API OF THE GRAMMATICA LIBRARY (FOR CREATING CHECKING SOFTWARE) VERSION 6.8

History

<u>6.2b:</u> introduction of `intransitiveErrorMask`
<u>6.4a:</u> introduction of `SFGSetStraightQuotes` and `SFGSetNbWordsLongSentence`
<u>6.5:</u> introduction of types SFint8, SFuint8, etc. instead of int8, uint8, etc.
<u>6.8:</u> introduction of SFGSetRegistrationKeys2, SFGSetRegistrationKeys3 and SFGIsFunctionRegistered. Introduction of list of constants A.10

A. List of constants

1. Constants defining the language of correction or of messages sent by Grammatica

```
#define English 0
#define French 3
#define Spanish 10
#define German 11
```

2. Common constants

```
#define kLenMaxKey 35
#define LENWORD 36
#define LENSUGG 3*LENWORD
#define LENABBREV 5
#define MAXNBERRORPARA 60
#define DICTMAXLENLINE 1000
#define MAXNUMBERSPECIALCHARS 256
#define MAXNUMSPACES 32
#define MAXNUMPUNCTSIGNS 32
#define doubleSpaceAllowed 0x80000000
```


`kLenMaxKey` indicates the maximum size of an activation key for the Grammatica library
`LENWORD` indicates the maximum size of a word
`LENSUGG` indicates the maximum size of a suggestion provided by Grammatica
`LENABBREV` indicates the maximum size of an abbreviation
`MAXNBERRORPARA` indicates the maximum number of errors that Grammatica can detect in a paragraph
`DICTMAXLENLINE` indicates the maximum length of a line from the dictionary of definitions and synonyms (one line contains one definition)
`MAXNUMBERSPECIALCHARS` indicates the size of the table to provide for the function `SFGSetSpecialChars`.
`MAXNUMSPACES` indicates the size of the table to provide for the functions `SFGGetSpaces` and `SFGSetSpaces`.
`MAXNUMPUNCTSIGNS` indicates the size of the table to provide for the functions `SFGGetPunctSigns` and `SFGSetPunctSigns`.
`DoubleSpaceAllowed` indicates that double space are allowed. This constant is used in the `spacesAllowed` field of `punctLeftSignReq` and `punctRightSignReq`.

3. Constants used in the function `SFGOptionNonUSUK`

These constants are used to define the English spellings accepted by Grammatica (British, American, or both)

```
#define USUKoption 0
#define NonUKoption 1
#define NonUSoption 2
```

`USUKoption` -> accepts American and British spellings
`NonUKoption` -> accepts American spellings and rejects British spellings
`NonUSoption` -> accepts British spellings and rejects American spellings

4. Constants used in the functions `SFGTestGrammarParagraph`, `SFGTestSpellingParagraph` and `SFGPunctParagraph`

These constants allow one to determine whether a mode of automatic correction by substitution of detected errors is possible or not. They constitute the possible values for the field `codeForAutoSubstitution` in the type `anErrorParagraphTab` used for the return of errors (see below).

```
#define ErrorAbbreviation 1
#define ErrorWithFixedSubstitution 2
#define ErrorNoAutomaticSubstitution 3
#define ErrorAutoSubstitutionAgreementNounCluster 4
#define ErrorAutoSubstitutionAgreementRepresentativeNounClause 5
#define ErrorAutoSubstitutionAgreementPastParticiple 6
#define ErrorAutoSubstitutionAgreementVerbWithSubject 7
#define ErrorAutoSubstitutionAgreementPredicateAdjective 8
#define ErrorAutoSubstitutionPunctuation 9
```

`ErrorAbbreviation` -> abbreviation of an acronym or of a proper noun
`ErrorWithFixedSubstitution` -> incorrect spelling of a common noun or of an acronym or abbreviation. The suggestion chosen by the user may be retained for future replacements.
`ErrorNoAutomaticSubstitution` -> error for which correction by substitution is not possible

The following five errors can give rise to an automatic substitution, using the first suggestion offered by Grammatica, if there is one.
   `ErrorAutoSubstitutionAgreementNounCluster` -> agreement of the noun cluster
   `ErrorAutoSubstitutionAgreementRepresentativeNounClause` -> agreement of gender between noun clusters and elements referring to the noun cluster (pronouns)
   `ErrorAutoSubstitutionAgreementPastParticiple` -> agreement of past participles
   `ErrorAutoSubstitutionAgreementVerbWithSubject` -> agreement (in number and person) between a verb and its subject
   `ErrorAutoSubstitutionAgreementPredicateAdjective` -> agreement of predicate adjectives

`ErrorAutoSubstitutionPunctuation` -> Punctuation error

5. Constants used (for French and English) in the function `SFGTestGrammarParagraph` to ask this function which optional errors it should return (parameter `mistakesMask`). These constants may be accumulated. For example, so that only errors relating to doubled words and ligatures are flagged, one uses
`duplicatedWordErrorMask+ligaturesErrorMask`
For all grammatical errors to be flagged, one uses `allErrorMask`

```
#define duplicatedWordErrorMask 0x00000001
#define ligaturesErrorMask 0x00000002
#define unCapitalizedProperNounErrorMask 0x00000004
#define widowedQuotationMarkErrorMask 0x00000008
#define widowedApostropheErrorMask 0x00000010
#define widowedParenthesesErrorMask 0x00000020
#define widowedBracketErrorMask 0x00000040
#define widowedCurlyBraceErrorMask 0x00000080
#define missingVerbErrorMask 0x00000100
#define missingQuestionMarkErrorMask 0x00000200
#define missingInterrogativeFormErrorMask 0x00000400
#define questionableConstructionErrorMask 0x00000800
#define incompleteNegationFormErrorMask 0x00001000
```

```
#define repeatedWordErrorMask 0x00002000
#define tooManyRelativePronounsErrorMask 0x00004000
#define complexPredicatePhrasesErrorMask 0x00008000
#define longSentenceErrorMask 0x00010000
#define nonStandardVocabularyErrorMask 0x00020000
#define intransitiveErrorMask 0x00040000

#define allErrorMask 0xFFFFFFFF
```

The explanation of these constants is provided below.

```
duplicatedWordErrorMask -> duplicate word
ligaturesErrorMask -> Ligatures
unCapitalizedProperNounErrorMask -> Proper nouns missing capitals
widowedQuotationMarkErrorMask -> Balance of quotation marks
widowedApostropheErrorMask -> Balance of apostrophes (inverted commas)
widowedParenthesesErrorMask -> Balance of parentheses
widowedBracketErrorMask -> Balance of square brackets
widowedCurlyBraceErrorMask -> Balance of curly brackets
missingVerbErrorMask -> Missing verb
missingQuestionMarkErrorMask -> Missing question mark
missingInterrogativeFormErrorMask -> Missing interrogative form
questionableConstructionErrorMask -> Problematic construction
incompleteNegationFormErrorMask -> Missing « ne » (in a French negative construction)
repeatedWordErrorMask -> Repeated words
tooManyRelativePronounsErrorMask -> Three or more relative pronouns
complexPredicatePhrasesErrorMask -> Overly heavy subordination
longSentenceErrorMask -> Excessively long sentence
nonStandardVocabularyErrorMask -> non standard vocabulary (slang, colloquial, unsophisticated, literary or
archaic)
intransitiveErrorMask -> intransivity mistakes

allErrorMask -> Returns all errors in the category of optional errors
```

