**Universidade Federal Rural de Pernambuco - UFRPE**

**Unidade Acadêmica de Garanhuns - UAG**

**Compiladores**

**Relatório do Compilador**

ALUNO: **Paulo Mateus da Silva**

PROFESSOR(A): **Maria Aparecida A. Sibaldo**

# Sumário

[Sumário: 2](#_Toc489955446)

[1. Analisador Léxico (Lexical.java) 3](#_Toc489955447)

[2. Analisador Sintático (Sintatic.java) 3](#_Toc489955448)

[3. Analisador Semântico (Semantic.java) 4](#_Toc489955449)

[4. Tradução para código de 3 endereços (IntermediateCodeGenerator.java) 5](#_Toc489955450)

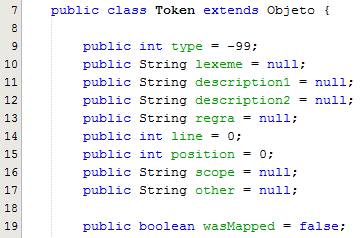
[5. Interface Gráfica (JFramePrincipal.java) 6](#_Toc489955451)

[6. Estrutura de Pacotes, dependências e github 9](#_Toc489955452)

[7. Gramática 10](#_Toc489955453)

# Analisador Léxico (Lexical.java)

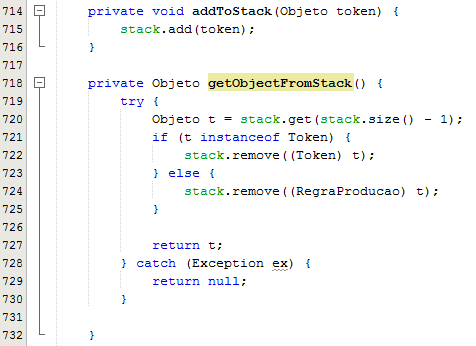
A separação dos lexemas foi feita da seguinte forma: tudo que começa com letra minúscula (letras e dígitos), é considerado um identificador, tudo que começa com dígito, e permanece com dígito, é considerado constante, a verificação por palavra reservada é feita e nos operadores (+, -, \*, /, <, >, <=, >=, !=, ==, &&, ||, (, ), [, ], {, }) foi dado espaçamento antes e depois, fazendo com que qualquer problema por o código fonte ter sido colocado junto é solucionado. Após isso, é feito um Split por espaço nessa string, logo, cada lexema terá uma posição diferente em um array, sem espaços adicionais. Neste mesmo momento, é obtido a linha, posição atual e escopo do lexema. Todas essas informações referentes ao lexema são guardadas em um objeto chamado Token. Tudo é salvo em um arraylist<Token> que será utilizado até o final do processo de compilação.



(Figura 1: classe Token com todos os seus atributos.)

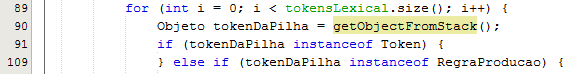
# Analisador Sintático (Sintatic.java)

É utilizado pilha, visto que seria mais fácil a derivação de uma regra de produção desta forma. O controle da pilha é feito por dois métodos: *addToStack* e *getObjectFromStack*, responsáveis por adicionar a pilha e obter/remover da pilha, respectivamente.



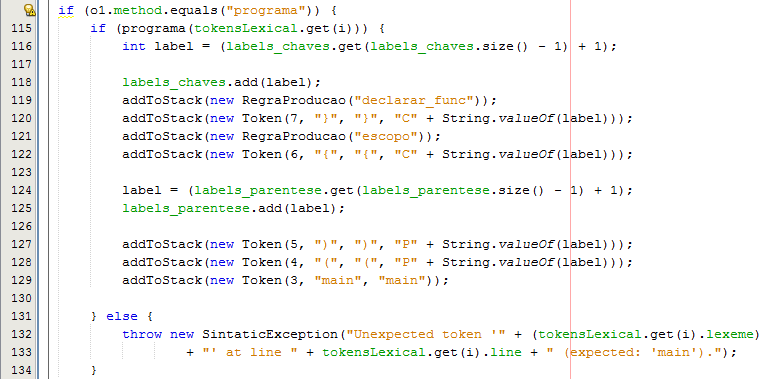
(Figura 2: métodos de controle da pilha.)

