position (10, 18)
Error: Unexpected token at position (24, 0)
```

5.5.4.2 Código Corrigido

```
routine

declare

int pontuacao, pontuacaoMaxina, disponibilidade;
char pontuacaoMinima;

begin
```

```
pontuacaoMinima := 50;
pontuacaoMaxima := 100;
write("Pontuacao do candidato: ");
read(pontuacao);
write("Disponibilidade do candidato: ");
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima)) do
    if ((pontuacao > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end;
    write("Pontuacao do candidato: ");
    read(pontuacao);
    write("Disponibilidade do candidato: ");
    read(disponibilidade)
end

end
```

Saida do Compilador:

```
Compilation successful!
```

5.5.5 Teste 05

5.5.5.1 Código Original

```
declare

integer a, b, c, maior;
char outro;

begin

repeat
```

```
    write("A: ");
    read(a);
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) end
        maior := a
    else
        if (b>c) then
            maior := b;
        else
            maior := c
        end
    end
    write("Maior valor:");
    write (maior);
    write ("Outro? (S/N)");
    read(outro);
until (outro = 'N' or outro = 'n')

end
```

Saida do Compilador:

```
Error: Unexpected token at position (1, 0)
```

5.5.5.2 Código com Token Routine

```
routine

declare

integer a, b, c, maior;
char outro;

begin

repeat
    write("A: ");
    read(a);
```

```
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) end
        maior := a
    else
        if (b>c) then
            maior := b;
        else
            maior := c
        end
    end
    write("Maior valor:");
    write (maior);
    write ("Outro? (S/N)");
    read(outro);
until (outro = 'N' or outro = 'n')

end
```

Saida do Compilador:

```
Error: Unexpected token at position (5, 8)
Error: Unexpected token at position (5, 10)
Error: Unexpected token at position (17, 0)
Error: Unexpected token at position (17, 0)
```

5.5.5.3 Código Corrigido

```
routine

declare

int a, b, c, maior;
char outro;

begin

repeat
    write("A: ");
```

```
    read(a);
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) then
        maior := a
    else
        if (b>c) then
            maior := b
        else
            maior := c
        end
    end;
    write("Maior valor:");
    write (maior);
    write ("Outro? (S/N)");
    read(outro)
until ((outro = 'N') or (outro = 'n'))

end
```

Saida do Compilador:

```
Compilation successful!
```

5.5.6 Teste 06

5.5.6.1 Código Original

```
routine

declare

int qtd, currentNum, currentDiv, divisorFound;
double divResul;

begin

write("Quantos numeros deseja imprimir?")
read(qtd);
```

```
write("Numeros primos: ");

currentNum := 2;
while (qtd > 0) do
    currentNum := currentNum + 1;
    currentDiv := 2;
    divisorFound := 0;

    repeat
        divResul := currentNum / currentDiv;

        while ((divResul - 0.0001) > 0) do
            divResul := divResul - 1;
        if (divResul >= -0.0001) then
            divisorFound =  1

        currentDiv := currentDiv + 1;

    until ( (currentDiv > currentNum / 2) and (divisorFound = 1))

    if (divisorFound == 0) then
        qtd := qtd - 1;
        write(currentNum);
```

Saida do Compilador:

```
Error: Unexpected token at position (6, 7)
```

5.5.6.2 Código Corrigido

```
routine

declare

int qtd, currentNum, currentDiv, divisorFound;
float divResul;

begin

write("Quantos numeros deseja imprimir?");
```

```
read(qtd);

write("Numeros primos: ");

currentNum := 2;
while (qtd > 0) do
    currentNum := currentNum + 1;
    currentDiv := 2;
    divisorFound := 0;

    repeat
        divResul := currentNum / currentDiv;

        while ((divResul - 0.0001) > 0) do
            divResul := divResul - 1
        end;
        if (divResul >= -0.0001) then
            divisorFound :=  1
        end;

        currentDiv := currentDiv + 1

    until ( (currentDiv > currentNum / 2) and (divisorFound = 1));

    if (divisorFound = 0) then
        qtd := qtd - 1;
        write(currentNum)
    end
end

end
```

Saida do Compilador:

```
Compilation successful!
```

5.5.7 Teste 07

5.5.7.1 Código Original

```
routine
```

```
int qtd, currentNum
double lastNum;

bein

write("Quantos numeros deseja imprimir?")
read(qtd);

write("Sequencia de Fibonacci: ");

currentNum := 1;
lastNum := 1;

while (qtd > 0) d
    qtd := qtd - 2;
    write(lastNum);
    wrtie(currentNum);
    lastNum := currentNum + lastNum;
    currentNum := lastNum + currentNum;
```

Saida do Compilador:

```
Error: Unexpected token at position (3, 4)
Error: Unexpected token at position (22, 5)
```

5.5.7.2 Código Corrigido

```
routine

declare

int qtd, currentNum;
float lastNum;

begin

write("Quantos numeros deseja imprimir?");
read(qtd);
```

```
write("Sequencia de Fibonacci: ");

currentNum := 1;
lastNum := 1;

while (qtd > 0) do
    qtd := qtd - 2;
    write(lastNum);
    write(currentNum);
    lastNum := currentNum + lastNum;
    currentNum := lastNum + currentNum
end

end
```

Saida do Compilador:

```
Compilation successful!
```

6 Analisador Semântico

6.1 Regras Semânticas

Foram implementadas as seguintes regras semânticas:

- **Regra 1**: Variáveis devem ser declaradas antes de serem utilizadas.
- **Regra 2**: Variáveis somente podem ser declaradas uma única vez.
- **Regra 3**: Variáveis do tipo **float** pode receber valores do tipo **int**, entretanto o inverso não é permitido.
- **Regra 4**: Resultados de divisão devem ser do tipo **float**.

6.2 Estrutura do Programa

Para o analisador Semântico, as regras foram implementadas junto com o código do Analisador Sintático. Com isso, assim que uma construção é analisada pelo Analisador Sintático, ela é passada para o Analisador Semântico.

6.3 Testes de validação

6.4 Testes de validação

6.4.1 Teste 01

6.4.1.1 Código Original

```
routine

declare

int a, b;
int resul;
float a,x;

begin

a := 12;
```

```
x := 12;
read (a);
read (b);
read (c);
result := (a*b + 1) / (c+2);
write ("Resultado: ");
write (result)

end
```

Saida do Compilador:

```
Error: Duplicate Identifier at position (7, 8)
Error: Undeclared Variable at position (16, 25)
Error: Undeclared Variable at position (16, 29)
Error: Undeclared Variable at position (18, 14)
```

6.4.1.2 Código com declaração corrigida

```
routine

declare

int a, b, c;
int result;
float x;

begin

a := 12;
x := 12;
read (a);
read (b);
read (c);
result := (a*b + 1) / (c+2);
write ("Resultado: ");
write (result)

end
```

Saida do Compilador:

```
Error: Incompatible types at position (16, 29)
```

6.4.1.3 Código Corrigido

```
routine

declare

int a, b, c;
float x, result;

begin

a := 12;
x := 12;
read (a);
read (b);
read (c);
result := (a*b + 1) / (c+2);
write ("Resultado: ");
write (result)

end
```

Saida do Compilador:

```
Compilation successful!
```

6.4.2 Teste 02

6.4.2.1 Código Original

```
routine

declare

int a, b, c;
float d, var;

begin

read (a);
```

```
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write (c);
val := 34.2;
c := val + 2.0 + a;
write (val)

end
```

Saida do Compilador:

```
Error: Incompatible types at position (12, 23)
Error: Undeclared Variable at position (14, 13)
Error: Undeclared Variable at position (15, 9)
Error: Undeclared Variable at position (16, 11)
```

6.4.2.2 Código com declaração corrigida

```
routine

declare

int a, b, c;
float d, val;

begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write (c);
val := 34.2;
c := val + 2.0 + a;
write (val)

end
```

Saida do Compilador:

```
Error: Incompatible types at position (12, 23)
Error: Incompatible types at position (15, 20)
```

6.4.2.3 Código Corrigido

```
routine

declare

int a, b;
float c, d, val;

begin

read (a);
b := a * a;
c := b + a/2 * (35/b); %aplica formula%
write (c);
val := 34.2;
c := val + 2.0 + a;
write (val)

end
```

Saida do Compilador:

```
Compilation successful!
```

6.4.3 Teste 03

6.4.3.1 Código Original

```
routine

declare

int a, aux;
float b;

begin

B := 0;
read (a);
read(b);
if (a<>b) then
```

```
    if (a>b) then
        aux := b;
        b := a;
        a := aux
    end;
    write(a);
    write(b)
else
    write("Numeros iguais.")
end

end
```

Saida do Compilador:

```
Error: Undeclared Variable at position (10, 8)
Error: Incompatible types at position (15, 18)
```

6.4.3.2 Código Corrigido

```
routine

declare

int a, b, aux;

begin

b := 0;
read (a);
read(b);
if (a<>b) then
    if (a>b) then
        aux := b;
        b := a;
        a := aux
    end;
    write(a);
    write(b)
else
    write("Numeros iguais.")
```