6. Constants used in the function `SFGGetWordAnalysis (French and English)`

```
#define AnalysisNoWordFound 0
#define AnalysisOK 1
#define AnalysisUnknownWord 2
#define AnalysisWithMistakes 3
```

```
AnalysisNoWordFound  -> No word has been selected
AnalysisOK  -> Correct analysis
AnalysisUnknownWord  -> The selected word is unknown to Grammatica
AnalysisWithMistakes  -> The selected word is included in a sentence containing one or more errrors
```

```
#define AnalysisSynonymDefinition 0x00000001
#define AnalysisSynonymDomain 0x00000002
```

These constants are used to decode the return parameter `synonymCase` of the function `SFGGetWordAnalysis`

7. Constants used in the functions `SFGInfinitive, SFGConjugate, and SFGPlural`

a. Constants returned by the functions

```
#define ErrorUnknownWord 9
```

The following constants are used exclusively `SFGInfinitive` and `SFGConjugate` with the exception of `NotInfinitive` which is used only in `SFGConjugate (French and English)`

```
#define ErrorNotVerbForm 11
#define ErrorNotInfinitive 20

#define FirstConjugation 21
#define SecondConjugation 22
#define ThirdConjugation 23
#define Auxiliary 24
#define Defective 25
#define VeryDefective 26
#define Impersonal 27
#define CommonThirdPerson 28
#define CommonPlural 29
#define CommonInfinitivePastParticiple 30
#define CommonInfinitive 31
#define RegularEnglish 32
#define IrregularEnglish 33
```

The following constants are used only in `SFGPlural` and `SFGSingular`

```
#define VariableNounAdjective0 0
#define VariableNounAdjective1 1
```

`VariableNounAdjective0` and `VariableNounAdjective1` mean that the word is a noun or adjective that is variable. The plural or the singular is returned by the corresponding function.

```
#define ErrorInvariable 34
#define ErrorNoPluralForm 35
#define ErrorVerbFormNoPluralForm 36
#define ErrorProperNounNoPluralForm 37
```

The folledingue constant is used only in `SFGSingular`

```
#define ErrorCantGetSingularForm 38
```


`ErrorUnknownWord` -> Word not recognized by Grammatica
`ErrorNotVerbForm` -> The word is not a verb form
`ErrorNotInfinitive` -> This word is not recognized as a verb in the infinitive

FRENCH AND ENGLISH ONLY :
`FirstConjugation` -> Regular verb of the first conjugation (French)
`SecondConjugation` -> Regular verb of the second conjugation (French)
`ThirdConjugation` -> Verb of the third conjugation (French)
`Auxiliary` -> Auxiliary verb
`Defective` -> Defective verb
`VeryDefective` -> Very defective verb
`Impersonal` -> Impersonal verb (uses « il » or « it » only)
`CommonThirdPerson` -> Verb commonly used only in the 3rd person
`CommonPlural` -> Verb commonly used only in the plural
`CommonInfinitivePastParticiple` -> Verb commonly used only in the infinitive and past participle
`CommonInfinitive` -> Verb commonly used only in the infinitive
`RegularEnglish` -> Regular verb in English
`IrregularEnglish` -> Irregular verb in English
`ErrorInvariable` -> Invariable or already in the plural form
`ErrorNoPluralForm` -> No plural form
`ErrorVerbFormNoPluralForm` -> Verb form : no plural

```
ErrorProperNounNoPluralForm  -> Proper noun : no plural

ErrorCantGetSingularForm  -> No singular form
```

b. Constants used by `SFGConjugate`.

```
#define PresentParticiple 0
#define PastParticiple 1
#define Present 2
#define Imperfect 3
#define Future 4
#define SimplePast 5
#define PresentSubjunctive 6
#define ImperfectSubjunctive 7
#define Conditional 8
#define Imperative 9
#define EnglishVerb 10
```

```
PresentParticiple ->  present participle
PastParticiple ->  past participle
Present ->  present indicative
Imperfect -> imperfect indicative
Future ->  future indicative
SimplePast -> simple past indicative
PresentSubjunctive ->  present subjunctive
ImperfectSubjunctive ->  imperfect subjunctive
Conditional ->  condiational
Imperative ->  imperative
EnglishVerb -> English mode
```

Spanish tenses :

```
#define Presente   0
#define ImperfectoDeIndicativo 1
#define Futuro 2
#define PreteritoPerfecto 3
#define PresenteDeSubjuntivo 4
#define ImperfectoDeSubjuntivo 5
#define ImperfectoDeSubjuntivoII 6
#define FuturoDeSubjuntivo 7
#define PresenteDeCondicional 8
#define Imperativo 9
#define Participio 10
#define Gerundio 11
```

German tenses :

```
#define IndicativPrasens   0
#define Prateritum 1
#define KonjunktivI 2
#define KonjunktivII 3
#define Imperativ 4
#define PartizipPerfekt 5
#define PartizipPrasens 6
```

c. Contents of the returned parameter `solutions`

In general, the solutions parameters contains six solutions for each person : 1st person singular, 2nd person singular, 3rd person singular, 1st person plural, 2nd person plural, 3rd person plural.

When the verb is defective, there are solutions that can be empty (empty zero-terminated string).

For the imperative in French, there are three solutions:
2nd person singular
1st person plural
2nd person plural

For the imperative in Spanish, there are five solutions:
2nd person singular
3rd person singular
1st person plural
2nd person plural
3rd person plural

For the past participle in French and in Spanish, there are four solutions:
past participle masculine singular
past participle masculine plural
past participle feminine singular
past participle feminine plural
If the verb is intransitive, there is one solution for the past participle.

For the present participle and past participle in German, there is one solution.

For the English mode, there are five solutions:

1st person present
3rd person present
Present participle
Preterit
Past participle

8. Constants used in the functions of the lexicon
8.a.Constants used in the functions `SFGAnalyseWordLex`, `SFGInsertLex`, `SFGChangeLex`, `SFGGetListWordsLex`, `SFGGetValuesWordLex` et `SFGRemoveLex`, as well as in the structure `lexWord`:


```
#define ProperNoun 1
#define CommonNoun 2
#define Adjective 3
#define Verb 5
#define Adverb 6

#define Masculine 1
#define Feminine 2
#define MasculineFeminine 3
#define Neuter 4


#define SingularPluralInS 1
#define Irregular 2
#define Plural 3
#define SingularPlural 4
#define SingularPluralInES 5
#define SingularPluralInX 6
#define SingularPluralInIES 7
#define SingularPluralInMEN 8
#define SingularPluralInAEandS 9
#define SingularPluralInOS 10
```

```
#define GermanInvGenetivePluralInN 11
#define GermanInvGenetivePluralInEN 12
#define GermanInvGenetivePluralInS 13
#define GermanInvGenetivePluralInES 14
#define GermanGenetiveInSPluralInN 15
#define GermanGenetiveInESPluralInEN 16
#define GermanGenetiveInSPluralInS 17
#define GermanGenetiveInESPluralInES 18
```

Constants definining the part of speech

`ProperNoun` -> Proper noun
`CommonNoun` -> Common noun
`Adjective` -> Adjective
`Verb` -> Verb
`Adverb` -> Adverb

Constants defining the gender of a word (except verbs or adverbs)

`Masculine` -> Masculine
`Feminine` -> Feminine
`MasculineFeminine` -> Masculine and Feminine
`Neuter` -> Neuter (only for German)

Constants defining the number of a word (except verbs or adverbs)

