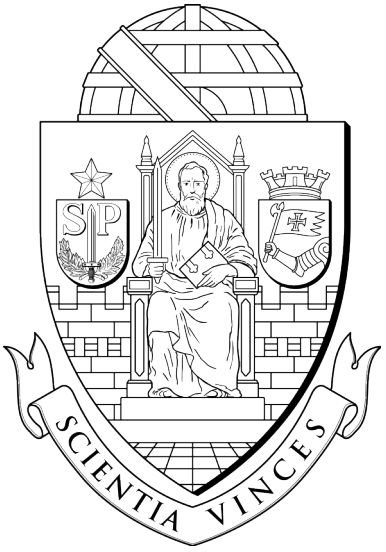

Universidade de São Paulo

Teoria da Computação e Compiladores

Analisador Léxico 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 a implementação em linguagem Java do analisador léxico, presente no *frontend* de um compilador, para processar a linguagem P– inspirada no Pascal. Nos tópicos seguintes serão demonstrados: todos os autômatos finitos determinísticos que processam os *tokens*, a tabela de símbolos reservados, bem como as regras para compilação do código.

2 Implementação

Escolheu-se a linguagem Java para a implementação do compilador unicamente por ser uma linguagem familiar a todos os membros do grupo. Foi unânime a escolha de uma linguagem orientada a objetos devido à presença de polimorfismo, o que permitiria uma boa fluidez no trabalho em equipe.

A depuração dos autômatos também foi simplificada pelo paradigma orientado a objetos por permitir a inserção e remoção de autômatos sem que houvesse prejuízo aos demais.

É evidente que uma linguagem como Java não é a melhor para a implementação de um compilador comercial, porém, todos os códigos podem ser facilmente transportados para as linguagens GO e C++, o que traria um grande ganho de performance.

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*.

Na análise dos *tokens*, executada pelos autômatos, decidiu-se por processar os caracteres inválidos individualmente. Um exemplo para a cadeia *be@gin* pode ser visto na Tabela 1.

Optou-se também por não criar um único autômato que incorporaria todos os autômatos descritos neste trabalho. Essa decisão foi tomada visando uma melhor coordenação do trabalho em equipe. Entretanto, por conta dessa decisão, houve

Símbolo	ID
be	ident
@	caractere inválido
gin	ident

Tabela 1: Simbolos para a cadeia be@gin

a necessidade de criar um autômato a mais, chamado *WhitelistAutomaton* para processar os *tokens* inválidos.

2.1 Autômatos Projetados

A seguir, apresentaremos as descrições das classes dos autômatos implementados para o funcionamento do analisador léxico, responsáveis pelo processamento de trechos inválidos ou contendo números, atribuições, operações relacionais, identificadores, comentários entre outros existentes na linguagem P–.

2.1.1 WhitelistAutomaton

A classe *WhitelistAutomaton*, como comentado anteriormente foi implementada para identificar *tokens* inválidos que podem existir nos programas P– testados. Ela retorna *null* para os caracteres válidos e o caractere inválido nos demais casos. O autômato está representado na Figura 1.

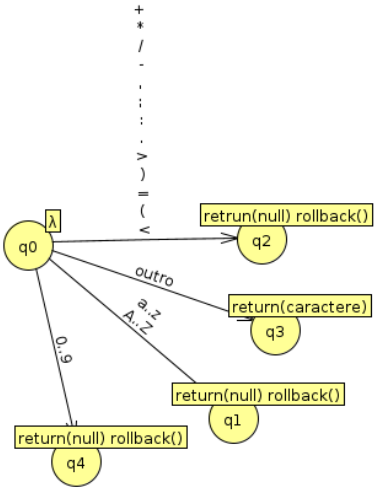

Figura 1: Whitelist

2.1.2 NumbersAutomaton

A classe *NumbersAutomaton* é responsável pela verificação dos trechos de código contendo números. Ela retorna um número inteiro, quando identifica a existência de um número inteiro ou um número real, caso identifique um número real. O autômato está representado na Figura 2.

