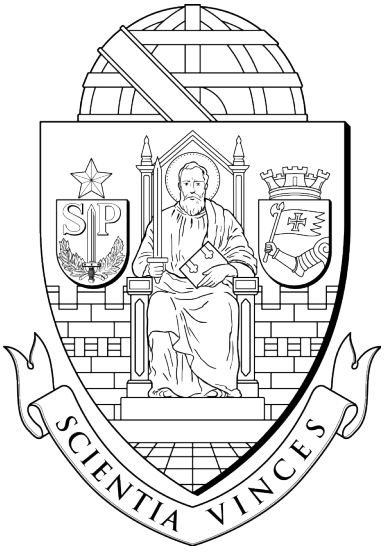

Universidade de São Paulo

Teoria da Computação e Compiladores

Analisador Sintático P–

Alunos:
Gabriel Guimarães Vilas Boas Marin - 11218521
Gustavo Schimiti - 7564002
Henrique Gomes Zanin - 10441321
Igor Guilherme Pereira Loredo - 11275071

Sumário

1 Introdução

Este trabalho tem como objetivo implementar um compilador para processar a linguagem P–, inspirada no Pascal. O compilador elaborado foi implementado com a linguagem Java e é composto por um analisador léxico e um analisador sintático. Os tópicos seguintes descreverão as informações referentes ao analisador sintático em detalhes de implementação, os grafos sintáticos utilizados e as regras para compilação do código.

2 Implementação

Escolheu-se a linguagem Java para a implementação do compilador, por se tratar de uma linguagem familiar a todos os membros do grupo. A escolha de uma linguagem orientada a objetos foi conveniente por possibilitar o uso de polimorfismo, permitindo uma boa fluidez no trabalho em equipe.

Outra vantagem do uso de Java e orientação a objetos é a possibilidade de transcrição para linguagens que seguem o mesmo paradigma, como GO, Python, C++ e outras, para suprir necessidades vigentes.

No âmbito das classes implementadas, a classe *Compiler* é responsável por passar a linha(fita) do programa em P– para o analisador léxico. Para simplificar a implementação e ser fiel às sugestões, criou-se uma classe chamada "Tape" que representa uma fita de entrada nos autômatos. A cada nova linha, a fita é substituída e enviada ao analisador léxico que devolve os *tokens* processados.

O analisador léxico corresponde à classe *LexicalAnalyzer*. Sua função é processar e validar os *tokens* do código fonte. O construtor da classe inicializa todos os autômatos em um *ArrayList* que é percorrido enquanto houver caracteres na fita de entrada. Assim, podemos inserir quantos autômatos forem necessários para interpretar os *tokens*.

Optou-se também por não criar um único autômato que incorporaria todos os autômatos necessários para o processamento dos símbolos. Essa decisão foi tomada visando uma melhor coordenação do trabalho em equipe. Entretanto, por conta dessa decisão, houve a necessidade de criar um autômato a mais, chamado *WhitelistAutomaton* para processar os *tokens* inválidos.

A estrutura utilizada no projeto permite a depuração de estruturas isoladas do compilador, permitindo a execução exclusiva do analisador léxico, analisador sintático ou analisador sintático com tratamento de erros com o modo pânico.

2.1 Grafos sintáticos

A seguir, serão apresentadas os grafos sintáticos utilizados na implementação do analisador sintático, responsáveis pelo processamento das cadeias de símbolos fornecidas pelo analisador léxico válidas na linguagem P–.

2.1.1 programa

O grafo sintático de **programa** é descrito por:

$$< programa > ::= \ program \ ident \ ; \ < corpo > \ .$$

Tomando o grafo sintático de **corpo**, descrito por:

$$< corpo > ::= < dc > \ begin \ < comandos > \ end$$

É possível gerar um grafo reduzido para **programa**, representado na Figura 1. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **programa**.