`SingularPluralInS` -> Singular with a plural ending in « s » (English, Spanish and French)
`Irregular` -> Singular without a plural form, or with in irregular plural (English, Spanish, German and French)
`Plural` -> Plural (English, Spanish, German and French)
`SingularPlural` -> Singular and plural (English, Spanish, German and French)
`SingularPluralInES` -> Singular with a plural ending in « es » (English and Spanish)
`SingularPluralInX` -> Singular with a plural ending in « x » (French only)
`SingularPluralInIES` -> Singular with a plural ending in « ies » (English only)
`SingularPluralInMEN` -> Singular with a plural ending in « men » (English only)
`SingularPluralInAEandS` -> Singular with a plural ending in « ae » and « s » (English only)
`SingularPluralInOS` -> Singular with a plural ending in « os » (Spanish only)

German only :

GermanInvGenetivePluralInN -> Singular with a plural ending in « n » and invariable genetive

GermanInvGenetivePluralInEN -> Singular with a plural ending in « en » and invariable genetive

GermanInvGenetivePluralInS -> Singular with a plural ending in « s » and invariable genetive

GermanInvGenetivePluralInES -> Singular with a plural ending in « es » and invariable genetive

GermanGenetiveInSPluralInN -> Singular with a plural ending in « n » and with a genetive ending in « s »

GermanGenetiveInESPluralInEN -> Singular with a plural ending in « en » and with a genetive ending in « s »

GermanGenetiveInSPluralInS -> Singular with a plural ending in « s » and with a genetive ending in « s »

GermanGenetiveInESPluralInES -> Singular with a plural ending in « es » and with a genetive ending in « s »

8.b. Constants used in `SFGGetListWordsLex`

```
#define EnglishSelection 1
#define FrenchSelection 8
#define SpanishSelection 1024
```

```
#define GermanSelection 2048
#define allLanguagesSel 0xFFFFFFFF
```

EnglishSelection -> `SFGGetListWordsLex` returns only English words (including proper nouns)
FrenchSelection -> `SFGGetListWordsLex` returns only French words (including proper nouns)
SpanishSelection -> `SFGGetListWordsLex` returns only Spanish words (including proper nouns)
GemanSelection -> `SFGGetListWordsLex` returns only German words (including proper nouns)
allLanguagesSel -> `SFGGetListWordsLex` returns all words

8.c. Constants representing error codes returned by the functions of the lexicon

```
#define LexNoErr 0
#define LexSpaceNotLogo 1
#define LexNoSpaceInAbbr 2
#define LexUnknownConjModel 3
#define LexConjModelNotVerb 4
#define LexConjModelNotVerbSameConj 5
#define LexConjModelNotInfinitive 6
#define LexNoMoreFourSpacesInLogo 7
#define LexIncorrectLenAbbr 8
#define LexAbbrAlreadyDefined  9
#define LexWordTooLong 10
#define LexIncorrectPlural  11
#define LexConflictAbbr 12
#define LexPastParticipleInvariable 13
#define LexConjModelNotValid 14
#define LexNoPluralInS 15
#define LexNoMoreTwoLigatures 16
#define LexNoLigatureInEnglish 17
#define LexEnglishAdjectiveInvariable 18
#define LexAtLeastTwoLetters 19
#define LexNoInvalidChar 20
#define LexVerbNotSuppressible 21
#define LexWordNotFound -12
#define LexNotOpen -11
#define LexIncorrectVersion -10
#define LexErrorInLexicon -9
#define LexMemoryError -8
#define LexProperNounUppercase -7
#define LexTooCloseToFourLogos -6
#define LexIOError -5
#define LexTooCommonWord -4
#define LexConflictWithAbbr -3
#define LexAlreadyDefined -2
#define LexAlreadyInDict -1
```

`LexNoErr` -> No error
`LexSpaceNotLogo` -> A word containing one or more spaces must be an acronym
`LexNoSpaceInAbbr` -> An abbreviation may not contain a space
`LexUnknownConjModel` -> This conjugation model is not present in the corpus or the lexicon
`LexConjModelNotVerb` -> This conjugation model is not a verb
`LexConjModelNotVerbSameConj` -> This conjugation model is not a verb with the same conjugation
`LexConjModelNotInfinitive` -> This conjugation model is not a verb in the infinitive
`LexNoMoreFourSpacesInLogo` -> An acronym may not contain more than four spaces
`LexIncorrectLenAbbr` -> An abbreviation must contain between two and four characters
`LexAbbrAlreadyDefined` -> This abbreviation is already defined in the lexicon. Choose another !
`LexWordTooLong` -> A word may contain a maximum of 34 characters if it is an acronym ; 24 for any other type
`LexIncorrectPlural` -> This word cannot have that plural form

`LexConflictAbbr` -> This abbreviation is a word in the corpus or lexicon. Please choose another !

`LexPastParticipleInvariable` -> This conjugation model is not suitable, for its past participle is invariable

`LexConjModelNotValid` -> This conjugation model is not suitable (defective verb)

`LexNoPluralInS` -> This word cannot have a plural ending in « s »

`LexNoMoreTwoLigatures` -> This word may not contain two ligatures

`LexNoLigatureInEnglish` -> Ligatures may not be used in English

`LexEnglishAdjectiveInvariable` -> English adjectives are always invariable

`LexAtLeastTwoLetters` -> A word must contain at least two letters

`LexNoInvalidChar` -> A word may not contain invalid characters

`LexVerbNotSuppressible` -> The verb cannot be removed until identical verbs with a particle are removed

`LexWordNotFound` -> This word cannot be deleted, for it does not exist in the lexicon !

`LexNotOpen` -> No lexicon is open !

`LexIncorrectVersion` -> This version of the lexicon is not compatible with Grammatica

`LexErrorInLexicon` -> The modification could not be recorded because of an internal problem in the handling of the lexicon

`LexMemoryError` -> Insufficient memory

`LexProperNounUppercase` -> A proper noun cannot begin with a lower case letter

`LexTooCloseToFourLogos` -> This word is too similar to four others already defined as an acroynm. Please choose another !

`LexIOError` -> Input/output error in the lexicon

`LexTooCommonWord` -> This word is part of a list of frequenetly used words and may not be entered as a proper noun or an acronym. Please choose another !

`LexConflictWithAbbr` -> This word has already been defined as an abbreviation in the lexicon. Please choose another !

`LexAlreadyDefined` -> This word has already been added into the lexicon with the same part of speech. Do you wish to replace it ?

`LexAlreadyInDict` -> This word has not been defined as a proper noun, and it is present in the corpus. Please choose another !

9. Constants used in the dictionary of definitions and synonyms

```
#define DictWordPresent 1
#define DictWordAbsent 0
```

`DictWordPresent` -> No error : the word is included in the dictionary
`DictWordAbsent` -> No error : the word is not present in the dictionary

10. Constants used in SFGIsFunctionRegistered

```
#define kFrenchSpelling "FRSP"
#define kFrenchGrammar "FRGR"
#define kEnglishSpelling "ENSP"
#define kEnglishGrammar "ENGR"
#define kDictionnary "DICT"
#define kStemming "STEM"
#define kSentenceAnalysis "ASEN"
#define kEnglishMedical "EMED"
#define kSpanishSpelling "ESSP"
#define kSpanishGrammar "ESGR"
#define kGermanSpelling "DESP"
#define kGermanGrammar "DEGR"
```

B. Types of data

1. Basic types

The following types of data are used as basic types:

```
typedef char SFint8;
typedef short SFint16;
typedef long SFint32;

typedef unsigned char SFuint8;
typedef unsigned short SFuint16;
typedef unsigned long SFuint32;
```

The type `SFGFile` represents a reference toward another open file of data. It is defined differently according to the platform :

```
typedef FILE* SFGFile;    // under Windows and Unix
typedef short SFGFile;    // under Macintosh (corresponds to a file refnum obtained by FSpOpenDF)
```

