CEFET-MG
Centro Federal de Educação Tecnológica de Minas Gerais

Compilador

Analisador Sintático

Erick H. D. de Souza

Lucas C. Dornelas

Násser R. F. Kilesse

Professora Orientadora: Kecia Aline Marques Ferreira

Maio de 2023

Sumário

Resumo

O analisador sintático é responsável por analisar a estrutura gramatical de uma sentença, verificando se está de acordo com as regras definidas pela gramática da linguagem. O desenvolvimento do analisador envolveu a utilização de técnicas como análise descendente e tabelas de análise, permitindo a identificação de elementos como sujeito, verbo e complementos em uma sentença. Foram empregadas um código em Java e ferramentas adequadas para a implementação do analisador de acordo com as aulas de Compiladores.

Palavras-chave: analisador sintático, estrutura gramatical, análise descendente, tabelas de análise, sujeito, verbo, complementos, compiladores, Java.

Introdução

A implementação de um analisador sintático é um tema de grande relevância no campo da ciência da computação e da linguística computacional. O analisador sintático desempenha um papel fundamental na compreensão da estrutura gramatical de uma sentença em uma determinada linguagem, permitindo uma análise precisa da sintaxe e da semântica. Nesse contexto, este relatório descreve o desenvolvimento de um analisador sintático como parte de um trabalho universitário.

O objetivo principal deste trabalho é criar um analisador sintático capaz de analisar a estrutura gramatical de sentenças em uma linguagem específica. Para isso, foram empregadas técnicas de análise descendente e tabelas de análise, que possibilitaram a identificação de componentes como sujeito, verbo e complementos em uma sentença. Além disso, o trabalho envolveu a escolha adequada de linguagens de programação e ferramentas que facilitassem a implementação do analisador sintático.

Ao desenvolver o analisador sintático, espera-se obter um sistema eficiente e preciso, capaz de analisar uma ampla variedade de sentenças em tempo hábil. A implementação do analisador contribuirá para a compreensão mais profunda da gramática da linguagem em questão e poderá ser aplicada em diferentes áreas, como processamento de linguagem natural, tradução automática e verificação de conformidade de código.

Desenvolvimento

Forma de uso do compilador

Para utilizar o compilador, siga os passos abaixo:

1. Extraia o arquivo **`Compilers.zip`**.
2. Abra o terminal e navegue até o diretório onde foi criada a pasta extraída.
3. Execute o compilador utilizando o comando:

   `java -jar Compilers.jar <file-path>`.

   3.1. Para executar um dos programas presentes dentro da pasta **`programs`**, onde **`<file>`** deve ser substituído pelo nome do arquivo fonte:

   `java -jar Compilers.jar ./programs/<file>.`

   3.2. Para executar programas que não estão dentro da pasta **`programs`** é necessário que o caminho para o arquivo fonte seja informado corretamente em relação a pasta extraída.

   3.3. O compilador espera receber como parâmetro, opcionalmente, o parâmetro `-d` que ira ativar o modo de debug, que exibe informações sobre o processo de compilação.

   `java -jar Compilers.jar ./programs/<file> -d.`

4. Para a execução do compilador, é necessário que a versão java instalada na máquina seja a **19.0.1** ou superior.

Descrição da abordagem utilizada na implementação

A seguir, temos as principais classes criadas dentro do compilador que se referem a execução do analisador sintático:

Class Parser

A implementação de um Parser LL(1) com recuperação de erro na inserção, que é responsável por receber como entrada os tokens identificados pelo analisador e criar uma representação da estrutura gramatical da sequência de tokens (árvore sintática). Além disso, também verifica se a ordem dos tokens respeita a sintaxe da linguagem.