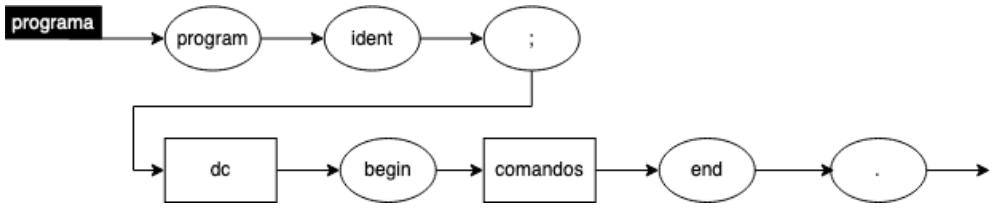

Figura 1: Grafo Sintático - programa

2.1.2 dc

O grafo sintático de **dc** é descrito por:

$$< dc > ::= < dc\_c > < dc\_v > < dc\_p >$$

Devido à complexidade dos grafos sintáticos de **dc_c**, **dc_v** e **dc_p**, optou-se por manter o grafo de **dc** sem redução. Com isso, ele foi implementada no projeto de acordo com a representação da Figura 2.

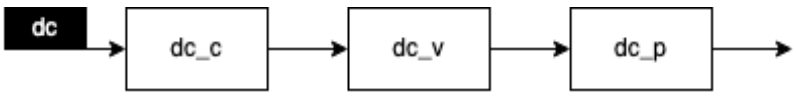

Figura 2: Grafo Sintático - dc

2.1.3 dc_c

O grafo sintático de **dc_c** é descrito por:

$$< dc\_c > ::= \ const\ ident\ = < numero > ; < dc\_c > \ | \ \lambda$$

Tomando o grafo sintático de **numero**, descrito por:

$$< numero > ::= < numero\_int > \ | \ < numero\_real >$$

É possível gerar um grafo reduzido para **dc_c**, representado na Figura 3. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **dc_c**.

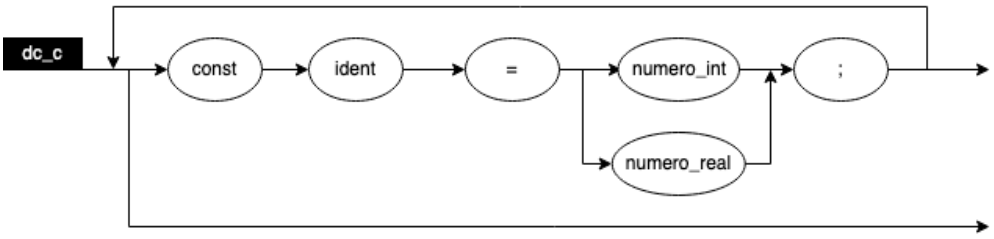

Figura 3: Grafo Sintático - dc_c

2.1.4 dc_v

O grafo sintático de **dc_v** é descrito por:

$$< dc\_v > ::= \ var \ < variaveis > : < tipo\_var > ; < dc\_v > \ | \ \lambda$$

Tomando os grafos sintáticos de **tipo_var**, **variaveis** e **mais_var** descritos por:

$$< tipo\_var > ::= \ real \mid integer$$

$$< variaveis > ::= \ ident \ < mais\_var >$$

$$< mais\_var > ::= , < variaveis > \ | \ \lambda$$

É possível gerar um grafo reduzido para **dc_v**, representado na Figura 4. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **dc_v**.

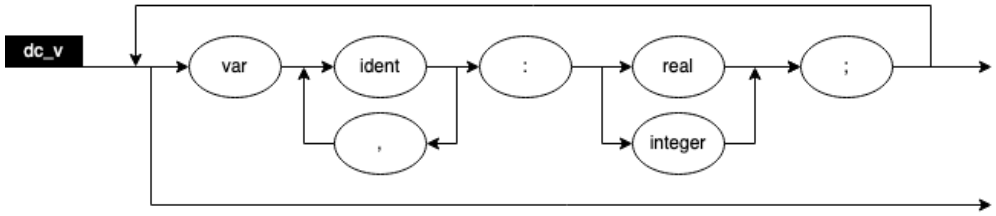

Figura 4: Grafo Sintático - dc_v

2.1.5 variaveis

O grafo sintático de **variaveis** é descrito por:

$$< variaveis > ::= < ident > < mais\_var >$$

Pode-se notar através da Figura 5 que não é possível realizar redução nele. Como ele não foi utilizado em reduções de todas as aparições em outros grafos sintáticos, foi realizada a implementação de sua função.