2. the type `errorParagraphTab`

The structure `anErrorParagraphTab` is used by the functions `SFGTestGrammarParagraph`, `SFGTestSpellingParagraph` and `SFGPunctParagraph`. The type `errorParagraphTab` is the type used to return the entire group of errors found in a paragraph.

```
typedef struct
    {
     SFint32 errorIndexSelStart;
     SFint32 errorIndexSelEnd;
     SFuint16 codeForAutoSubstitution;
     SFuint8* message;
     SFint16 suggestionsCount;
     SFuint8 suggestions[12][LENSUGG];
    } anErrorParagraphTab;

typedef anErrorParagraphTab errorParagraphTab[MAXNBERRORPARA];
```

The description of the different attributes of `anErrorParagraphTab` is as follows :
   `errorIndexSelStart` is the index of the first character to select to flag the error
   `errorIndexSelEnd` is the index of the last character to select to flag the error
   `codeForAutoSubstitution` indicates if the error may or may not be automatically corrected by substitution
   `message` which is the message to display (which is a zero-terminated string)
   `suggestionsCount` which is the number of suggestions for the errror
   `suggestions` which is the table of the suggestions, each one being a zero-terminated string.

3. Types used for punctuation

A table defining the parameters for checking punctuation has the following structure :

```
typedef punctSignValue anPunctSignTab[MAXNUMPUNCTSIGNS];
```

Keeping in mind that :

```
typedef struct {
     punctLeftSignReq leftReq;
     punctRightSignReq rightReq;
     SFuint8 punctChar;
} punctSignValue;
```

`leftReq` designates the constraints to the left of the sign
`rightReq` designates the constraints to the right of the sign
`punctChar` designates the ASCII code of the punctuation sign

```
typedef struct {
      SFuint32 punctSignsAllowed;
      SFuint32 spacesAllowed;
      SFuint8  otherSignsAllowed;
      SFuint8  suggestExists;
      SFuint8  SpaceDeletion;
      SFuint8  SpaceInsertRemplaceIndex;
} punctLeftSignReq;
```

`punctSignsAllowed` -> each bit corresponds to a different element of a table `anPunctSignTab`. They are the punctuation mark allowed to the left of the punctuation mark in question

`spacesAllowed` -> each bit corresponds to a different element of the table established by `SFGSetSpaces`. They are the spaces allowed to the left of the punctuation mark in question. If the constant `doubleSpaceAllowed` is added, then double space are allowed.

`otherSignsAllowed` -> 1 indicates that the other characters (special characters set by `SFGSetSpecialChars` are included) are authorized to the left of the punctuation mark in question ; 0 indicates that they are not authorized.

`suggestExists` -> 1 indicates that there exists a suggestion in the case of an error to the left of the mark ; 0 indicates that there is no suggestion.

`SpaceDeletion` -> (used if `suggestExists` equals 1), 1 indicates that the suggestion consists of deleting all spaces to the left of the punctuation mark ; 0 indicates that one must use the space contained in `SpaceInsertRemplaceIndex`

`SpaceInsertRemplaceIndex` -> (used if `suggestExists` equals 1 and if `SpaceDeletion` equals 0), contains the value of the space which is to be inserted or which is to replace the space that exists to the left of the punctuation mark.

When two punctuation marks are in sequence, Grammatica uses the left limits (`punctLeftSignReq`) of the second mark. Thus, it is not necessary to specify which punctuation marks may follow others. The structure for the right limits thus contains no `punctSignsAllowed` attribute.

```
typedef struct {
      SFuint32 spacesAllowed;
      SFuint8  otherSignsAllowed;
      SFuint8  suggestExists;
      SFuint8  SpaceDeletion;
      SFuint8  SpaceInsertRemplaceIndex;
} punctRightSignReq;
```

`spacesAllowed` -> each bit corresponds to a different element of the table established by `SFGSetSpaces`. They are the spaces allowed to the right of the punctuation mark in question. If the constant `doubleSpaceAllowed` is added, then double space are allowed.

`otherSignsAllowed` -> 1 indicates that the other characters (special characters set by `SFGSetSpecialChars` are included) are authorized to the right of the punctuation mark in question ; 0 indicates that they are not authorized.

`suggestExists` -> 1 indicates that there exists a suggestion in the case of an error to the right of the mark ; 0 indicates that there is no suggestion.

`SpaceDeletion` -> (used if `suggestExists` equals 1), 1 indicates that the suggestion consists of deleting all spaces to the right of the punctuation mark ; 0 indicates that one must use the space contained in `SpaceInsertRemplaceIndex`

`SpaceInsertRemplaceIndex` -> (used if `suggestExists` equals 1 and if `SpaceDeletion` equals 0), contains the value of the space which is to be inserted or which is to replace the space that exists to the right of the punctuation mark.

4. The structure `lexWord` returned by `SFGGetListWordsLex`

```
typedef struct {
      SFuint8 word[LENWORD];
```

```
        SFint16 typeWord;
        SFint16 logo;
        SFint16 language;
        SFuint8 abbrev[LENABBREV];
} lexWord,*lexWordPtr;
```

The description of the different attributes of `lexWord` is as follows:

`word` contains the word added to the lexicon
`typeWord` contains the word type (part of speech) of the word (see the constants defined in A.8.a)
`logo` contains 1 if the word is an acronym ; otherwise 0.
`language` contains English or French. However, for proper nouns, it is always French.
`abbrev` contains the abbreviation of a word (possible only if the word is a proper noun)

5. The structure `dictDefinition` used by the functions `SFGLitDictWord` and `SFGLitDictOffset`

```
typedef struct {
        SFint16      nbrLettersWord;
        SFuint8      definitionWord[DICTMAXLENLINE];
} dictDefinition,*dictDefinitionPtr;
```

The description of the different attributes of `dictDefinition` is as follows:

`nbrLettersWord` gives, at the beginning of `definitionWord`, the number of letters in a word of which the definition is given afterward.
`definitionWord` is a string containing a word then its definition

C. List of functions

The parameters passed to entries are prefixed by « -> », and those returned are prefixed by « <-».
The parameters that are both passed to and returned entries are prefixed by « <-> ».

```
PrefixFunc(void)  SFGSetRegistrationKeys(SFuint8 keys[][kLenMaxKey],
        SFint16 nbrKeys);
```

This function allows one to enter activation keys for the Grammatica library.
-> `keys` contains an array of activation keys
-> `nbrKeys` indicates the number of keys contained in the array `keys`.

```
PrefixFunc(SFint16)  SFGSetRegistrationKeys2(SFuint8 keys[][kLenMaxKey],SFint16
        nbrKeys);
```

This function allows one to enter activation keys for the Grammatica library.
-> `keys` contains an array of activation keys
-> `nbrKeys` indicates the number of keys contained in the array `keys`.