```
end

end
```

Saida do Compilador:

```
Compilation successful!
```

6.4.4 Teste 04

6.4.4.1 Código Original

```
routine

declare

int pontuacao, pontuacaoMaxina, disponibilidade;
char pontuacaoMinima;

begin

pontuacaoMinima := 50;
pontuacaoMaxima := 100;
write("Pontuacao do candidato: ");
read(pontuacao);
write("Disponibilidade do candidato: ");
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima)) do
    if ((pontuacao > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end;
    write("Pontuacao do candidato: ");
    read(pontuacao);
    write("Disponibilidade do candidato: ");
```

```
    read(disponibilidade)
end

end
```

Saida do Compilador:

```
Error: Incompatible types at position (10, 23)
Error: Undeclared Variable at position (11, 24)
Error: Undeclared Variable at position (21, 51)
Error: Incompatible types at position (22, 38)
```

6.4.4.2 Código Corrigido

```
routine

declare

int pontuacao, pontuacaoMinima, pontuacaoMaxima, disponibilidade;

begin

pontuacaoMinima := 50;
pontuacaoMaxima := 100;
write("Pontuacao do candidato: ");
read(pontuacao);
write("Disponibilidade do candidato: ");
read(disponibilidade);

%
Processamento
%

while (pontuacao>0 and (pontuacao<=pontuacaoMaxima)) do
    if ((pontuacao > pontuacaoMinima) and (disponibilidade=1)) then
        write("Candidato aprovado.")
    else
        write("Candidato reprovado.")
    end;
    write("Pontuacao do candidato: ");
    read(pontuacao);
```

```
    write("Disponibilidade do candidato: ");
    read(disponibilidade)
end

end
```

Saida do Compilador:

```
Compilation successful!
```

6.4.5 Teste 05

6.4.5.1 Código Original

```
routine

declare

int a, b, c, maior;
char outro;

begin

repeat
    write("A: ");
    read(a);
    write("B: ");
    read(b);
    write("C: ");
    read(c);
    if ( (a>b) and (a>c) ) then
        maior := a
    else
        if (b>c) then
            maior := b
        else
            maior := c
        end
    end;
    write("Maior valor:");
    write (maior);
```

```
    write ("Outro? (S/N)");
    read(outro)
until ((outro = 'N') or (outro = 'n'))

end
```

Saida do Compilador:

```
Compilation successful!
```

6.4.6 Teste 06

6.4.6.1 Código Original

```
routine

declare

int qtd, currentNum, currentDiv, divisorFound;
float divResul;

begin

write("Quantos numeros deseja imprimir?");
read(qtd);

write("Numeros primos: ");

currentNum := 2;
while (qtd > 0) do
    currentNum := currentNum + 1;
    currentDiv := 2;
    divisorFound := 0;

    repeat
        divResul := currentNum / currentDiv;

        while ((divResul - 0.0001) > 0) do
            divResul := divResul - 1
        end;
        if (divResul >= -0.0001) then
```

```
            divisorFound :=  1
        end;

        currentDiv := currentDiv + 1

    until ( (currentDiv > currentNum / 2) and (divisorFound = 1));

    if (divisorFound = 0) then
        qtd := qtd - 1;
        write(currentNum)
    end
end

end
```

Saida do Compilador:

```
Compilation successful!
```

6.4.7 Teste 07

6.4.7.1 Código Original

```
routine

declare

int qtd, currentNum;
float lastNum;

begin

write("Quantos numeros deseja imprimir?");
read(qtd);

write("Sequencia de Fibonacci: ");

currentNum := 1;
lastNum := 1;

while (qtd > 0) do
```

```
    qtd := qtd - 2;
    write(lastNum);
    write(currentNum);
    lastNum := currentNum + lastNum;
    currentNum := lastNum + currentNum
end

end
```

Saida do Compilador:

```
Error: Incompatible types at position (24, 0)
```

6.4.7.2 Código Corrigido

```
routine

declare

int qtd, currentNum, lastNum;

begin

write("Quantos numeros deseja imprimir?");
read(qtd);

write("Sequencia de Fibonacci: ");

currentNum := 1;
lastNum := 1;

while (qtd > 0) do
    qtd := qtd - 2;
    write(lastNum);
    write(currentNum);
    lastNum := currentNum + lastNum;
    currentNum := lastNum + currentNum
end

end
```

Saida do Compilador:

```
Compilation successful!
```

7 Gerador de Código

8 Conclusões