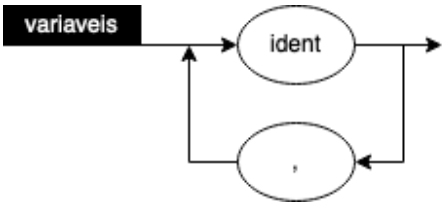

Figura 5: Grafo Sintático - variaveis

2.1.6 dc_p

O grafo sintático de **dc_p** é descrito por:

$$< dc\_p > ::= \ procedure\ ident\ \ < parametros > \ ; \ < corpo\_p > < dc\_p > \ | \ \lambda$$

Tomando o grafo sintático de **parametros**, descrito por:

$$< parametros > ::= \ (\ < lista\_par > \ ) \ | \ \lambda$$

É possível gerar um grafo reduzido para **dc_p**, representado na Figura 6. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **dc_p**.

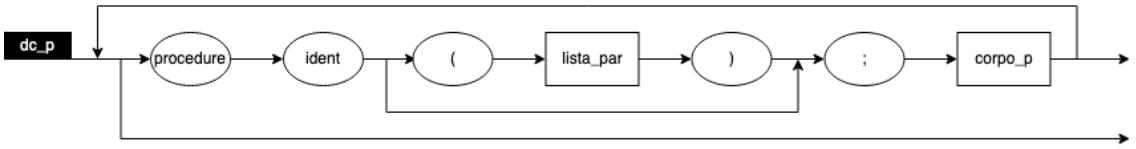

Figura 6: Grafo Sintático - dc_p

2.1.7 lista_par

O grafo sintático de **lista_par** é descrito por:

$$< lista\_par > ::= < variaveis > : < tipo\_var > < mais\_par >$$

Tomando o grafo sintático de **mais_par**, descrito por:

$$< mais\_par > ::= ; \ < lista\_par > \ | \ \lambda$$

É possível gerar um grafo reduzido para **lista_par**, representado na Figura 7. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **lista_par**.

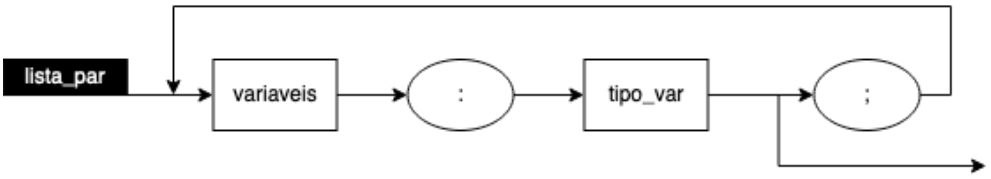

Figura 7: Grafo Sintático - lista_par

2.1.8 corpo_p

O grafo sintático de **corpo_p** é descrito por:

$$< corpo\_p > ::= < dc\_loc > \ begin \ < comandos > \ end \ ;$$

Tomando o grafo sintático de **dc_loc**, descrito por:

$$< dc\_loc > ::= < dc\_v >$$

É possível gerar um grafo reduzido para **corpo_p**, representado na Figura 8. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **corpo_p**.

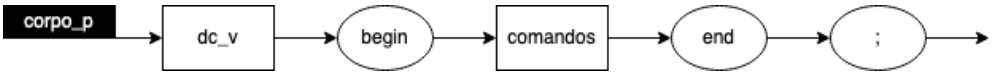

Figura 8: Grafo Sintático - corpo_p

2.1.9 tipo_var

O grafo sintático de **tipo_var** é descrito por:

$$< tipo\_var > ::= \ real \mid integer$$

Pode-se notar através da Figura 9 que não é possível realizar redução nele. Como ele não foi utilizado em reduções de todas as aparições em outros grafos sintáticos, foi realizada a implementação de sua função.