<- return : the return of the function is the number of accepted keys

```
PrefixFunc(SFint16)  SFGSetRegistrationKeys3(SFuint8* key);
```

This function allows one to enter one activation key for the Grammatica library.
-> `key` contains the activation key

<- return : the return of the function is 1 if the key is accepted 0 otherwise

```
PrefixFunc(SFint16)   SFGIsFunctionRegistered(SFuint8* function);
```

This function allows one to know if a function is registered in the Grammatica library.
-> function is one of the constants defined in A.10

<- return : the return of the function is 1 if the function is registered 0 otherwise


```
PrefixFunc(void)   SFGInit(SFint16 language,SFint16 loadEnglishData);
```

This function initializes the engine of Grammatica
-> language contains the value of the working language desired (French, English, German or Spanish)
-> loadEnglishData equals 1 if the data used for checking in English should also be loaded ; otherwise it is 0. (no longer used).

```
PrefixFunc(void)   SFGEnd(void);
```

This function allows one to interrupt the engine. This frees up the memory blocks it occupied

```
PrefixFunc(void)   SFGSetLanguage(SFint16 language);
```

This function allows one to change the working language.
-> language equals either French, English, German or Spanish


```
PrefixFunc(void) SFGSetInterfLanguage(SFint16 interfaceLangage);
```

This function sets the language of the interface (and error messages)
        -> interfaceLangage equals either French, English, German or Spanish


```
PrefixFunc(void)   SFGSetSpecialChars(
        SFuint8 theSpecialChars[MAXNUMBERSPECIALCHARS]);
```
FRENCH OR ENGLISH ONLY

This function allows one to specify the special characters which are to be accepted in names and numbers
-> theSpecialChars is indexed by the ASCII code of each character. It contains zero if the character is not to be allowed in names and numbers. It contains one if the character is to be accepted.


```
PrefixFunc(void) SFGOptionNonUSUK(SFint16 optionNonUSUK);
```
ENGLISH ONLY

This function allows one to detect and to refuse American or British spellings of words
-> optionNonUSUK must be given the value of one of the following constants defined in A.3

```
PrefixFunc(void)   SFGOptionMed(SFint16 newOptionMed);
```
ENGLISH ONLY

This function allows one to activate the English medical dictionary.
-> newOptionMed must be set to one in order to activate the English medical dictionary.

```
PrefixFunc(void)   SFGOptionLangUniq(SFint16 newOptionLangUniq);
```
FRENCH AND ENGLISH ONLY

When the checking language is English, this function allows one to deactivate suggestions containing French words.
When the checking language is French, this function allows one to deactivate suggestions containing English words.
-> newOptionLangUniq must be set to one to deactivate suggestions containing words in the other language.

```
PrefixFunc(SFint16) SFGTestGrammarParagraph(SFuint8* theParagraph,
         SFint32 indexSelStart,
         SFint32 indexSelEnd,
         errorParagraphTab ParagraphTab,
         SFuint32 mistakesMask,
         SFint16 deepSearch,
         SFint16 stopUnknowCapWord,
         SFint16 accentsOnCapital,
         SFint16 substituteAbrev);
```

Grammar checking of a text.

-> `theParagraph` contains the text to check, in the form of a zero-terminated string.
-> `indexSelStart` contains the index of the first character of the selection
-> `indexSelEnd` contains the index of the last character of the selection
 (if the selection is empty, send `indexSelStart` and `indexSelEnd` as index of the first character following the insertion point of the cursor ; the engine launches checking from the beginning of the phrase containing `indexSelStart` and finishing with the phrase containing `indexSelEnd`)
<- `ParagraphTab` is a table containing the errors returned (see description in chapter B.2). One should then call `SFGClearParagraphTab` to clear the memory used by this table.
-> `mistakesMask` allows one to filter the errors that the user does not wish to receive (see the description of constants to be used in chapter A.5)
-> `deepSearch` equals one if one wishes extended search for suggestions corresponding to spelling errors ; otherwise zero
-> `stopUnknowCapWord` equals one if one wishes to flag unknown words beginning with a capital ; otherwise zero
-> `accentsOnCapital` equals one if one wishes to require that capitals included accents, when appropriate ; otherwise zero
-> `substituteAbrev` equals one if one wishes to require that abbreviations entered into the lexicon be automatically replaced ; oherwise zero. If `substituteAbrev` equals one, abbreviations will never be signaled to the user as an error, but the first suggestion will automatically replace the selected text. To allow this automatic replacement, `SFGTestGrammarParagraph` recalculates the index of errors which follows the replacing of the abbreviation..

<- return : the return of the function is the number of errors found in the paragraph.


```
PrefixFunc(SFint16) SFGTestSpellingParagraph(SFuint8* theParagraph,
         SFint32 indexSelStart,
         SFint32 indexSelEnd,
         errorParagraphTab ParagraphTab,
         SFint16 deepSearch,
         SFint16 stopUnknowCapWord,
         SFint16 accentsOnCapital,
         SFint16 substituteAbrev);
```

Spell-checking of a text (spelling errors, ligatures, incorrectly spelled acronyms, abbreviations
-> `theParagraph` contains the text to correct in the form of a zero-terminated string
-> `indexSelStart` contains the index of the first character of the selection
-> `indexSelEnd` contains the index of the last character of the selection
 (if the selection is empty, send `indexSelStart` and `indexSelEnd` as the index of the first character follwoing the insertion point comme; the engine will launch checking from the start of the sentence containing `indexSelStart` and will end it with the sentence `indexSelEnd`)
<- `ParagraphTab` is a table containing the errors returned (see the description in chapter B.2). One should then call `SFGClearParagraphTab` to clear the memory used by this table.
-> `deepSearch` equals one if one wishes to launch the extended search for suggestions corresponding to the spelling errors ; otherwise zero.
-> `stopUnknowCapWord` equals one if one wishes to flag unknown words beginning with a capital or a digit ; otherwise zero.

-> `accentsOnCapital` equals one if one wishes to require that capitals show accents (when the letter is accented) ; otherwise zero.

-> `substituteAbrev` equals one if one wishes to require that abbreviations entered in the lexicon be systematically replaced ; otherwise zero. If `substituteAbrev` equals one, these abbreviations will never be signaled to the user as an error, but the first suggestion will automatically replace the text indicated in the selection, `SFGTestGrammarParagraph` recalculates the indices of the errors that follow the replacement of the abbreviation..

<- return : the return of the function is the number of errors found in the paragraph :

```
PrefixFunc(void)  SFGClearParagraphTab(errorParagraphTab ParagraphTab,
          SFint16 nbErrorsParagraph);
```

After exploitation of a table of the type : `errorParagraphTab`, sent by `SFGTestGrammarParagraph`, by `SFGTestSpellingParagraph` or by `SFGPunctParagraph`, one must call `SFGClearParagraphTab` to free the space used by this table.

-> `ParagraphTab` contains a table of the type : `errorParagraphTab`, sent by `SFGTestGrammarParagraph`, by `SFGTestSpellingParagraph` or by `SFGPunctParagraph`

-> `nbErrorsParagraph` contains the number of errors contained in the table `ParagraphTab` (return of the function `SFGTestGrammarParagraph`, `SFGTestSpellingParagraph` or `SFGPunctParagraph`)

```
PrefixFunc(SFint16) SFGGetWordAnalysis(SFuint8* theParagraph,
          SFint32 *indexSelStart,
          SFint32 *indexSelEnd,
          SFint16 stopUnknowCapWord,
          SFint16 accentsOnCapital,
          SFint16 *nbHomographs,
          SFuint8 homographs[8][120],
          SFint16 *nbHomophones,
          SFuint8 homophones[12][120],
          SFint16 *nbLineGram,
          SFuint8 tableGram[10][120],
          SFint16* synonymCase,
          SFint16* nbSynonyms,
          SFuint8 synonyms[11][DICTMAXLENLINE],
          SFuint8 canonicalForm[LENWORD]);
```

FRENCH OR ENGLISH ONLY

Sends back the analysis of the first word included in a selection

-> `theParagraph` contains the text containing a word in the form of a zero-terminated string. This text must contain the sentence in which the word is located.

<-> `indexSelStart` contains when it is outgoing the index of the first character of the selection. Upon return, *indexSelStart contains the index of the first character of the word.