No método principal dessa classe, *Sintatic.java*, é verificado se o objeto atual da pilha é um *token* ou é uma *regra de produção*. Se for *token*, é feita uma conferência se aquele *token* corresponde ao *token* que realmente foi inserido na palavra (código fonte). Caso seja uma regra de produção, o fluxo do código será direcionado para que seja feita a derivação dessa regra.



(figura 3: momento em que é feito a comparação da pilha se objeto atual é uma regra de produção ou um terminal/token.)

Exemplificando como seria feita a derivação. Se na pilha, o objeto atual for a regra de produção “programa” (primeira regra a ser inserida em todos os casos), o código iria para esse fluxo:

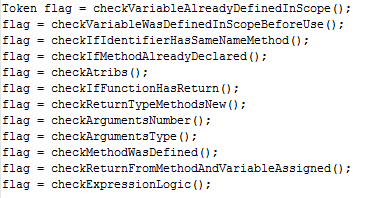


(Figura 4: momento que a regra de produção programa é lida, é sua derivação é feita.)

de forma que seria adicionado a pilha o *token* main, *token* (, *token* ), *token* {, regra de produção escopo, *token* } e regra de produção declarar\_func. Após cada iteração do for principal, eu corrijo o índice, caso seja uma regra de produção, para que a comparação entre o objeto atual da pilha e o *token* da palavra (código fonte) seja possível. Todas as regras de produção, obviamente, possuem sua derivação, ao longo da classe *Sintatic.java*. Após esse analisador, é garantido que a palavra (código fonte) respeita a gramática, porém, ainda pode ocorrer erros, como por exemplo, haver uma comparação 1 == true, ou true <= 2, etc... Ainda no analisador sintático, são adicionadas algumas informações pertinentes ao lexema no arraylist<Token> principal, que serão utilizadas no analisador semântico, como por exemplo a chave correspondente àquela chave, o else correspondente a qual if, e coisas do gênero.

# Analisador Semântico (Semantic.java)

As verificações no analisador semântico são feitas por tema, exemplificando, será verificado todas as expressões lógicas, depois será verificado todas as expressões aritméticas, depois será verificado a quantidade de parâmetros no momento da chamada de um método, depois será verificado se os tipos de argumentos passados correspondem aos parâmetros da definição, e assim sucessivamente. Isso é importante ressaltar, porque no caso de haver mais de um erro, o compilador pode informar a linha de erro posterior ao do primeiro erro. Utilizando o exemplo citado acima, seria assim: Na linha 10 há um erro na expressão aritmética, na linha 25 há um erro na expressão lógica, porém, como as expressões lógicas são verificadas antes das expressões aritméticas, o compilador apontará erro na linha 25, ao invés da linha 10 (onde o erro, para o programador, acontece antes). Os métodos de verificação são os seguintes:



(Figura 5: métodos de verificação do analisador semântico, em ordem cronológica.)

a ordem desses métodos não deve ser alterada, visto que em alguns métodos, algumas informações são salvas para os métodos posteriores. Todos esses métodos podem ser conferidos no método principal da classe *Semantic.java*:



(Figura 6: método principal do analisador semântico, contendo todas as verificações, e lançamentos de erros.)

# Tradução para código de 3 endereços (IntermediateCodeGenerator.java)

A tradução possui algumas peculiaridades, por exemplo, a condição do if|while é resolvida antes de ser colocada. Por exemplo, esse trecho de código , será traduzido como : , já que só é possível colocar 3 registradores, essa foi a melhor solução que pude encontrar. A resolução de condição, if|while, será feita de igual forma a resolução de expressões aritméticas, parênteses possuem precedência, depois <, >, <=, >=, ==, !=, após isso && e ||, respectivamente. Aproveitando, o ensejo, a precedência de expressões aritméticas são os parênteses, depois \*, /, + e -. Dessa forma consigo garantir a corretude da expressão lógica.