Gramática da Linguagem

A seguir, temos a gramática da linguagem fornecida inicialmente:

```
program'        ::= program $
program         ::= program identifier begin [decl-list] stmt-list end "."
decl-list       ::= decl ";" { decl ";"}
decl            ::= ident-list is type
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 |"!" factor | - factor

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

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

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

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

constant ::= integer_const | float_const | char_const

digit ::= [0-9]

carac ::= *um dos caracteres ASCII*

caractere ::= *um dos caracteres ASCII, exceto quebra de linha*

integer_const ::= digit $^{+}$

float_const ::= digit $^{+}$ "." digit $^{+}$

char_const ::= "'" carac "'"

```
literal         ::= "{" caractere* "}"
identifier      ::= letter (letter | digit | "_")*
letter          ::= [A-Za-z]
```

Podemos observar pelas marcações acima, que a gramática possui alguns erros sintáticos que necessitam ser tratados para que seja possível a construção do compilador. A seguir temos listados os erros:

1. `if-stmt`: Prefixo comum;
2. `expression`: Prefixo comum;
3. `simple-expr`: Recursão a esquerda;
4. `term`: Recursão a esquerda;

Além disso, adicionamos mais uma regra para definir a primeira expressão da linguagem:

```
program'        ::= program $
program         ::= program identifier begin [decl-list] stmt-list end "."
decl-list       ::= decl ";" { decl ";"}
decl            ::= ident-list is type
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 if-stmt'
if-stmt'        ::= end | 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 expression'

expression' ::= relop simple-expr| λ

simple-expr ::= term simple-expr'

simple-expr' ::= simple-expr addop simple-expr' | λ

term ::= factor-a term'

term' ::= mulop factor-a term' | λ

factor-a ::= factor |"!" factor | - factor

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

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

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

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

constant ::= integer_const | float_const | char_const

digit ::= [0-9]

carac ::= *um dos caracteres ASCII*

caractere ::= *um dos caracteres ASCII, exceto quebra de linha*

integer_const ::= digit +

float_const ::= digit + "." digit +

char_const ::= "'" carac "`"

literal ::= "{" caractere* "}"

identifier ::= letter (letter | digit | "_")*

letter ::= [A-Za-z]

Tabelas

A seguir, temos as tabelas para a construção do analisador sintático:

Tabela First/Follow

SÍMBOLO TERMINAL	FIRST	FOLLOW
program'	program	$
program	program	$
decl-list	a-z, A-Z	a-z, A-Z, if, while, repeat, read, write, ;
decl	a-z, A-Z	;
ident-list	a-z, A-Z	is
type	int, float, char	;
stmt-list	a-z, A-Z, if, while, repeat, read, write	end, else, until, ;
stmt	a-z, A-Z, if, while, repeat, read, write	;, a-z, A-Z, if, while, repeat, read, write
assign-stmt	a-z, A-Z	;, a-z, A-Z, if, while, repeat, read, write
if-stmt	if	;, a-z, A-Z, if, while, repeat, read, write

if-stmt’	end, else	end, else
condition	a-z, A-Z, 0-9, ’, (, !, -	then, ;, a-z, A-Z, if, while, repeat, read, write, do
repeat-stmt	repeat	;, a-z, A-Z, if, while, repeat, read, write
stmt-suffix	until	;, a-z, A-Z, if, while, repeat, read, write
while-stmt	while	;, a-z, A-Z, if, while, repeat, read, write
stmt-prefix	while	a-z, A-Z, if, while, repeat, read, write
read-stmt	read	;, a-z, A-Z, if, while, repeat, read, write
write-stmt	write	;, a-z, A-Z, if, while, repeat, read, write
writable	a-z, A-Z, 0-9, ’, (, !, -, {	)
expression	a-z, A-Z, 0-9, ’, (, !, -	), then, ;, a-z, A-Z, if, while, repeat, read, write, do
expression’	=, >, <, !, λ	), then, ;, a-z, A-Z, if, while, repeat, read, write, do
simple-expr	a-z, A-Z, 0-9, ’, (, !, -	), =, >, <, !,), then, ;, a-z, A-Z, if, while, repeat, read, write, do, +, -, |
simple-expr’	a-z, A-Z, 0-9, ’, (, !, -, λ	), =, >, <, !,), then, ;, a-z, A-Z, if, while, repeat, read, write, do, +, -, |
term	a-z, A-Z, 0-9, ’, (, !, -	a-z, A-Z, 0-9, ’, (, !,), =, >, <, !,), then, ;, if, while, repeat, read, write, do, +, -, |
term’	*, /, &, λ	a-z, A-Z, 0-9, ’, (, !,), =, >, <, !,), then,

