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

1. 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.

```
7   public class Token extends Objeto {
8
9       public int type = -99;
10      public String lexeme = null;
11      public String description1 = null;
12      public String description2 = null;
13      public String regra = null;
14      public int line = 0;
15      public int position = 0;
16      public String scope = null;
17      public String other = null;
18
19      public boolean wasMapped = false;
```

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

2. Analisador Sintático (Sintatic.java)

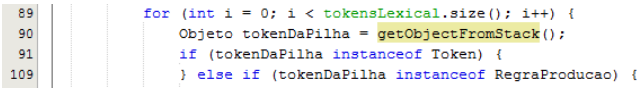
É utilizado pilha, visto que seria mais fácil a derivação de uma regra de produção desta forma. O controle da pilha é feito por dois métodos: *addToStack* e *getObjectFromStack*, responsáveis por adicionar a pilha e obter/remover da pilha, respectivamente.

```
714     private void addToStack(Objeto token) {
715         stack.add(token);
716     }
717
718     private Objeto getObjectFromStack() {
719         try {
720             Objeto t = stack.get(stack.size() - 1);
721             if (t instanceof Token) {
722                 stack.remove((Token) t);
723             } else {
724                 stack.remove((RegraProducao) t);
725             }
726
727             return t;
728         } catch (Exception ex) {
729             return null;
730         }
731
732     }
```

(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

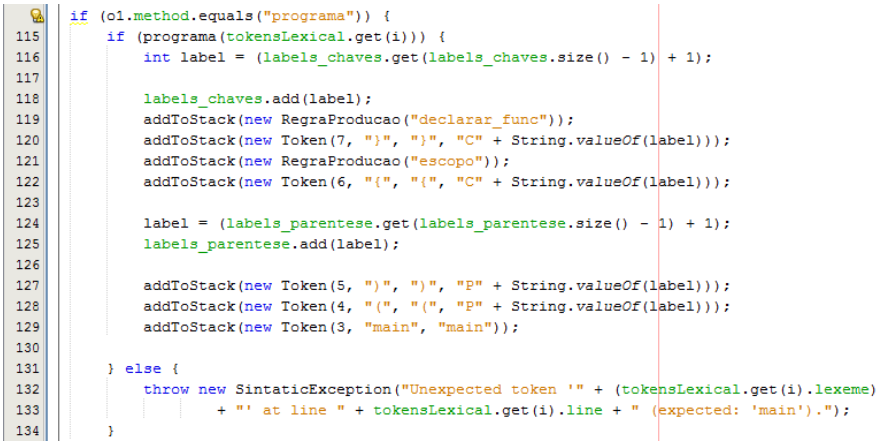
fonte). Caso seja uma regra de produção, o fluxo do código será direcionado para que seja feita a derivação dessa regra.



```
89          for (int i = 0; i < tokensLexical.size(); i++) {
90              Objeto tokenDaPilha = getObjectFromStack();
91              if (tokenDaPilha instanceof Token) {
109             } else if (tokenDaPilha instanceof RegraProducao) {
```

(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:



```
if (o1.method.equals("programa")) {
115     if (programa(tokensLexical.get(i))) {
116         int label = (labels_chaves.get(labels_chaves.size() - 1) + 1);
117
118         labels_chaves.add(label);
119         addToStack(new RegraProducao("declarar_func"));
120         addToStack(new Token(7, "}", "}", "C" + String.valueOf(label)));
121         addToStack(new RegraProducao("escopo"));
122         addToStack(new Token(6, "{", "{", "C" + String.valueOf(label)));
123
124         label = (labels_parentese.get(labels_parentese.size() - 1) + 1);
125         labels_parentese.add(label);
126
127         addToStack(new Token(5, ")", ")", "P" + String.valueOf(label)));
128         addToStack(new Token(4, "(", "(", "P" + String.valueOf(label)));
129         addToStack(new Token(3, "main", "main"));
130
131     } else {
132         throw new SintaticException("Unexpected token '" + (tokensLexical.get(i).lexeme)
133                 + "' at line " + tokensLexical.get(i).line + " (expected: 'main').");
134     }
```

(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.