Outra peculiaridade é que o if + else é tratado de forma diferente de apenas o if *standalone*, isso porque o if+else é necessário dois labels para sua contrução, e o if *standalone*, é necessário apenas um label. Assim sendo, todas as aparições de ifs *standalone*, terá sua condição negada, caso seja positivo a condição, ou seja, a condição do if não é satisfeita, prossegue a instrução seguinte a do if, caso a condição seja falsa, ou seja, a condição do if será satisfeita, irá para o goto, livrando o fluxo de execução de executar o if. No if+else, após a instrução do if, todo o código correspondente ao else é alocado, após esta alocação, é adicionado um goto, para que o fluxo de execução não execute a parte do if, que só será executado caso o a condição do if seja verdadeira. Caso a condição do if seja verdadeira, vai para o goto, livrando o fluxo de execução de executar o código correspondente ao do else.

Outra peculiaridade, é que antes do encerramento da chave do while, as condições do while são atualizadas, isso para evitar erros de comparação. Por exemplo, no trecho de código , o i é utilizado para fazer a verificação, podendo ser alterado dentro do while, logo, no momento que for feito a comparação, a condição estará desatualizada, já que a resolução da condição é feita fora do if, como já citado anteriormente. Ao atualizar esse valor no final do while, garanto a corretude das comparações.

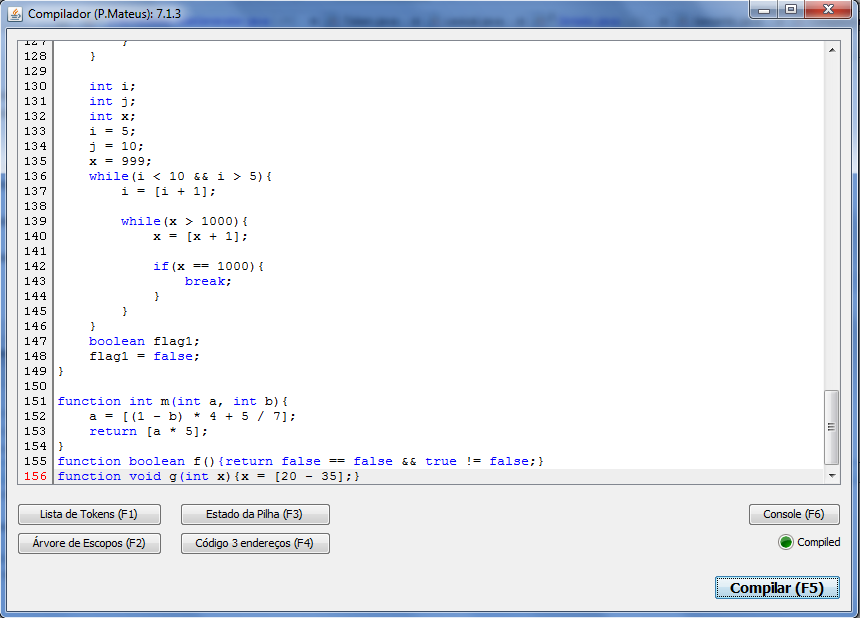
A tradução das expressões aritméticas/lógicas é feita com recursão, caso haja parênteses. Nestes casos, as expressões dentro dos parênteses são tratadas como uma expressões adjacentes. É resolvida apenas essa expressão, e todo seu conteúdo é substituído por uma variável, que quando voltar na pilha de execução, faça com que a expressão original substitua aquela expressão inteira entre parêntese por uma variável, e a partir disso, siga o fluxo da precedência sem os parênteses (\*, /, +, -).

A declaração dos procedimentos também são mostradas logo abaixo de todo o código de 3 endereços. Mostrando seus parâmetros de entrada, e o retorno, caso haja algum.

Os casos de continue|break: podem ser utilizados em qualquer parte da palavra (código fonte), porém só terão efetividade quando estiverem dentro de um while. Para ambos os terminais, sempre será referenciado ao while mais próximo. Essa informação é importante para o caso de haver whiles aninhados.