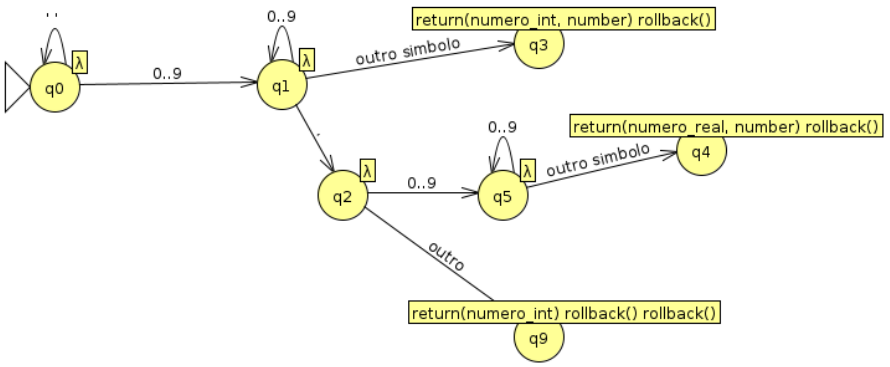

Figura 2: Numbers

2.1.3 AssignmentAutomaton

A classe *AssignmentAutomaton* é responsável por mapear o comportamento do autômato da Figura 3. Sua função é retornar o símbolo de atribuição ":=" ou o símbolo ":". Qualquer caractere diferente retorna *null* como valor.

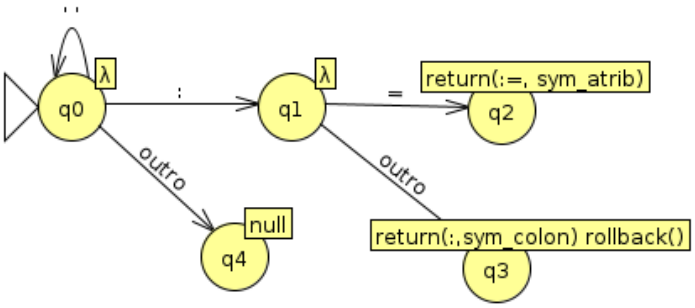

Figura 3: Assignment

2.1.4 RelationalAutomaton

Essa classe é responsável por processar todos os símbolos relacionais, qualquer caractere diferente dos exibidos na Figura 4 retorna *null* como valor.

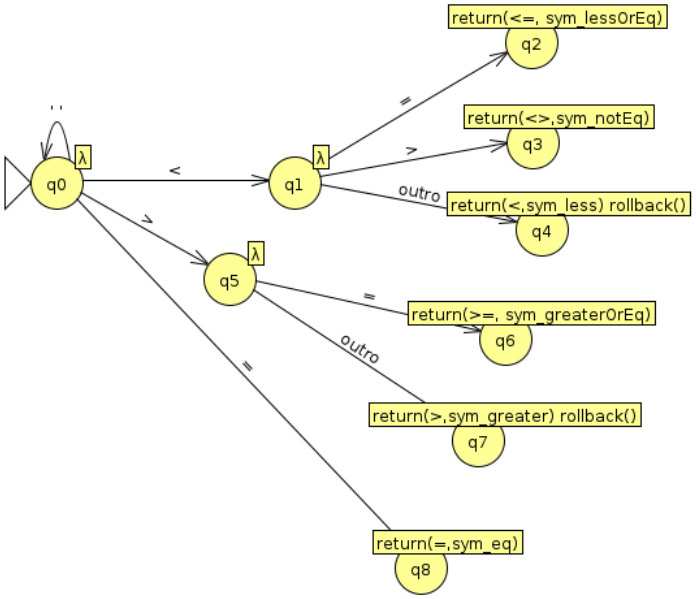

Figura 4: Relational

2.1.5 IdentifierAutomaton

Essa classe processa todos os identificadores válidos do código fonte. Palavras reservadas são capturadas após a execução desse autômato, sendo feita por meio da conferência da tabela de símbolos *symbolTable* presente na classe *LexicalAnalyzer*.