<-> `indexSelEnd` contains when it is outgoing the index of the last character of the selection. Upon return, *indexSelEnd contains the index of the last character of the word

-> `stopUnknowCapWord` equals one if one wishes to flag unknown words beginning with a capital ; otherwise zero.

-> `accentsOnCapital` equals one if one wishes to require that capitals show accents (when the letter is accented) ; otherwise zero.

<- `nbHomographs` returns the number of entries in `homographs` if the return of the function is `AnalysisOK` or `AnalysisWithMistakes`

<- `homographs` returns the homographs if the return of the function is `AnalysisOK` of the possible grammatical values of the word in the case of return of `AnalysisWithMistakes`

<- `nbHomophones` returns the number of entries in `homophones` if the return of the function is `AnalysisOK` or `AnalysisWithMistakes`

<- `homophones` sends back the homophones if the return of the function is `AnalysisOK` ou `AnalysisWithMistakes`

<- `nbLineGram` returns the number of entries in `tableGram` if the return of the function is `AnalysisOK`

<- `tableGram` sends back the grammatical analysis of the word if the return of the function is `AnalysisOK`

<- `synonymCase` can take the following values (`valid` if the return of the function is `AnalysisOK` or `AnalysisWithMistakes`) :

    if `synonymCase & AnalysisSynonymDefinition != 0` then

        `synonyms [0]` contains a definition of the word

    if `synonymCase & AnalysisSynonymDomain != 0` then

        `synonyms [1]` contains an indication about the nature of the word

<- `nbSynonyms` sends back the number of synonyms of the word if the return of the function is `AnalysisOK` or `AnalysisWithMistakes`

<- `synonyms` sends back, starting with index 2, the synonyms of the word

<- `canonicalForm` sends back the canonical form of the word .

<- return :

`AnalysisNoWordFound` -> no word can be selected

`AnalysisOK` -> correct analysis. The parameters `nbHomographs, homographs, nbHomophones, homophones, nbLineGram` and `tableGram` are valid

`AnalysisUnknownWord` -> the selected word is not recognized by Grammatica

`AnalysisWithMistakes` -> the selected word is included in a sentence in which there are one or more errors. The parameters `nbHomographs, homographs, nbHomophones et homophones` are valid


```
PrefixFunc(void) SFGParagraphSentencesStart(SFuint8* theParagraph,
        SFint16 nbMaxSentences, SFint16 *nbSentences,SFuint32
        sentencesStart[]);
```
Sends back the start offset of the sentences contained in a paragraph

-> `theParagraph` contains the text containing a word in the form of a zero-terminated string.

-> `nbMaxSentences` contains the maximal number of sentences for which the offset can be returned in `sentencesStart` (i.e. the number of elements of `sentencesStart`)

<- `nbSentences` sends back the number of sentences contained in the paragraph

<- `sentencesStart` sends back the start offset of the sentences contained in the paragraph


```
PrefixFunc(void) SFGCountParagraphSentencesStart(SFuint8* theParagraph,
        SFint16 *nbSentences);
```
Sends back the number of sentences contained in a paragraph

-> `theParagraph` contains the text containing a word in the form of a zero-terminated string.

<- `nbSentences` sends back the number of sentences contained in the paragraph


```
PrefixFunc(SFint16) SFGGetIndexWord(SFuint8* theParagraph,
        SFint32 *indexSelStart,
        SFint32 *indexSelEnd);
```

This function will, beginning with any selection, select the first word it includes.

This is necessary before calling the functions SFGInfinitive, SFGConjugate, SFGPlural, SFGLitDictWord, as well as the functions of the lexicon that asks for a word.

-> `theParagraph` contains the text containing the word in the form of a zero-terminated string. This text must contain the sentence in which the word is located..

<-> `indexSelStart` contains when it is outgoing the index of the first character of the selection. Upon return, *indexSelStart contains the index of the first character of the word

<-> `indexSelEnd` contains when it is outgoing the index of the last character of the selection. Upon return, *indexSelEnd contains the index of the last character of the word

<- return: the function sends back zero if no word has been selected ; otherwise one

```
PrefixFunc(SFint16) SFGInfinitive(SFuint8 *theWord,
        SFint16 *nbSol,
        SFuint8 solutions[3][LENWORD],
        SFint16 accentsOnCapital,
        SFint16 language);
```

This function sends back the infinitive(s) of a conjugated verb form.
-> `theWord` contains the conjugated verb form in the form of a zero-terminated string.
<- `nbSol` contains in the return the number of solutions
<- `solutions` contains in the return the solutions
-> `accentsOnCapital` equals one if one wishes to require that accented capitals have their accents ; otherwise zero
-> `language` allows to specify the language : either `English` or `French`

<- return: the return of the function is one of the constants defined in A.7.a


```
PrefixFunc(SFint16) SFGConjugate(SFuint8 *theWord,
        SFint16 mode,
        SFint16 *nbSol,
        SFuint8 solutions[6][LENWORD],
        SFint16 *intransitiveVerb,
        SFint16 accentsOnCapital,
        SFint16 language);
```

This function sends back the conjugations of a verb infintive
-> `theWord` contains the infinitive in the form of a zero-terminated string
-> `mode` is among the constants defined in A.7.b
<- `nbSol` contains in the return the number of solutions
<- `solutions` contains in the return the solutions
<- `intransitiveVerb` equals one in the return if the verb is intransitive; otherwise zero. In German, it is one if the verb is strong; otherwise zero.
-> `accentsOnCapital` equals one if one wishes to require that accented capitals have their accents ; otherwise zero
-> `language` allows to specify the language : either English or French

<- return : the return of the function is one of the constants defined in A.7.a


```
PrefixFunc(SFint16) SFGPlural(SFuint8 *theWord,
        SFint16 *nbSol,
        SFuint8 solutions[2][LENWORD],
        SFint16 accentsOnCapital,
        SFint16 language);
```

This function sends back the plural(s) of a noun or adjective.
-> `theWord` contains the noun or adjective
<- `nbSol` contains in the return the number of solutions
<- `solutions` contains in the return the solutions
-> `accentsOnCapital` equals one if one wishes to require that accented capitals have their accents ; otherwise zero
-> `language` allows to specify the language : either English or French

<- retour : the return of the function is one of the constants defined in A.7.a. Every return value other than one of the errors indicated in A.7.a indicates that the plural has been found.

```
PrefixFunc(SFint16) SFGSingular(SFuint8 *theWord,
        SFint16 *nbSol,
```

```
        SFuint8 solutions[2][LENWORD],
        SFint16 accentsOnCapital,
        SFint16 language);
```

FRENCH OR ENGLISH ONLY

This function sends back the singular(s) of a noun or adjective
-> `theWord` contains the noun or adjective
<- `nbSol` contains in the return the number of solutions
<- `solutions` contains in the return the solutions
-> `accentsOnCapital` equals one if one wishes to require that accented capitals have their accents ; otherwise zero
-> `language` allows to specify the language : either English or French

<- return: the return of the function is one of the constants defined in A.7.a. Every return value other than one of the errors indicated in A.7.a indicates that the singular has been found

```
PrefixFunc(void) SFGSetSpaces(SFuint16 numOfSpaces,
        SFuint8 spaces[MAXNUMSPACES],
        SFint16 language);
```
FRENCH OR ENGLISH ONLY

Allows to set the spaces that Grammatica is to take into account. Be careful not to use 13 (CR)
-> `numOfSpaces` indicates the number of spaces contained in the table `spaces`
-> `spaces` contains the spaces
-> `language` allows to specify the language : either English or French

```
PrefixFunc(void) SFGGetSpaces(SFuint16 *numOfSpaces,
        SFuint8 spaces[MAXNUMSPACES],
        SFint16 language);
```
FRENCH OR ENGLISH ONLY