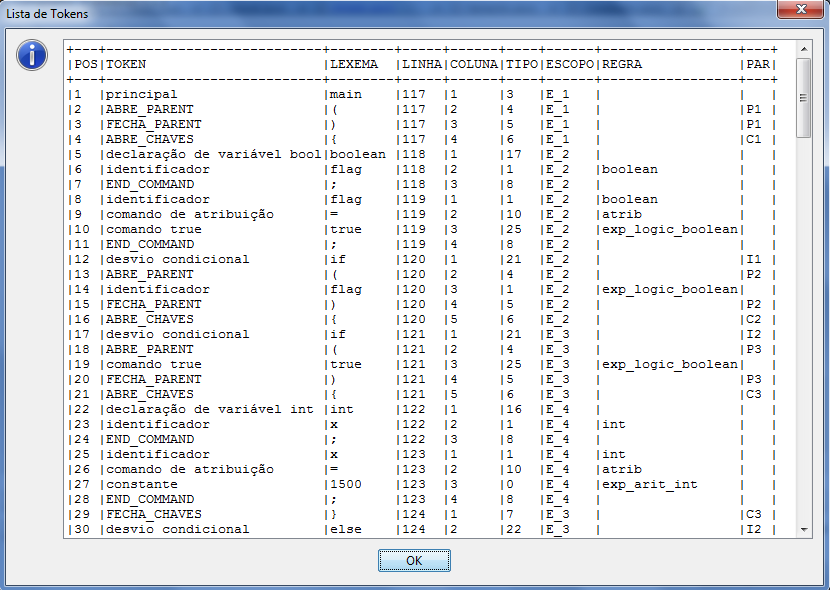
# Interface Gráfica (JFramePrincipal.java)

Há 6 botões na interface, para ajudar o programador a entender melhor seu código e como ele foi compilado.



(Figura 7: interface do programa)

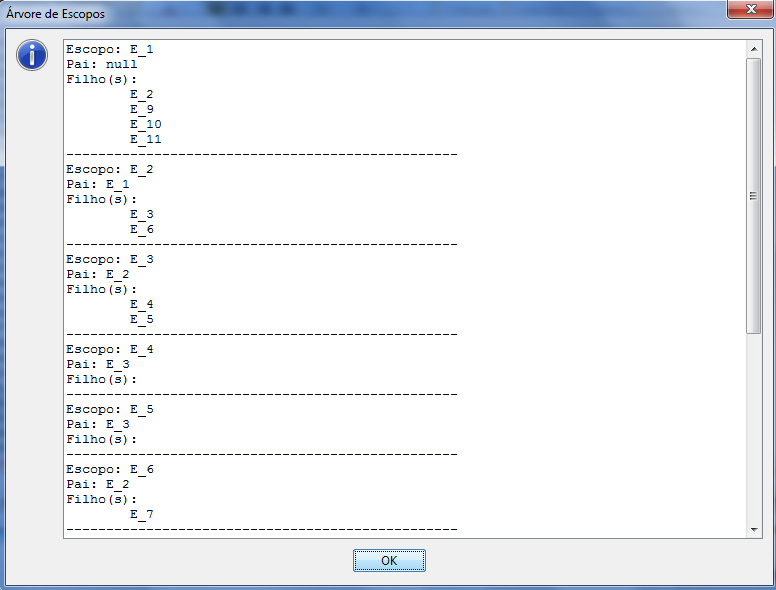
F1: lista de tokens



(Figura 8: interface da lista de tokens, criada no analisador léxico, mas alterada e utilizada até o final da compilação.)

Contém as principais informações do arraylist<Token>, facilitando a visualização dessas informações.