3. 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:

```
Token flag = checkVariableAlreadyDefinedInScope();
flag = checkVariableWasDefinedInScopeBeforeUse();
flag = checkIfIdentifierHasSameNameMethod();
flag = checkIfMethodAlreadyDeclared();
flag = checkAtribs();
flag = checkIfFunctionHasReturn();
flag = checkReturnTypeMethodsNew();
flag = checkArgumentsNumber();
flag = checkArgumentsType();
flag = checkMethodWasDefined();
flag = checkReturnFromMethodAndVariableAssigned();
flag = checkExpressionLogic();
```

(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*:

```
19 ⊞  public void init(ArrayList<Token> tokens, ArrayList<Escopo> escopos) throws SemanticException {...90 linhas }
```

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

4. 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 `while(i < 10 && i > 5)`, será traduzido como :

```
VAR_1 := i < 10
VAR_2 := i > 5
VAR_3 := VAR_1 && VAR_2
if NOT(VAR_3) then goto
```

, 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 `while(i < 10 && i > 5)`, 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.

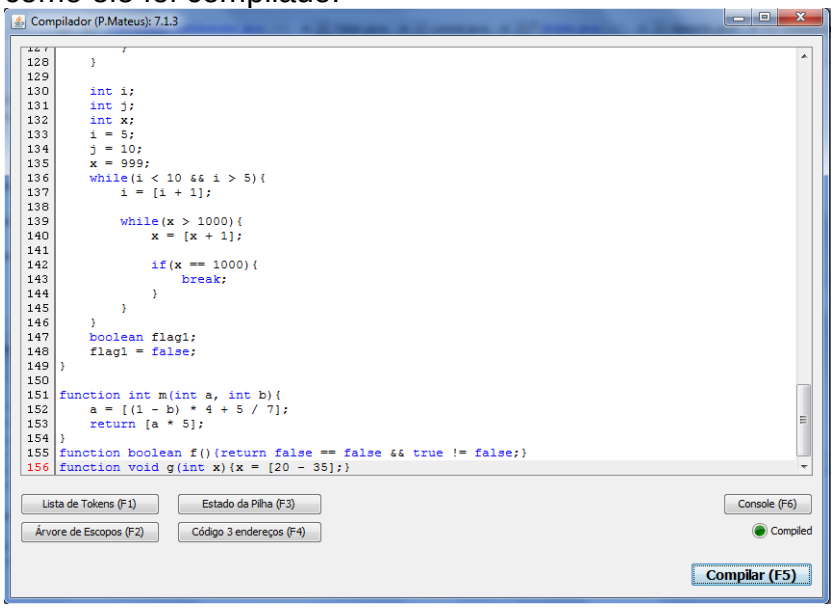
A tradução das expressões aritméticas/lógicas é feita com recursão, caso haja parênteses. Nestes casos, as expressões dentro dos parênteses são tratadas como uma expressões adjacentes. É resolvida apenas essa expressão, e todo seu conteúdo é substituído por uma variável, que quando voltar na pilha de 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.

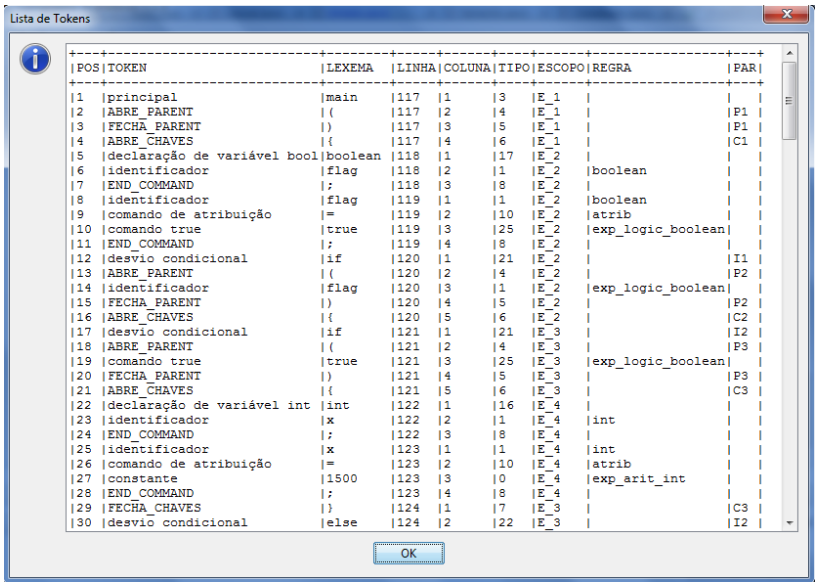
5. Interface Gráfica (JFramePrincipal.java)