		;, if, while, repeat, read, write, do, +, -, |
factor-a	a-z, A-Z, 0-9, ', (, !, -	*, /, &, a-z, A-Z, 0-9, ', (, !,), =, >, <, !,), then, ;, if, while, repeat, read, write, do, +, -, |
factor	a-z, A-Z, 0-9, ', (	*, /, &, a-z, A-Z, 0-9, ', (, !,), =, >, <, !,), then, ;, if, while, repeat, read, write, do, +, -, |
relop	=, >, <, !	a-z, A-Z, 0-9, ', (, !, -
addop	+, -, |	a-z, A-Z, 0-9, ', (, !, -,), =, >, <, !,), then, ;, a-z, A-Z, if, while, repeat, read, write, do, +, -, |
mulop	*, /, &	a-z, A-Z, 0-9, ', (, !, -
constant	0-9, '	0-9, `
digit	0-9	0-9, ., a-z, A-z, _,
carac	*um dos caracteres ASCII*	'
caractere	*um dos caracteres ASCII, exceto quebra de linha*	*um dos caracteres ASCII, exceto quebra de linha*, },
integer_const	0-9	0-9, `
float_const	0-9	0-9, `
char_const	'	0-9, `
literal	{	)
identifier	a-z, A-Z	*, /, &, a-z, A-Z, 0-9, ', (, !,), =, >, <, !,), then, ;, if, while, repeat, read, write, do, +, -, |, `,', begin, is
letter	a-z, A-Z	a-z, A-Z, 0-9, _

Programas e Resultados

A seguir, temos os códigos fontes utilizados para a realização do analisador sintático, e podemos ver a sequência de expressões identificadas:

program teste1

```
 1   programa teste1
 2
 3       a, b is int;
 4       result is int;
 5       a,x is float;
 6
 7   begin
 8
 9       a = 12a;
10       x = 12.1;
11       read (a);
12       read (b);
13       read (c)
14       result = (a*b + 1) / (c+2);
15       write {Resultado: };
16       write (result);
17
18   end.
```

O programa acima foi executado pelo analisador sintático e suas saídas foram a seguinte:

Teste 1:

```
utils.CompilerException: Linha: 1
Erro: (EAT) Erro na leitura do token: Lexeme: programa | Tag:ID Token
esperado: PROGRAM
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 1, onde foi lido a Tag “PROGRAMA” porém o esperado era “PROGRAM”. Sendo assim, agora temos a seguinte saída:

Teste 1 (Correção 1):

```
utils.CompilerException: Linha: 3
Erro:
```

```
(EAT) Erro na leitura do token: Lexeme: a | Tag:ID
Token esperado: BEGIN
```

Conforme identificado pela saída, foi encontrado mais um erro na leitura do token, dessa vez na linha 3, onde foi lido o token "a" mas era esperado o token "BEGIN". Sendo assim, agora temos a seguinte saída:

Teste 1 (Correção 2):

```
utils.CompilerException: Linha: 7
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: begin | Tag:BEGIN
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 7, pois havia um "BEGIN" em uma posição errônea. Sendo assim, agora temos a seguinte saída:

Teste 1 (Correção 3):

```
utils.CompilerException: Linha: 7
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: a | Tag:ID
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 7, pois havia um caractere adicional "a" junto a uma variável numérica, o que não é permitido pela gramática. Sendo assim, agora temos a seguinte saída:

Teste 1 (Correção 4):

```
utils.CompilerException: Linha: 12
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: result | Tag:ID
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 12, pois faltava o caractere ";" para fechar a declaração da variável. Sendo assim, agora temos a seguinte saída:

Teste 1 (Correção 5):

```
utils.CompilerException: Linha: 13
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: Resultado:  | Tag:LITERAL
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 13, pois faltavam os caracteres "(" e ")" para que a variável seja declarada da forma esperada pela gramática. Após essa alteração, o código foi analisado sintaticamente sem erros.

program teste2