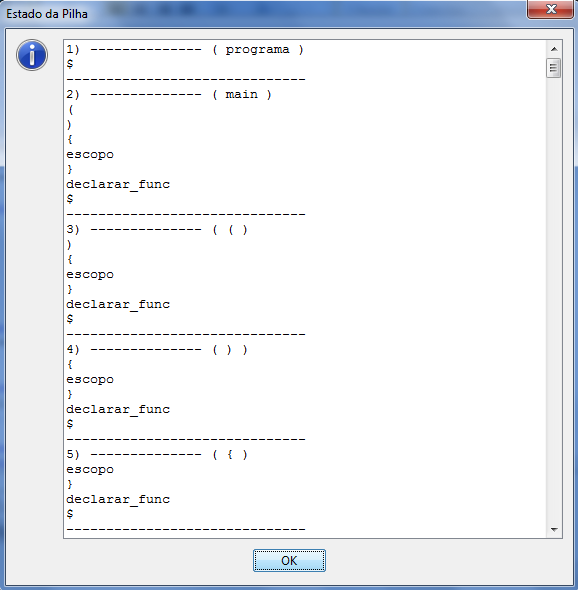
F2: Árvore de Escopos



(Figura 9: interface da árvore de escopos, criado no analisador léxico.)

É mostrado todos os escopos criados, e qual escopo é filho de qual escopo. Podendo facilitar a compreensão do programador, junto ao botão F1.

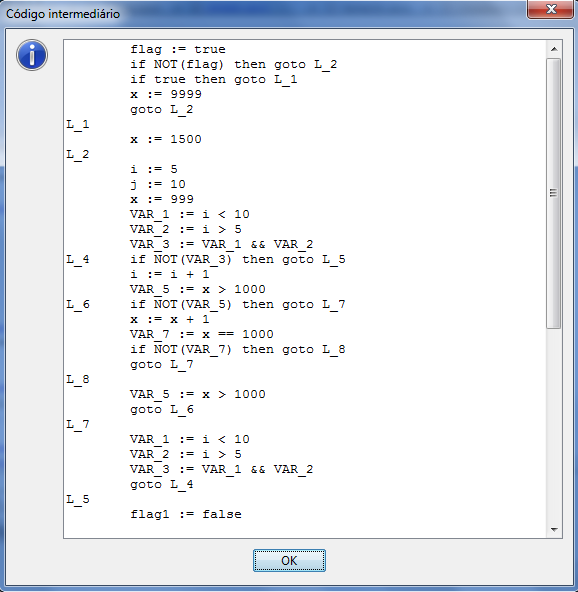
F3: Estado da pilha



(Figura 10: interface da pilha, mostrando o fluxo seguido pelo analisador sintático.)

No estado da pilha, é mostrado apenas como foi feita a derivação das regras de produção/terminais.

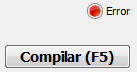
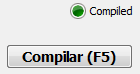
F4: Código de 3 endereços



(Figura 11: interface do código de 3 endereços.)

É mostrado o código de 3 endereços correspondente a palavra (código fonte) compilado.

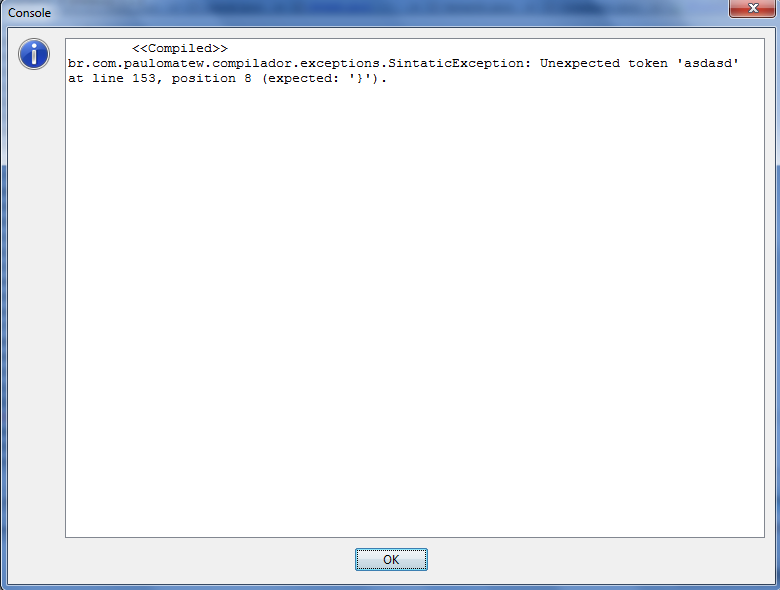
F5: Compilar



(Figura 12: imagem do botão de compilar e led informando se código foi compilado ou não.)

