Analisador Sintático:

Construir uma classe que implemente um analisador sintático. Essa classe deve se chamar "`Sintatico`", e deve ser um analisador descendente recursivo.

O objetivo desse módulo é verificar se uma seqüência de *tokens* pode ser gerada pela gramática definida abaixo. O analisador sintático deve ser montado num esquema Produtor/Consumidor, com o analisador léxico (classe Lexico).

O analisador sintático deve ser o ponto de entrada do compilador, e deve chamar o procedimento l`exico.nextToken()` cada vez que um novo *token* seja necessário.

O uso da tabela de símbolos torna-se fundamental neste ponto, auxiliando no controle de escopo. Esse controle será realizado no próximo passo de implementação.

Verificações Semânticas:

O módulo Sintático deve também executar as diversas verificações semânticas requeridas pela linguagem. No caso do presente projeto, será exigida minimamente as verificações semânticas definidas abaixo:

1. Todo identificador deve ser declarado ANTES de ser utilizado em qualquer operação. Considere que o identificador usado como nome do programa é declarado na cláusula que inicia um programa.

2. Um identificador só pode ser declarado uma vez. Não haverá controle de escopo no programa (todos os identificadores compartilham o mesmo escopo global).

Gramática:

A gramática abaixo descreve as regras sintáticas da linguagem de programação que deve ser implementada. O aluno deve dedicar um tempo ao estudo dessa gramática antes de iniciar a implementação do analisador sintático. Esse estudo tem como objetivo identificar pontos de melhoria, como ambigüidades, recursões à esquerda e possibilidades de fatoração.

O aluno deve construir uma tabela sintática preditiva, que auxiliará na construção das funções de reconhecimento do analisador.

Como parte da entrega desta etapa, o aluno deve incluir um relatório contendo todas as alterações que julgou serem necessárias.

Notação: palavras em letras **minúsculas** representam *tokens* (símbolos terminais da gramática). Tratam-se dos mesmos *tokens* definidos na 1a Etapa do projeto, mas grafados em letras minúsculas. Espaços em branco são usados para fins de clareza na leitura das produções, não devendo interferir no processo de reconhecimento.

#	Produções Originais
1	S ::= **program id term** BLOCO **end_prog term**
2	BLOCO ::= **begin** CMDS **end** | CMD
3	CMDS ::= CMD CMDS | CMD | ε
4	CMD ::= DECL | COND | REP | ATRIB
5	DECL ::= **declare id type term**
6	COND ::= **if l_par** EXP-L **r_par then** BLOCO CND-2
7	CND-2 ::= **else** BLOCO | ε
8	ATRIB ::= **id attrib_op** EXP **term**
9	EXP ::= EXP-L | EXP-R | EXP-N | **literal**
10	EXP-L ::= VAL-L | EXP-R | EXP-R **logic_op** EXP-R | VAL-L **logic_op** EXP-L
11	EXP-R ::= EXP-N **rel_op** EXP-N
12	EXP-N ::= TERMO-N | EXP-N **addsub_op** TERMO-N
13	TERMO-N::= VAL-N | TERMO-N **multdiv_op** VAL-N
14	VAL-L ::= **logic_val** | **id**
15	VAL-N ::= **num_int | num_float** | **id | l_par** EXP-N **r_par**
16	REP ::= REP-F | REP-W
17	REP-F ::= **for id attrib_op** EXP-N **to** EXP-N BLOCO
18	REP-W ::= **while l_par** EXP-L **r_par** BLOCO

	Produções Processadas
1	S -> program id term BLOCO end_prog term
2	BLOCO -> begin CMDS end
3	| CMD
4	CMDS -> declare DCFLW
5	| if IFFLW
6	| REPF CMDS
7	| REPW CMDS
8	| id IDFLW
9	| ε
10	IFFLW -> l_par EXPL r_par then BLOCO CMDS
11	IDFLW -> attrib_op EXP term CMDS
12	DCFLW -> id type term CMDS
13	CMD -> DECL
14	| COND
15	| REP
16	| ATRIB
17	DECL -> declare id type term
18	COND -> if l_par EXPL r_par then BLOCO CNDB
19	CNDB -> else BLOCO
20	| ε
21	ATRIB -> id attrib_op EXP term
22	EXP -> logic_val LOGFLW
23	| id GENFLW
24	| num_int GENFLW1
25	| num_float GENFLW1
26	| l_par EXPN r_par GENFLW1
27	| literal
28	EXPL -> logic_val LOGFLW
29	| id GENFLW
30	| num_int GENFLW1
31	| num_float GENFLW1
32	| l_par EXPN r_par GENFLW1

```
33 LOGFLW  -> logic_op EXPL
34          | ε
35 GENFLW  -> logic_op EXPL
36          | GENFLW1
37 GENFLW1 -> TERMON1 EXPN1 GENFLW2
38 GENFLW2 -> rel_op EXPN GENFLW3
39          | ε
40 GENFLW3 -> logic_op EXPR
41          | ε
42 EXPR    -> EXPN rel_op EXPN
43 EXPN    -> TERMON EXPN1
44 EXPN1   -> addsub_op TERMON EXPN1
45          | ε
46 TERMON  -> VALN TERMON1
47 TERMON1 -> multdiv_op VALN TERMON1
48          | ε
49 VALN    -> num_int
50          | num_float
51          | id
52          | l_par EXPN r_par
53 REP     -> REPF
54          | REPW
55 REPF    -> for id attrib_op EXPN to EXPN BLOCO
56 REPW    -> while l_par EXPL r_par BLOCO
```