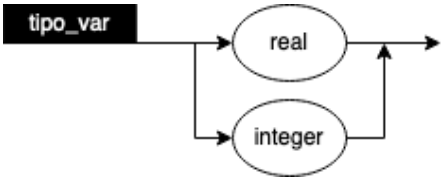

Figura 9: Grafo Sintático - tipo_var

2.1.10 argumentos

O grafo sintático de **argumentos** é descrito por:

$$< argumentos > ::= < ident > < mais\_ident >$$

Tomando o grafo sintático de **mais_ident**, descrito por:

$$< mais\_ident > ::= \ ; \ < argumentos > \ \mid \lambda$$

É possível gerar um grafo reduzido para **argumentos**, representado na Figura 10. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **argumentos**.

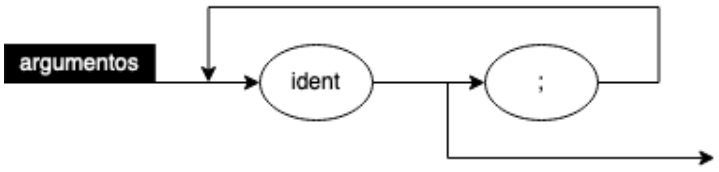

Figura 10: Grafo Sintático - argumentos

2.1.11 comandos

O grafo sintático de **comandos** é descrito por:

$$< comandos > ::= < cmd > ;\ comandos \mid \lambda$$

Devido à complexidade do grafo sintático de **cmd** optou-se por manter o grafo de **comandos** sem redução. Com isso, ele foi implementada no projeto de acordo com a representação da Figura 11.

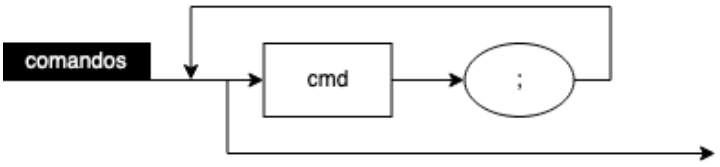

Figura 11: Grafo Sintático - comandos

2.1.12 cmd

O grafo sintático de **cmd** é descrito por:

$$\begin{aligned}
< cmd > \ &::= \ read\ (\ < variaveis >\ ) \mid \\
&::= \ write\ (\ < variaveis >\ ) \mid \\
&::= \ while\ (\ < condicao >\ )\ do\ \ < cmd >\ \mid \\
&::= \ if\ \ < condicao >\ \ then\ \ < cmd > < pfalsa >\ \mid \\
&::= \ ident\ \ := < expressao >\ \mid \\
&::= \ ident\ \ < lista_arg >\ \mid \\
&::= \ begin\ \ < comandos >\ \ end
\end{aligned}$$

Tomando os grafos sintáticos de **pfalsa** e **lista_arg** descritos por:

$$< pfalsa > ::= \ else\ \ < cmd >\ \mid \lambda$$

$$< lista\_arg > ::= \ (\ < argumentos >\ ) \mid \lambda$$

É possível gerar um grafo reduzido para **cmd**, representado na Figura 12. Apesar da redução realizada, o grafo foi segmentado em várias funções durante a implementação para facilitar o desenvolvimento e a depuração do código. Um grafo sintático para o comando **for** também foi incluído dentro de **cmd**.