Ao clicar em compilar, a palavra (código fonte) vai compilar ou não. Assim sendo, um sinal será mostrado, como nas imagens acima, informando ao programador se o código foi compilado ou não.

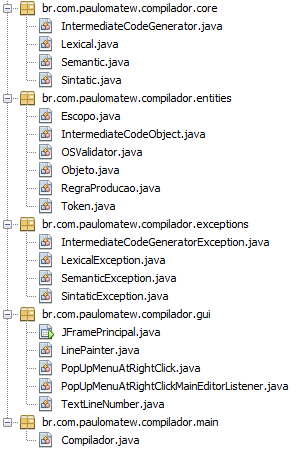
F6: Console



(Figura 13: interface do console, mostrando as saídas do compilador.)

Vai mostrar todo o histórico de erros ou de compilações. Caso o código tenha sido compilado, mostrará o texto “<<compiled>>”, caso não, mostrar qual o erro, linha e posição do mesmo.

# Estrutura de Pacotes, dependências e github



Foi utilizado uma biblioteca chamada text table formatter, que auxilia mostrar textos em tabela de forma mais organizada, utilizado na interface gráfica do botão F1 (Lista de Tokens). O projeto pode ser encontrado em: <https://github.com/TroniPM/Compilador>.

# Gramática

|  |  |  |  |
| --- | --- | --- | --- |
| <programa> | | **P1** | **main(){**<escopo>**}** <declarar\_func> |
|  | |  |  |
| <escopo> | | **P2** | <tipo\_var> <ident>**;** <escopo>| #declaração |
| **P3** | <ident> **=** <atrib>**;** <escopo>| #atribuição |
| **P4** | <printar>**;** <escopo>| #print |
| **P5** | <chamar\_func>**;** <escopo>| #call |
| **P6** | <loop\_laco> <escopo>| #if, while |
| **P6A** | <comando\_loop>**;** <escopo>| #continue;,break; |
| **P7** | **ε** |
|  | |  |  |
| <tipo\_var> | | **P8A** | **int** |
| **P8B** | **boolean** |
|  | |  |  |
| <ident> | | **P9** | <letra> <ident\_sec> |
|  | |  |  |
| <ident\_sec> | | **P10** | <letra> <ident\_sec> | |
| **P11** | <letra\_ma> <ident\_sec> | |
| **P12** | <digito> <ident\_sec> | |
| **P13** | **ε** |
|  | |  |  |
| <atrib >  (com lookahead) | | **P14** | <ident> | |
| **P15** | <numero> | |
| **P16** | <valor\_logico> | |
| **P17** | <chamar\_func | |
| **P18** | <exp\_logic> | |
| **P19** | **[** <exp\_arit> **]** | |
|  | |  |  |
| <exp\_arit> | | **P23A** | **(**<exp\_arit>**)** <oper\_arit> | |
| **P23B** | <numero> <oper\_arit> | |
| **P23C** | <ident> <oper\_arit> |
|  | |  |  |
| <oper\_arit> | | **P24** | **+** <exp\_arit> | |
| **P25** | **-** <exp\_arit> | |
| **P26** | **\*** <exp\_arit> | |
| **P27** | **/** <exp\_arit> | |
| **P28** | **ε** |
|  | |  |  |
| <declarar\_func> | | **P29** | **function** <func\_tipo> <ident> **(**<lista\_param>**)**  **{**<escopo> <retorno\_func>**}** <declarar\_func> |
| **P29A** | **ε** |
|  |  | | |
| <func\_tipo> | | **P30A** | **int |** |
| **P30B** | **boolean |** |
| **P30C** | **void** |
|  | |  |  |
| <lista\_param> | | **P31** | <tipo\_var> <ident> <lista\_param\_sec> | |
| **P32** | **ε** |
|  | |  |  |
