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
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 EXP_L r_par then BLOCO CMDS
11	IDFLW -> attrib_op ATRIB2 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 EXP_L r_par then BLOCO CNDB
19	CNDB -> else BLOCO
20	| ε
21	ATRIB -> id attrib_op ATRIB2 term
22	ATRIB2 -> id ATR_ID
23	| VAL_N ATR_VN
24	| l_par EXP_N r_par
25	| literal
26	ATR_ID -> EXP_L2
27	ATR_VN -> EXP_N2 AT_VN2
28	AT_VN2 -> rel_op EXP_N
29	| ε
30	EXP_L -> logic_val EXP_L2
31	| id EXP_L2
32	| VAL_N EXP_N2 rel_op EXP_N
33	| l_par EXP_N r_par
34	EXP_L2 -> rel_op EXP_N
35	| EXP_L3
36	EXP_L3 -> logic_op EXP_L
37	| ε
38	EXP_N -> id
39	| VAL_N EXP_N2
40	| l_par EXP_N r_par
41	EXP_N2 -> addsub_op EXP_N
42	| multdiv_op EXP_N
43	| ε
44	VAL_N -> num_int
45	| num_float
46	REP -> REPF
47	| REPW
48	REPF -> for id attrib_op EXP_N to EXP_N BLOCO
49	REPW -> while l_par EXP_L r_par BLOCO