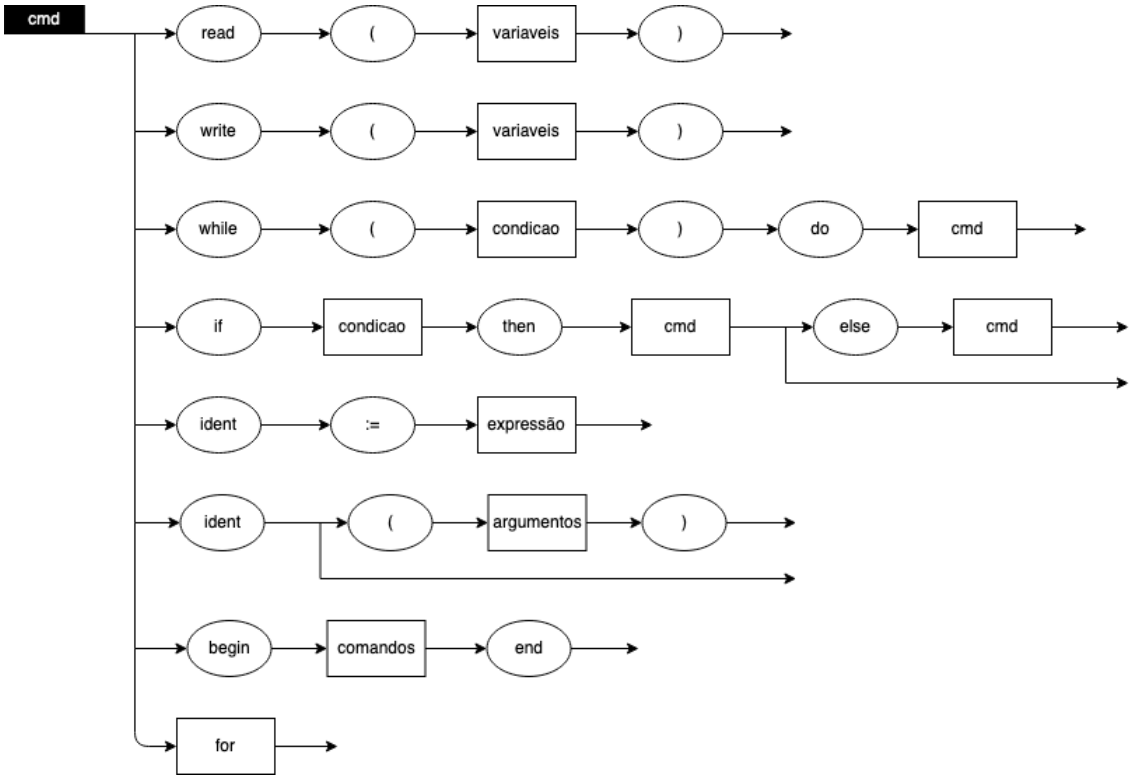

Figura 12: Grafo Sintático - cmd

2.1.13 condicao

O grafo sintático de **condicao** é descrito por:

$$< condicao > ::= < expressao > < relacao > < expressao >$$

Tomando o grafo sintático de **relacao**, descrito por:

$$< relacao > ::= = \mid >= \mid <= \mid > \mid <$$

É possível gerar um grafo reduzido para **condicao**, representado na Figura 13. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **condicao**.

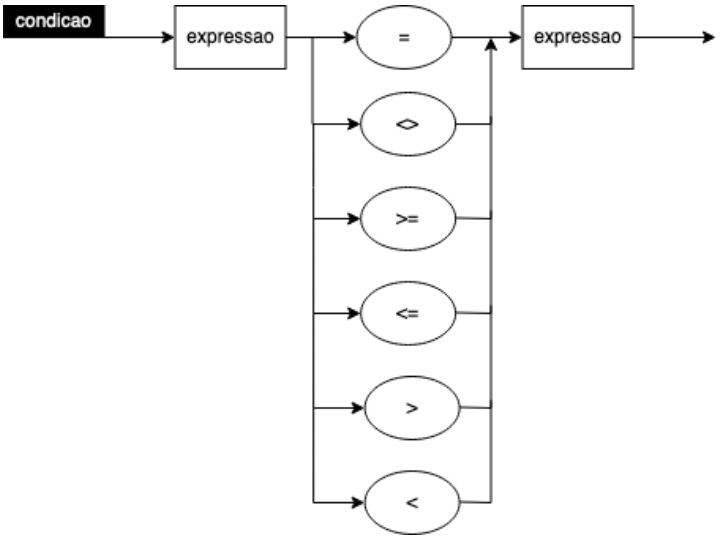

Figura 13: Grafo Sintático - condicao

2.1.14 expressao

O grafo sintático de **expressao** é descrito por:

$$< expressao > ::= < termo > < outros\_termos >$$

Optou-se por manter o grafo de **expressao** sem redução por conta das demais regras existentes dentro dos grafos de **termo** e **outros_termos**. Com isso, ele foi implementada no projeto de acordo com a representação da Figura 14.

