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**	-> DECL CMDS
5		| COND CMDS
6		| REPF CMDS
7		| REPW CMDS
8		| ATRIB CMDS
9		| ε
10	**CMD**	-> DECL
11		| COND
12		| REP
13		| ATRIB
14	**DECL**	-> declare id type term
15	**COND**	-> if l_par EXPLO r_par then BLOCO CNDB
16	**CNDB**	-> else BLOCO
17		| ε
18	**ATRIB**	-> id assign EXP term
19	**EXP**	-> logic_val FVALLOG
20		|id FID
21		|num_int FNUM
22		|num_float FNUM
23		|l_par FLPAR
24		|literal
25	**FID**	-> FVALLOG
26		|OPNUM FOPNUM
27	**FOPNUM**	-> EXPNUM FEXPNUM_1
28	**FEXPNUM_1**	-> relop EXPNUM
29		| ε
30	**FNUM**	-> OPNUM FOPNUM
31		| ε
32	**FLPAR**	-> EXPNUM FEXPNUM
33	**FEXPNUM**	-> r_par FRPAR
34	**FRPAR**	-> relop EXPNUM
35		| ε
36	**EXPLO**	-> logic_val FVALLOG
37		|id FID_1
38		|num_int OPNUM EXPNUM relop EXPNUM
39		|num_float OPNUM EXPNUM relop EXPNUM
40		|l_par EXPNUM r_par relop EXPNUM
41	**FID_1**	-> FVALLOG
42		|relop EXPNUM
43		|OPNUM EXPNUM relop EXPNUM
44	**FVALLOG**	-> logic_op EXPLO
45		| ε
46	**EXPNUM**	-> VAL XEXPNUM
47		|l_par EXPNUM r_par
48	**XEXPNUM**	-> OPNUM EXPNUM
49		| ε
50	**OPNUM**	-> arit_as
51		| arit_md
52	**VAL**	-> id
53		|num_int
54		|num_float
55	**REP**	-> REPF
56		| REPW
57	**REPF**	-> for id attrib EXPNUM to EXPNUM BLOCO
58	**REPW**	-> while l_par EXPLO r_par BLOCO