Returns the spaces taken into account by Grammatica
<- `numOfSpaces` indicates the number of spaces contained in the table `spaces`
<- `spaces` contains the spaces taken into account by Grammatica
-> `language` allows to specify the language : either English or French

```
PrefixFunc(void) SFGDefaultSpaces(void);
```

FRENCH OR ENGLISH ONLY

Indicates to Grammatica to use default spacing

```
PrefixFunc(void) SFGSetPunctSigns(SFuint16 numOfPunctSigns,
        anPunctSignTab punctSigns,
        SFint16 language);
```

Allows to set the punctuation marks that Grammatica is to take into account. Be careful not to use 13 (CR)
-> `numOfPunctSigns` indicates the number of punctuation marks contained in the table `punctSigns`
-> `punctSigns` contains the punctuation marks and their constraints
-> `language` allows to specify the language : either English or French

```
PrefixFunc(void) SFGGetPunctSigns(SFuint16 *numOfPunctSigns,
        anPunctSignTab punctSigns,
        SFint16 language);
```

FRENCH OR ENGLISH ONLY

Returns the punctuation marks taken into account by Grammatica.

<- numOfPunctSigns indicates the number of punctuation marks contained in the table punctSigns

<- punctSigns contains the punctuation marks and their constraints

-> language allows to specify the language : either English or French

```
PrefixFunc(void) SFGDefaultPunctSigns(void);
```

FRENCH OR ENGLISH ONLY

Indicates to Grammatica to use default punctuation with their constraints

```
PrefixFunc(SFint16) SFGPunctParagraph(SFuint8* theParagraph,
        SFint32 indexSelStart,
        SFint32 indexSelEnd,
        errorParagraphTab ParagraphTab);
```

FRENCH OR ENGLISH ONLY

Checking the punctuation of a text.

-> theParagraph contains the text to correct in the form of a zero-terminated string

-> indexSelStart contains the index of the first character of the selection

-> indexSelEnd selection contains the index of the last character of the selection

 (if the selection is empty, send indexSelStart and indexSelEnd as index of the first character following the insertion point ; the engine launches checking from the beginning of the sentence containing indexSelStart and ends it with the sentence containing indexSelEnd)

<- ParagraphTab is a table of the errors returned (see the description in chapter B.2). One should then call SFGClearParagraphTab to free the memory used by the table.

<- return: the return of the function is the number of errors found in the paragraph

```
PrefixFunc(SFint16) SFGOpenLex(SFuint8 *fileName);
```

Opening of the lexicon. It will be created if it does not already exist.

-> fileName contains the name of the file. On Macintosh, the name of the file may be partial as long as the default volume is correctly set before the call.

<- return: the return of the function can equal LexNoErr , LexIOError, LexMemoryError or LexIncorrectVersion

```
PrefixFunc(SFint16) SFGCloseLex(SFuint8 *fileName);
```

Closing the lexicon.

-> fileName contains the name of the file. On Macintosh, the name of the file may be partial as long as the default volume is correctly set before the call.

 <- return: the return of the function can equal LexNoErr or LexIOError

```
PrefixFunc(SFint16) SFGAnalyseWordLex(SFuint8 *theWord,
        SFint16 *logo,
        SFint16 *type,
        SFint16 *gender,
        SFint16 *pluralForm,
        SFint16 *intransitiveVerb,
        SFuint8 *modelConj,
        SFint16 language);
```

Automatic analysis of a word in order to automatically propose choses to the end user
-> `theWord` contains the word to analyse
<- `logo` contains 1 upon return if the word is proposed as an acronym ; otherwise zero. This parameter is only valid if `type` equals `ProperNoun`.
<- `type` contains upon return the proposed type (part of speech) for the word
<- `gender` contains upon return the proposed gender the word
<- `pluralForm` contains upon return the proposed number (singular or plural) of the word
<- `intransitiveVerb` equals 1 if the proposed verb is transitive ; otherwise 0. The `type` must be `Verb` and the `language` must be `French` for this parameter to be useful.
<- `modelConj` proposes a conjugation model. `Type` must equal `Verb` for this parameter to be useful
-> `language` allows to specify the language : either English or French
<- return: the return of the function can equal `LexNoErr` or `LexAlreadyInDict`

```
PrefixFunc(SFint16) SFGConjugateLex(SFuint8 *theWord,
        SFint16 mode,
        SFint16 *nbSol,
        SFuint8 solutions[6][LENWORD],
        SFint16 intransitiveVerb,
        SFuint8 *modelConj,
        SFint16 language);
```

`SFGConjugateLex` allows to conjugate a verb not yet added to the lexicon.
-> `theWord` contains the infinitive
-> `mode` takes its values as in the function `SFGConjugate`
<- `nbSol` contains the number of solutions
<- `solutions` contains the solutions
-> `intransitiveVerb` equals 1 if the proposed verb is transitive ; otherwise 0.
-> `modelConj` is the proposed conjugation model.
-> `language` allows to specify the language : either English or French
<- return: the return of the function can return the following values : `LexNoErr, LexUnknownConjModel, LexConjModelNotVerb, LexConjModelNotVerbSameConj, LexConjModelNotInfinitive, LexPastParticipleInvariable, LexConjModelNotValid` or `LexAlreadyInDict`

```
PrefixFunc(SFint16) SFGInsertLex(SFuint8 *theWord,
        SFint16 logo,
        SFint16 type,
        SFint16 gender,
        SFint16 pluralForm,
        SFint16 intransitiveVerb,
        SFuint8 *modelConj,
        SFuint8 *abbrev,
        SFint16 language);
```

`SFGInsertLex` allows one to add a word to the lexicon of Grammatica
-> `theWord` contains the word to insert
-> `logo` contains one if the proper noun is to be added as an acronym ; otherwise 0.
-> `type` contains the word type (part of speech) to be added, using the constants defined in A.8.a
-> `gender` contains the gender of the word to be added, using the constants defined in A.8.a
-> `pluralForm` contains the number (singular or plural) of the word to be added, using the constants defined in A.8.a
-> `intransitiveVerb` equals 1 if the proposed verb is transitive ; otherwise 0. The `type` must be `Verb` and the `language` must be `French` for this parameter to be useful
-> `modelConj` contains the conjugation model to use. `Type` must equal `Verb` for this parameter to be useful
-> `abbrev` contains the abbreviation of the proper noun. This parameter is only valid if `type` equals `ProperNoun`.
-> `language` allows to specify the language : either English or French
<- return: the return of the function can return the error codes defined in A.8.c with the exception of `LexIncorrectVersion` and `LexWordNotFound`

The case of the error `LexAlreadyDefined` is special. In this case one must ask the user if he wishes to modify the lexicon by erasing the pre-exisiting form. If he responds OK, one must call the function `SFGChangeLex`

Important comments

In English, adjectives necessarily have a number equal to `Irregular,` which is to say that they are always invariable.
In English, nouns and adjectives have no gender. On must use the gender `MasculineFeminine` (both masculine and feminine).

Only proper nouns (type = `ProperNoun`) can be acronyms (logos). An acronym is a proper noun that may contain spaces, punctuation marks, and the typography of which must be respected in a text. If a proper noun is an acronym, its number can only be `Irregular`, `Plural` or `SingularPlural`.
One can only associate abbreviations to proper nouns (type = `ProperNoun`).