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

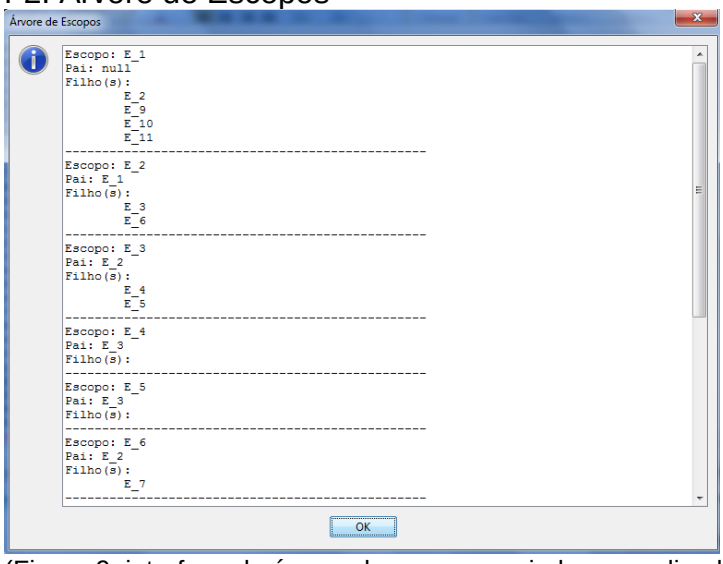


POS	TOKEN	LEXEMA	LINHA	COLUNA	TIPO	ESCOPO	REGRA	PAR
1	principal	main	117	1	3	E_1		
2	ABRE_PARENT	(	117	2	4	E_1		P1
3	FECHA_PARENT	)	117	3	5	E_1		P1
4	ABRE_CHAVES	{	117	4	6	E_1		C1
5	declaração de variável bool	boolean	118	1	17	E_2		
6	identificador	flag	118	2	1	E_2	boolean	
7	END_COMMAND	;	118	3	8	E_2		
8	identificador	flag	119	1	1	E_2	boolean	
9	comando de atribuição	=	119	2	10	E_2	atrib	
10	comando true	true	119	3	25	E_2	exp_logic_boolean	
11	END_COMMAND	;	119	4	8	E_2		
12	desvio condicional	if	120	1	21	E_2		I1
13	ABRE_PARENT	(	120	2	4	E_2		P2
14	identificador	flag	120	3	1	E_2	exp_logic_boolean	
15	FECHA_PARENT	)	120	4	5	E_2		P2
16	ABRE_CHAVES	{	120	5	6	E_2		C2
17	desvio condicional	if	121	1	21	E_3		I2
18	ABRE_PARENT	(	121	2	4	E_3		P3
19	comando true	true	121	3	25	E_3	exp_logic_boolean	
20	FECHA_PARENT	)	121	4	5	E_3		P3
21	ABRE_CHAVES	{	121	5	6	E_3		C3
22	declaração de variável int	int	122	1	16	E_4		
23	identificador	x	122	2	1	E_4	int	
24	END_COMMAND	;	122	3	8	E_4		
25	identificador	x	123	1	1	E_4	int	
26	comando de atribuição	=	123	2	10	E_4	atrib	
27	constante	1500	123	3	0	E_4	exp_arit_int	
28	END_COMMAND	;	123	4	8	E_4		
29	FECHA_CHAVES	}	124	1	7	E_3		C3
30	desvio condicional	else	124	2	22	E_3		I2

(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.

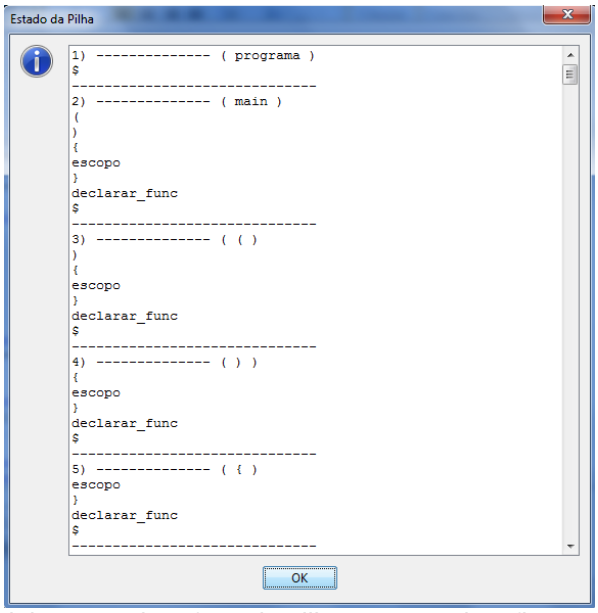
F2: Árvore de Escopos



(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.