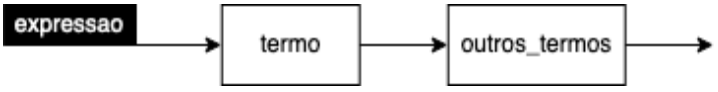

Figura 14: Grafo Sintático - expressao

2.1.15 outros_termos

O grafo sintático de **outros_termos** é descrito por:

$$< outros\_termos > ::= < op\_ad > < termo > < outros\_termos > \ | \ \lambda$$

Tomando o grafo sintático de **op_ad**, descrito por:

$$< op\_ad > ::= \ + \ | \ -$$

É possível gerar um grafo reduzido para **outros_termos**, representado na Figura 15. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **outros_termos**.

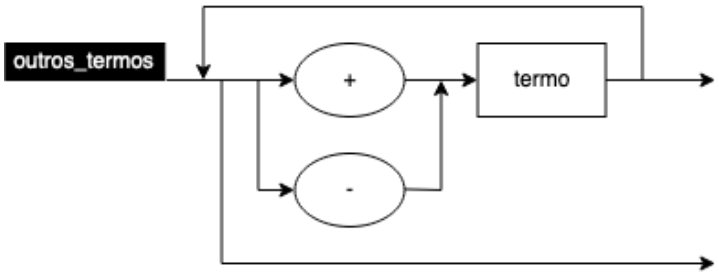

Figura 15: Grafo Sintático - outros_termos

2.1.16 termo

O grafo sintático de **termo** é descrito por:

$$< termo > ::= < op\_un > < fator > < mais\_fatores >$$

Tomando os grafos sintáticos de **op_un** e **op_ad** descritos por:

$$< op\_un > ::= < op_ad > \ | \ \lambda$$

$$< op\_ad > ::= \ + \ | \ -$$

É possível gerar um grafo reduzido para **termo**, representado na Figura 16. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **termo**.

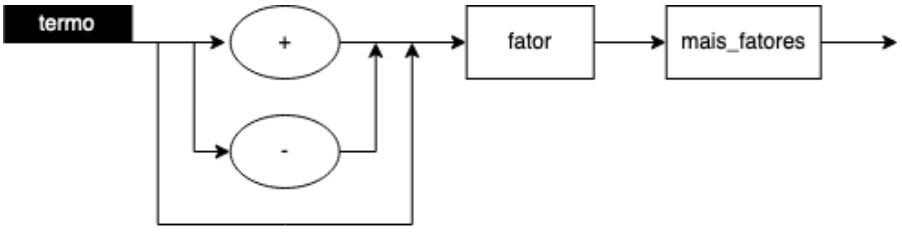

Figura 16: Grafo Sintático - termo

2.1.17 mais_fatores

O grafo sintático de **mais_fatores** é descrito por:

$$< mais\_fatores > ::= < op\_mul > < fator > < mais\_fatores > \ | \ \lambda$$

Tomando o grafo sintático de **op_mul**, descrito por:

$$< op\_mul > ::= \ * \ | \ /$$

É possível gerar um grafo reduzido para **mais_fatores**, representado na Figura 17. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **mais_fatores**.