| <lista\_param\_sec> | | **P33** | **,** <tipo\_var> <ident> <lista\_param\_sec> | |
| **P34** | **ε** |
|  | |  |  |
| <retorno\_func>  (com lookahead) | | **P35** | **return** <retorno\_func\_sec>**;** |
| **P35A** | **ε** |
|  | |  |  |
| <retorno\_func\_sec>  (com lookahead) | | **P36** | <ident> | |
| **P37** | <numero> | |
| **P38** | <valor\_logico> | |
| **P39** | <chamar\_func> | |
| **P40** | <exp\_logic> | |
| **P41** | <exp\_arit> |
|  | |  |  |
| <chamar\_func> | | **P42** | **call** <ident> **(**<lista\_arg>**)** |
|  | |  |  |
| <lista\_arg> | | **P43** | <ident> <lista\_arg\_sec>| |
| **P44** | <numero> <lista\_arg\_sec>| |
| **P45** | <boolean> <lista\_arg\_sec>| |
| **P45A** | **ε** |
|  |  | | |
| <lista\_arg\_sec> | | **P45B** | **,** <lista\_arg\_ter> <lista\_arg\_sec>| |
| **P45C** | **ε** |
|  |  | | |
| <lista\_arg\_ter> | | **P45D** | <ident> | |
| **P45E** | <numero> | |
| **P45F** | <boolean> |
|  | |  |  |
| <bloco\_if> | | **P46** | **if(**<exp\_logic>**){**<escopo>**}** <bloco\_else> |
|  | |  |  |
| <bloco\_else> | | **P47** | **else{** <escopo> **} |** |
| **P48** | **ε** |
|  | |  |  |
| <bloco\_while> | | **P49** | **while(**<exp\_logic>**){**<escopo>**}** |
|  | |  |  |
| <comando\_loop> | | **P50** | **break** | |
| **P51** | **continue** |
|  | |  |  |
| <exp\_logic>  (com lookahead) | | **P53A** | **(** <exp\_logic> **)** <oper\_logic> <exp\_logic\_cont>| |
| **P53B** | <valor\_logico> <oper\_logic> | |
| **P53C** | <ident> <oper\_logic> | |
| **P53D** | <numero> <oper\_logic> | |
|  | |  |  |
| <oper\_logic> | | **P54** | **<** <exp\_logic> <exp\_logic\_cont> | |
| **P55** | **>** <exp\_logic> <exp\_logic\_cont> | |
| **P56** | **<=** <exp\_logic> <exp\_logic\_cont> | |
| **P57** | **>=** <exp\_logic> <exp\_logic\_cont> | |
| **P58** | **!=** <exp\_logic> <exp\_logic\_cont> | |
| **P59** | **==** <exp\_logic> <oper\_log\_cont> | |
| **P59A** | **ε** |
|  | |  |  |
| <oper\_log\_cont> | | **P66** | **&&** <exp\_logic> | |
| **P67** | **||** <exp\_logic> | |
| **P68** | **ε** |
|  | |  |  |
| <printar> | | **P69** | **print(**<printar\_sec>**)** |
|  | |  |  |
| <printar\_sec> | | **P71** | <ident> | |
| **P72** | <numero> | |
| **P73** | <valor\_logico>| |
| **P70** | **ε** |
|  | |  |  |
| <loop\_laco> | | **P74** | <bloco\_if> | |
| **P75** | <bloco\_while> |
|  | |  |  |
| <numero> | | **P76** | <digito> <numero> | |
| **P77** | <digito> |
|  | |  |  |
| <digito> | | **P78** | **0|...|9** |
|  | |  |  |
| <letra> | | **P79** | **a|...|z** |
|  | |  |  |
| <letra\_ma> | | **P80** | **A|...|Z** |
|  | |  |  |
| <valor\_logico> | | **P81** | **true | false** |