```
 1   program teste2
 2
 3           a, b, c:int;
 4           d, var_: float;
 5
 6       teste2 = 1;
 7       Read (a);
 8       b = a * a;
 9       c = b + a/2 * (35/b);
10       write c;
11       val := 34.2
12       c = val + 2.2 + a;
13       write (val)
14   end.
```

Teste 2:

```
utils.CompilerException: Linha: 3
Erro:
(EAT) Erro na leitura do token: Lexeme: a | Tag:ID
 Token esperado: BEGIN
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 3, pois o identificador esperado era "BEGIN", mas o encontrado foi "a". Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 1):

```
utils.CompilerException: Linha: 3
Erro:
(STMT LIST) Erro na leitura do token: COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 3, pois o identificador esperado era ":", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 2):

```
utils.CompilerException: Linha: 4
Erro:
(STMT LIST) Erro na leitura do token: COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 4, pois o identificador esperado era ":", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 3):

```
utils.CompilerException: Linha: 7
Erro:
(STMT LIST) Erro na leitura do token: OPEN_PAR
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 7, pois o identificador esperado era "read", porém foi encontrado "Read". Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 4):

```
utils.CompilerException: Linha: 10
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: c | Tag:ID
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 10, pois o identificador esperado era "(", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 5):

```
utils.CompilerException: Linha: 10
Erro:
(STMT LIST) Erro na leitura do token: SEMI_COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 10, pois o identificador esperado era ")", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 6):

```
utils.CompilerException: Linha: 11
Erro:
(STMT LIST) Erro na leitura do token: COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 11, pois o identificador esperado era "=", porém foi encontrado um ":" o antecedendo. Sendo assim, agora temos a seguinte saída:

Teste 2 (Correção 7):

```
utils.CompilerException: Linha: 12
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: c | Tag:ID
```

Teste 2 (Correção 8):

```
utils.CompilerException: Linha: 14
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: end | Tag:END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 14, pois o identificador esperado era ";", porém não foi encontrado "end". Após essa alteração, o código foi analisado sintaticamente sem erros.

program UNDEFINED

```
 1   program
 2       begin
 3           a, aux is int;
 4           b is float
 5
 6           b = 0;
 7           in (a);
 8           in(b);
 9           if (a>b) then //troca variaveis
10               aux = b;
11               b = a;
12               a = aux
13           write(a;
14           write(b)
15       end;
```

Undefined:

```
utils.CompilerException: Linha: 2
Erro:  (EAT) Erro na leitura do token: Lexeme: begin | Tag:BEGIN
 Token esperado: ID
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 2, pois o identificador esperado era o nome do arquivo, porém foi encontrado "BEGIN". Sendo assim, agora temos a seguinte saída:

Undefined (Correção 1):

```
utils.CompilerException: Linha: 6
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: b | Tag:ID
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 6, pois o identificador esperado era ";" ao fim da declaração da variável, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Undefined (Correção 2):

```
utils.CompilerException: Linha: 7
Erro:
(STMT LIST) Erro na leitura do token: OPEN_PAR
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 7, pois o identificador esperado era "read", porém foi encontrado "in". Sendo assim, agora temos a seguinte saída:

Undefined (Correção 3):

```
utils.CompilerException: Linha: 13
Erro:
(STMT LIST) Erro na leitura do token: SEMI_COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 13, pois o identificador esperado era ";" ao fim da atribuição da variável, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Undefined (Correção 4):

```
utils.CompilerException: Linha: 15
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: end | Tag:END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 15, mas na verdade o erro foi na linha 13 pois o identificador esperado era "end" ao fim da execução do if, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Undefined (Correção 5):

```
utils.CompilerException: Linha: 14
Erro:
(STMT LIST) Erro na leitura do token: SEMI_COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 14, pois o identificador esperado era “)” ao fim da chamada da função, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Undefined (Correção 6):

```
utils.CompilerException: Linha: 16
Erro:
(EAT) Erro na leitura do token: SEMI_COLON
 Token esperado: DOT
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 16, pois o identificador esperado era “.” logo após o identificador end, porém foi encontrado “;”. Após essa alteração, o código foi analisado sintaticamente sem erros.

