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 FNUMINT
22		|num_float FNUMFLOAT
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	FNUMINT	-> OPNUM FOPNUM 1
31	FOPNUM 1	-> EXPNUM FEXPNUM 2
32	FEXPNUM 2	-> relop EXPNUM
33		| ε
34	FNUMFLOAT	-> OPNUM FOPNUM 2
35	FOPNUM 2	-> EXPNUM FEXPNUM 3
36	FEXPNUM 3	-> relop EXPNUM
37		| ε
38	FLPAR	-> EXPNUM FEXPNUM
39	FEXPNUM	-> r_par FRPAR
40	FRPAR	-> relop EXPNUM
41		| ε
42	EXPLO	-> logic val FVALLOG
43		|id FID 1
44		|num int OPNUM EXPNUM relop EXPNUM
45		|num_float OPNUM EXPNUM relop EXPNUM
46		|l par EXPNUM r par relop EXPNUM
47	FID_1	-> FVALLOG
48		|OPNUM EXPNUM relop EXPNUM
49	FVALLOG	-> logic_op EXPLO
50		| ε
51	EXPNUM	-> VAL XEXPNUM
52		|l par EXPNUM r par
53	XEXPNUM	-> OPNUM EXPNUM
54		| ε
55	OPNUM	-> arit_as
56		| arit md
57	VAL	-> id
58		|num int
59		|num_float
60	REP	-> REPF
61		| REPW
62	REPF	-> for id attrib EXPNUM to EXPNUM BLOCO
63	REPW	-> while l_par EXPLO r_par BLOCO
64	FNUMINT	-> ε
65	FNUMFLOAT	-> ε
66	FID 1	-> relop EXPNUM