Proper nouns are defined in French. The engine recognizes French proper nouns during French or English grammar and spell-checking.

```
PrefixFunc(SFint16) SFGChangeLex(SFuint8 *theWord,
        SFint16 logo,
        SFint16 type,
        SFint16 gender,
        SFint16 pluralForm,
        SFint16 intransitiveVerb,
        SFuint8 *modelConj,
        SFuint8 *abbrev,
        SFint16 language);
```

`SFGChangeLex` applies only when `SFGInsertLex` has sent back `LexAlreadyDefined` and the user has decided to modify a pre-existing word in the lexicon. The function then takes exactly the same parameters as those used in the calling of `SFGInsertLex` which has returned `LexAlreadyDefined`

<- return: the return of the function can equal `LexNoErr, LexErrorInLexicon, LexMemoryError, LexNotOpen` or `LexIOError`

```
PrefixFunc(SFint32) SFGGetListWordsLex(lexWordPtr *lexicon,
        SFint32 languageSelection);
```

SFGGetListWordsLex sends back the list of the lexicon's words. This list is sorted in lexicographical order.
<- `lexicon` contains upon return the list of words (memory allocation is handled by `SFGGetListWordsLex`)
-> `languageSelection` indicates the language of the words which one desires to get in return. It can equal :
  - `EnglishSelection` to return only English words;
  - `FrenchSelection` to return only French words;
  - `EnglishSelection+FrenchSelection` to return both French and English words;
  - `allLanguagesSel` to return all words in all languages
<- return: the return of the function can equal either the number of words returned present in the lexicon `lexicon` (0 or more) or `LexMemoryError`

Careful : proper nouns are always returned, regardless of the language selected. The attribute `Language` in `LexWord` thus equals `French`

```
PrefixFunc(void)SFGDisposeListWordsLex(lexWordPtr lexicon);
```

`SFGDisposeListWordsLex` allows to clear from memory the list of words sent back by `SFGGetListWordsLex`.

`-> lexicon` contains this list of the lexicon's words.

```
PrefixFunc(void)  SFGGetValuesWordLex(SFuint8 *theWord,
        SFint16 type,
        SFint16 *logo,
        SFint16 *gender,
        SFint16 *pluralForm,
        SFint16 *intransitiveVerb,
        SFuint8 *modelConj,
        SFuint8 *abbrev,
        SFint16 language);
```

`SFGGetValuesWordLex` allows one to get the values of a word from the lexicon, giving it along with its type.

`-> theWord` contains the word to be searched in the lexicon

`-> type` contains the word type (part of speech), using the constants defined in A.8.a

`<- logo` contains 1 upon return if the word is an acronym ; otherwise zero. This parameter is only valid if `type` equals `ProperNoun`.

`<- gender` contains upon return the gender of the word, using the constants defined in A.8.a

`<- pluralForm` contains upon return the number (singular or plural) of the word, using the constants defined in A.8.a

`<- intransitiveVerb` contains 1 upon return if the verb is transitive ; otherwise 0. The `type` must be `Verb` and the `language` must be `French` for this parameter to be useful

`<- modelConj` contains upon return the conjugation model to use. `Type` must equal `Verb` for this parameter to be useful

`<- abbrev` contains upon return the abbreviation of the proper noun. This parameter is only valid if `type` equals `ProperNoun`

`->` language allows to specify the language : either English or French

```
PrefixFunc(SFint16) SFGRemoveLex(SFuint8 *theWord,SFint16 type,SFint16
        language);
```

`SFGRemoveLex` can delete a word from the lexicon, giving it along with its type

`-> theWord` contains the word to delete

`-> type` contains the word type, using the constants defined in A.8.a

`->` language allows to specify the language : either English or French

`<-` return: the return of the function can equal `LexNoErr, LexErrorInLexicon, LexNotOpen, LexWordNotFound` or `LexIOError`

```
PrefixFunc(SFint16) SFGLitDictWord(SFuint8 *theWord,
        dictDefinitionPtr dictDefinitions,
        SFint16 tabDefHeigth,
        SFint16 *returnedTabDefHeigth,
        SFint32 *tabDefOffsetWord);
```
FRENCH ONLY

`SFGLitDictWord` sends back a table of definitions beginning with `theWord` or, if this word does not exist, with the word of the dictionary that immediately follows it.

`-> theWord` contains the word

<-> dictDefinitions contains a table of tabDefHeigth elements typed by dictDefinition. Upon return, the table is filled with returnedTabDefHeigth elements
-> tabDefHeigth is the number of elements requested from the dictionary. It is the size of dictDefinitions.
<- returnedTabDefHeigth contains upon return the number of elements returned in dictDefinitions
<- tabDefOffsetWord contains upon return the global index in the dictionary of the word contained in theWord or, if this word doesn't exist, of the word of the dictionary which immediately follows it.
<- return: the return of the function can equal DictMemoryError, DictWordAbsent or DictWordPresent

```
PrefixFunc(SFint16) SFGLitDictOffset(SFint32 tabDefOffsetAsked,
        dictDefinitionPtr dictDefinitions,
        SFint16 tabDefHeigth,
        SFint16 *returnedTabDefHeigth);
```

FRENCH ONLY

SFGLitDictOffset sends back a table of definitions beginning at the index tabDefOffsetAsked (global index of the dictionary)
-> tabDefOffsetAsked is the index of the beginning of the requested definitions
<-> dictDefinitions contains a table of tabDefHeigth elements typed by dictDefinition. Upon return, the table is filled with returnedTabDefHeigth elements
-> tabDefHeigth is the number of elements requested from the dictionary. It is the size of dictDefinitions.
<- returnedTabDefHeigth contains upon return the number of elements returned in dictDefinitions
<- return: the return of the function can equal DictMemoryError, DictWordAbsent or DictWordPresent

```
PrefixFunc(SFint16) SFGStemAnalysis(SFuint8* theParagraph,
        SFint16 accentsOnCapital,
        SFint16 nbMaxWords,
        SFint16 *nbWords,
        SFuint8 Words[][LENWORD]);
```

FRENCH OR ENGLISH ONLY

This function sends back the canonical form of all nouns and verbs found in the paragraph.
-> theParagraph contains the text to stem, in the form of a zero-terminated string.
-> accentsOnCapital equals one if one wishes to require that accented capitals have their accents ; otherwise zero
-> nbMaxWords contains the maximum words that the Words array can contain
<- nbWords contains in the return the number of words (nouns and verbs) returned in the Words array
<- Words contains the canonical form of all nouns and verbs found in the paragraph.

```
PrefixFunc(SFint16) SFGCountMaxStemAnalysisForms(SFuint8* theParagraph,
        SFint16 accentsOnCapital,
        SFint16 *nbWords);
```

FRENCH OR ENGLISH ONLY

This function sends back the number of nouns and verbs found in the paragraph
-> theParagraph contains the text to stem, in the form of a zero-terminated string.
-> accentsOnCapital equals one if one wishes to require that accented capitals have their accents ; otherwise zero
<- nbWords contains in the return the number of words (nouns and verbs) found in the paragraph

```
PrefixFunc(void)  SFGSetStraightQuotes(SFint16 straightQuotes);
```

FRENCH OR ENGLISH ONLY

This function allows for the use of straight quotes instead of smart quotes in the suggestions list.

-> `straightQuotes` must be set to one for straight quotes, and to zero for smart quotes

By default, Grammatica uses smart quotes for the suggestions.

`PrefixFunc(void)   SFGSetNbWordsLongSentence(SFint16 nbWordsLongSentence);`

FRENCH OR ENGLISH ONLY

This function allows one to set the number of words used to flag a sentence as long or not.
-> `nbWordsLongSentence` must be set to the minimum number of words which will flag if a sentence is long.

By default, Grammatica assumes that sentences longer than 50 words will be flagged as long.