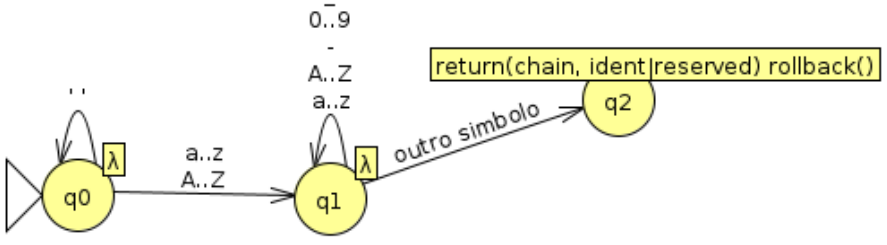

Figura 5: Identifier

2.1.6 CommentAutomaton

Responsável por processar as linhas de comentários. Cadeias de caracteres iniciadas com “{“ e terminadas com “}” são removidas do processo de compilação.

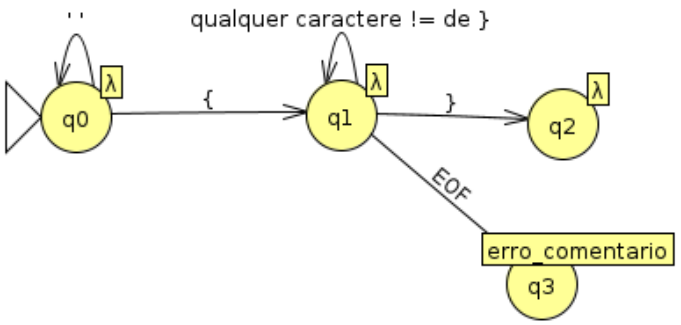

Figura 6: Comment

2.1.7 OtherSymbolsAutomaton

Sua função é resolver os demais símbolos reservados que não foram processados pelos outros autômatos. Como os demais, esse processo de verificação de símbolos reservados envolve a consulta da tabela definida na classe *LexicalAnalizer*

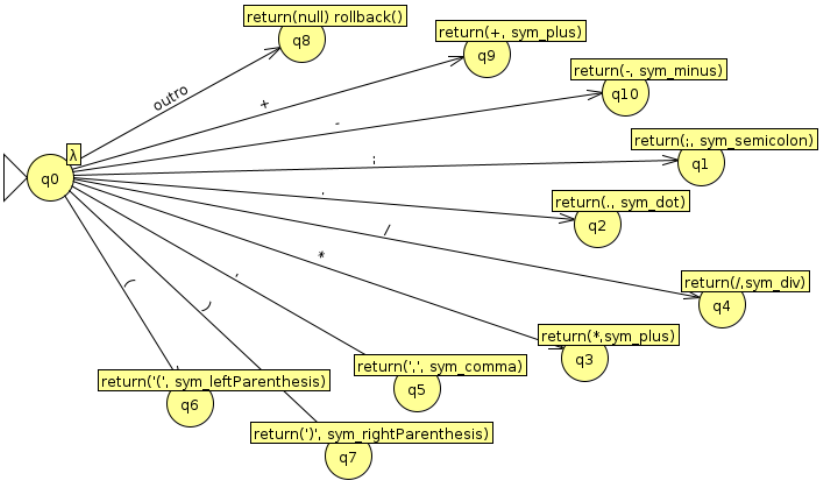

Figura 7: OtherSymbols

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 2: Tabela de simbolos reservados

3 Execução do código

3.1 Compilação e Execução

Para compilar execute:

Código 1: Compilação

```
$ javac -d out/ src/*.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(opcional)

```
$ java -jar pcompiler.jar testePequeno.p

```

3.2 Padrão de Entrada

A entrada aceita pela execução do código consiste em um arquivo .txt contendo o programa escrito na linguagem P– a ser analisado lexicalmente.

3.3 Padrão de Saída

A saída produzida pela execução do código é exibida no console, seguindo a convenção de saída do enunciado do projeto, contendo um par cadeia-token por linha, indicando os erros léxicos existentes.

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
21. <condicao> ::= <expressao> <relacao> <expressao>
22. <relacao> ::= = | <> | >= | <= | > | <
23. <expressao> ::= <termo> <outros_termos>
24. <op_un> ::= + | - | 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> <comandos>

Figura 8: p–