program teste4

```
 1   programa teste4
 2
 3   /* Teste4 do meu compilador */
 4
 5       pontuacao, pontuacaoMaxina, disponibilidade is inteiro;
 6       pontuacaoMinima is char;
 7
 8   begin
 9       pontuacaoMinima = 50;
10       pontuacaoMaxima = 100;
11       write({Pontuacao do candidato: });
12       read(pontuacao);
13       write({Disponibilidade do candidato: });
14       read(disponibilidade);
15
16       while (pontuacao>0 && (pontuacao<=pontuacaoMaxima) do
17           if ((pontuação > pontuacaoMinima) && (disponibilidade==1)) then
```

```
 18              write({Candidato aprovado.})
 19          else
 20              write({Candidato reprovado.})
 21          end
 22
 23          write({Pontuacao do candidato: });
 24          read(pontuacao);
 25          write({Disponibilidade do candidato: });
 26          read(disponibilidade);
 27      end
 28  end
```

Teste 4:

```
utils.CompilerException: Linha: 1
Erro:
(EAT) Erro na leitura do token: Lexeme: programa | Tag:ID
 Token esperado: PROGRAM
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 1, pois o identificador esperado era “PROGRAM”, porém foi encontrado “PROGRAMA”. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 1):

```
utils.CompilerException: Linha: 5
Erro:
(EAT) Erro na leitura do token: Lexeme: pontuacao | Tag:ID
 Token esperado: BEGIN
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 5, pois o identificador esperado era “BEGIN”, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 2):

```
utils.CompilerException: Linha: 5
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: inteiro | Tag:ID
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 5, pois o identificador esperado era a palavra reservada “int”, porém foi encontrado “inteiro”. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 3):

```
utils.CompilerException: Linha: 8
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: begin | Tag:BEGIN
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 8, pois foi encontrado o identificador ”BEGIN” fora do local adequado. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 4):

```
utils.CompilerException: Linha: 15
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: do | Tag:DO
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 15, pois o identificador esperado era “)” ao fim da chamada do “while”, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 5):

```
utils.CompilerException: Linha: 18
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: else | Tag:ELSE
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 18, pois o identificador esperado era ";" ao fim da chamada da função, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 6):

```
utils.CompilerException: Linha: 20
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: end | Tag:END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 20, pois o identificador esperado era ";" ao fim da chamada da função, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 4 (Correção 7):

```
utils.CompilerException: Linha: 27
Erro:
(EAT) Erro na leitura do token: END_OF_FILE Token esperado: DOT
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 27, pois o identificador esperado era "." logo após o identificador end, porém não foi encontrado. Após essa alteração, o código foi analisado sintaticamente sem erros.

program teste5

```
 1   /* Teste do meu compilador */
 2
 3   program teste5
 4       a, b, c, maior is int;
 5       outro is char;
 6
 7   begin
 8       repeat
 9           write({A});
10           read(a);
11           write({B});
```

```
 12           read(b);
 13           write({C});
 14           read(c);
 15
 16
 17
 18           if ( (a>b) && (a>c) ) end
 19               maior = a
 20
 21           else
 22               if (b>c) then
 23                   maior = b;
 24
 25               else
 26                   maior = c
 27               end
 28           end;
 29           write({Maior valor:});
 30           write (maior);
 31           write ({Outro? (S/N)});
 32           read(outro);
 33       until (outro == 'N' || outro == 'n')
 34   end
```

Teste 5:

```
utils.CompilerException: Linha: 18
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: end | Tag:END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 18, pois o identificador esperado era “THEN”, porém foi encontrado “END”. Sendo assim, agora temos a seguinte saída:

Teste 5 (Correção 1):

```
utils.CompilerException: Linha: 21
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: else | Tag:ELSE
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 21, pois o identificador esperado era ";", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 5 (Correção 2):

```
utils.CompilerException: Linha: 27
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: end | Tag:END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 27, pois o identificador esperado era ";", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

Teste 5 (Correção 3):

```
utils.CompilerException: Linha: 28
Erro:
(STMT LIST) Erro na leitura do token: SEMI_COLON
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 28, pois o identificador esperado não era ";", pois após a instrução de condição, não é esperado o token ";". Após essa alteração, o código foi analisado sintaticamente sem erros. Sendo assim, agora temos a seguinte saída:

Teste 5 (Correção 4):

```
utils.CompilerException: Linha: 34
Erro:
(EAT) Erro na leitura do token: END_OF_FILE
 Token esperado: DOT
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 34, pois o identificador esperado era “DOT”, porém foi encontrado “END_OF_FILE”. Após essa alteração, o código foi analisado sintaticamente sem erros.

program order

```
 1   program order
 2
 3     num1, num2, temp is int;
 4
 5     read(num1);
 6     read(num2);
 7
 8     if num1 > num2 then
 9       temp is num1;
10       num1 is num2;
11       num2 is temp;
12     end;
13
14     write({Menor: });
15     write(num1);
16     write({Maior: });
17     write(num2);
18   end.
```

order:

```
utils.CompilerException: Linha: 3
Erro:
(EAT) Erro na leitura do token: Lexeme: num1 | Tag:ID
 Token esperado: BEGIN
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 3, pois o identificador esperado era “BEGIN”, porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

order (Correção 1):

```
utils.CompilerException: Linha: 9
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 9, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

order (Correção 2):

```
utils.CompilerException: Linha: 10
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 10, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

order (Correção 3):

```
utils.CompilerException: Linha: 11
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 11, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

order (Correção 4):

```
utils.CompilerException: Linha: 12
Erro:
(EAT) Erro na leitura do token: SEMI_COLON
 Token esperado: END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 12, pois o identificador esperado não era ";", pois após a instrução de condição, não é esperado o

token ";". Após essa alteração, o código foi analisado sintaticamente sem erros. Após essa alteração, o código foi analisado sintaticamente sem erros.

program factorial

```
 1   program factorial
 2
 3     num, fact is int;
 4
 5     read(num);
 6
 7     fact is 1;
 8     i is 1;
 9
10     while i <= num do
11       fact is fact * i;
12       i is i + 1;
13     end;
14
15     write(fact);
16   end.
```

factorial:

```
utils.CompilerException: Linha: 3
Erro:
(EAT) Erro na leitura do token: Lexeme: num | Tag:ID
 Token esperado: BEGIN
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 3, pois o identificador esperado era "BEGIN", porém não foi encontrado. Sendo assim, agora temos a seguinte saída:

factorial (Correção 1):

```
utils.CompilerException: Linha: 8
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 8, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

factorial (Correção 2):

```
utils.CompilerException: Linha: 9
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 9, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

factorial (Correção 3):

```
utils.CompilerException: Linha: 12
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 12, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

factorial (Correção 4):

```
utils.CompilerException: Linha: 13
Erro:
(STMT LIST) Erro na leitura do token: Lexeme: is | Tag:IS
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 13, pois o identificador esperado era "=", porém foi encontrado "IS". Sendo assim, agora temos a seguinte saída:

factorial (Correção 5):

```
utils.CompilerException: Linha: 14
Erro:
(EAT) Erro na leitura do token: SEMI_COLON
 Token esperado: END
```

Conforme identificado pela saída, foi encontrado um erro na leitura do token na linha 14, pois o identificador esperado não era ";", pois após a instrução de repetição, não é esperado o token ";". Após essa alteração, o código foi analisado sintaticamente sem erros. Após essa alteração, o código foi analisado sintaticamente sem erros.

Conclusão

Em conclusão, a implementação do analisador sintático como parte deste trabalho foi bem-sucedida, alcançando os objetivos propostos. O analisador demonstrou ser eficiente na análise da estrutura gramatical de sentenças, fornecendo resultados precisos e contribuindo para uma melhor compreensão da sintaxe e semântica da linguagem em estudo. Além disso, a utilização de técnicas de análise descendente e tabelas de análise mostrou-se adequada, permitindo a identificação correta de elementos-chave, como sujeito, verbo e complementos.

A partir deste trabalho, novas melhorias e aprimoramentos podem ser explorados, como a incorporação de regras adicionais da gramática e a implementação de técnicas de análise mais avançadas. No geral, a implementação do analisador sintático representa um avanço significativo no campo da linguística computacional e contribui para o desenvolvimento de soluções mais eficientes e precisas na análise de estruturas linguísticas.