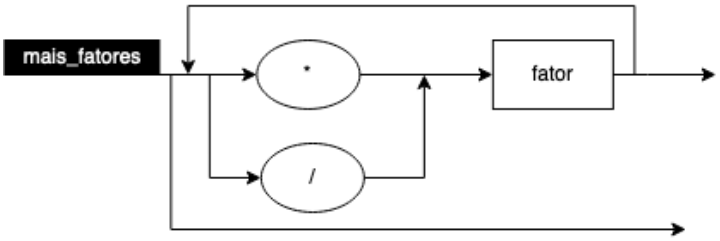

Figura 17: Grafo Sintático - mais_fatores

2.1.18 fator

O grafo sintático de **fator** é descrito por:

$$< fator > ::= \ ident \ | \ < numero > \ | \ (\ < expressao > \ )$$

Tomando o grafo sintático de **numero**, descrito por:

$$< numero > ::= \ numero\_int \ | \ numero\_real$$

É possível gerar um grafo reduzido para **fator**, representado na Figura 18. Esta abordagem foi utilizada no projeto para a implementação da função que trata as regras de **fator**.

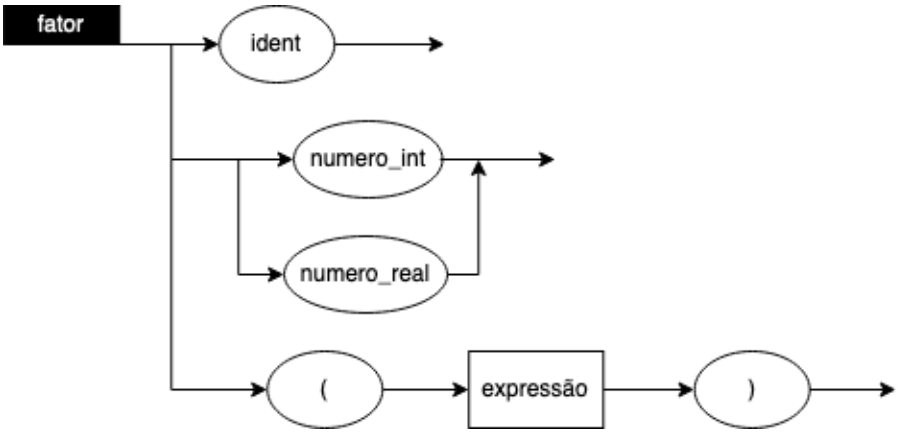

Figura 18: Grafo Sintático - fator

2.1.19 for

O grafo sintático de **for** é descrito por:

$$for \ ::= \ for \ ident \ := < expressao > \ to \ < expressao > \ do \ < cmd >$$

Optou-se por manter o grafo de **for** sem redução por conta das demais regras existentes dentro dos grafos de **expressao** e **cmd**. Com isso, ele foi implementada no projeto de acordo com a representação da Figura 19.

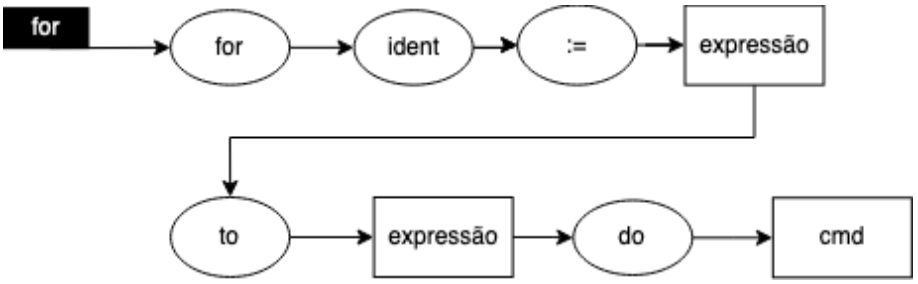

Figura 19: Grafo Sintático - for

2.2 Tabela de Símbolos

Símbolo	**ID**
program	sym_program
;	sym_semicolon
.	sym_dot
begin	sym_begin
end	sym_end
const	sym_const
var	sym_var
:	sym_colon
real	sym_real
integer	sym_int
procedure	sym_procedure
else	sym_else
read	sym_read
write	sym_write
while	sym_while
do	sym_do
if	sym_if
then	sym_then
=	sym_eq
<>	sym_notEq
>=	sym_greaterOrEq
<=	sym_lessOrEq
>	sym_greater
<	sym_less
+	sym_plus
-	sym_minus
*	sym_plus
/	sym_div
:=	sym_atrib
to	sym_to
,	sym_comma
(	sym_leftParenthesis
)	sym_rightParenthesis
for	sym_for