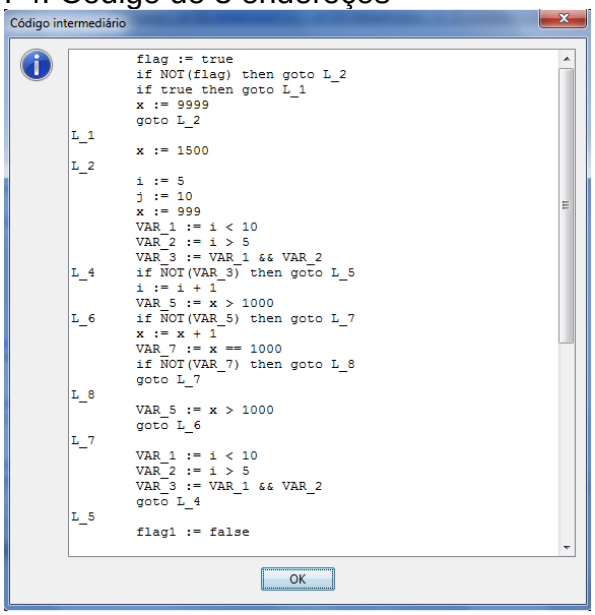
F3: Estado da pilha



(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.

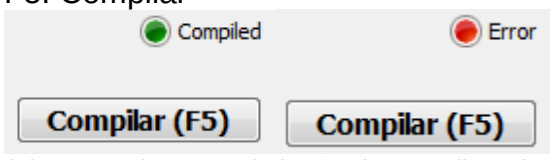
F4: Código de 3 endereços



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

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

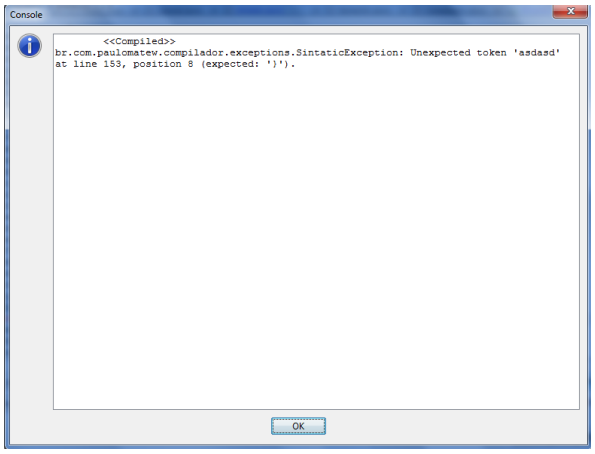
F5: Compilar



(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.

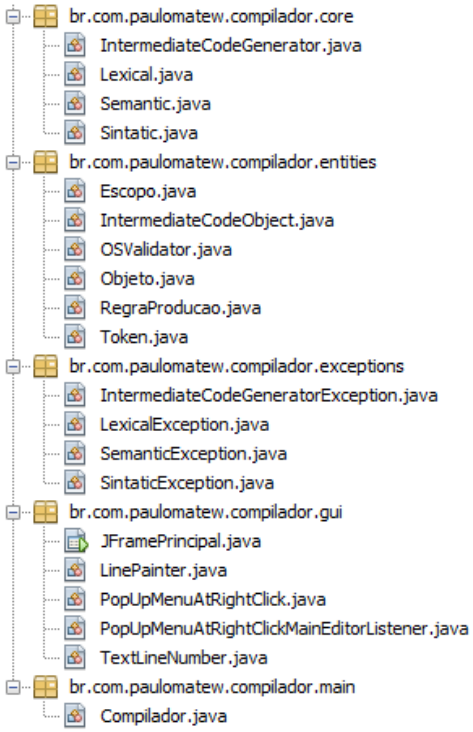
F6: Console



(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.

6. 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.

7. 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**