Tabela 1: Tabela de simbolos reservados

3 Execução do código

3.1 Compilação e Execução

Fornecemos junto com esse documento os códigos fonte do analisador lexico e sintático, bem como os arquivos bytecode java para execução. É possível testar isoladamente cada etapa de compilação, como: Analise léxica, analise sintática sem modo pânico e analise sintática com modo pânico. Todos os arquivos compilados estão na pasta *jar*.

Para compilar execute:

Código 1: Compilação

```
$ javac -d out/ sintatico/*.java

```

* Todos os arquivos .class do java estarão na pasta out

Código 2: Execução

```
$ java -classpath out/ Main testePequeno.p

```

* Troque o "testePequeno.p" pelo caso de teste desejado

Um arquivo do tipo jar já compilado está disponível no diretório "jar". Para executá-lo faça o seguinte processo:

Código 3: **Execução do JAR(aconselhado)**

```
$ java -jar sintaticoComPanico.jar teste/testePequeno.p
$ java -jar sintaticoSemPanico.jar testes/testePequeno.p
$ java -jar lexico.jar testes/testePequeno.p

```

3.2 Padrão de Entrada

A entrada aceita pelo código consiste em um arquivo .p contendo o programa escrito na linguagem P– a ser compilado.

3.3 Padrão de Saída

A saída produzida pela execução do código é exibida no console. Se a compilação resulta em erro, todos os erros encontrados são informados. Caso contrário, uma mensagem de sucesso é exibida.

4 Linguagem P–

1. <programa> ::= program ident ; <corpo> .
2. <corpo> ::= <dc> begin <comandos> end
3. <dc> ::= <dc_c> <dc_v> <dc_p>
4. <dc_c> ::= const ident = <numero> ; <dc_c> | λ
5. <dc_v> ::= var <variaveis> : <tipo_var> ; <dc_v> | λ
6. <tipo_var> ::= real | integer
7. <variaveis> ::= ident <mais_var>
8. <mais_var> ::= , <variaveis> | λ
9. <dc_p> ::= procedure ident <parametros> ; <corpo_p> <dc_p> | λ
10. <parametros> ::= (<lista_par>) | λ
11. <lista_par> ::= <variaveis> : <tipo_var> <mais_par>
12. <mais_par> ::= ; <lista_par> | λ
13. <corpo_p> ::= <dc_loc> begin <comandos> end ;
14. <dc_loc> ::= <dc_v>
15. <lista_arg> ::= (<argumentos>) | λ
16. <argumentos> ::= ident <mais_ident>
17. <mais_ident> ::= ; <argumentos> | λ
18. <pfalsa> ::= else <cmd> | λ
19. <comandos> ::= <cmd> ; <comandos> | λ
20. <cmd> ::=
    a. read (<variaveis>) |
    b. write (<variaveis>) |
    c. while (<condicao>) do <cmd> |
    d. if <condicao> then <cmd> <pfalsa> |
    e. ident := <expressão> |
    f. ident <lista_arg> |
    g. begin <comandos> end |
    h. <for>
21. <condicao> ::= <expressao> <relacao> <expressao>
22. <relacao> ::= = | <> | >= | <= | > | <
23. <expressao> ::= <termo> <outros_termos>
24. <op_un> ::=<op_ad> | λ
25. <outros_termos> ::= <op_ad> <termo> <outros_termos> | λ
26. <op_ad> ::= + | -
27. <termo> ::= <op_un> <fator> <mais_fatores>
28. <mais_fatores> ::= <op_mul> <fator> <mais_fatores> | λ
29. <op_mul> ::= * | /
30. <fator> ::= ident | <numero> | (<expressao>)
31. <numero> ::= numero_int | numero_real
32. <for> ::= ident := <expressão> to <expressão> do <cmd>