**RC23976 (W0607-010) July 7, 2006 (updated 11/14/2007)**
**Computer Science**

# IBM Research Report

## A Formal System for Slot Grammar

**Michael C. McCord**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# A Formal System for Slot Grammar

Michael C. McCord
IBM T. J. Watson Research Center

**Abstract**

The purpose of this document is to describe a formalism, **SGF**, for Slot Grammar (SG). SGF consists of three subformalisms, corresponding to different components of an SG: (1) The centerpiece is **SSF**, the SG syntactic rule formalism – the core of an SG. (2) An auxiliary formalism is **SFF**, the SG *feature formalism* – which allows the grammar writer to specify features and relationships among them. (3) **SLF** is the SG lexical formalism. SLF is described in another report, and we concentrate on SSF and SFF in the current report. SSF follows the central theme of Slot Grammar – that analysis (or parsing) consists of slot-filling. A typical rule of SSF deals with filling a slot $S$, where the proposed filler (or modifier) phrase $M$ and the higher phrase $H$ (the "owner" of the slot $S$) have been tentatively chosen by the parsing algorithm. The *main* function of such a slot-filling rule is simply to decide whether $M$ can indeed fill $S$ in $H$. But the rule can do other things, such as setting features in $H$ or $M$, "raising" slots from $M$ into $H$, or affecting a numerical parse score for $H$ with its new modifier. The SSF formalism makes it easy to "explore" the phrases $M$ and $H$, examining features or other ingredients in any subphrases within these phrases, or even querying alternative parses for existing word spans. There is a rich and flexible set of operators for examining the components of parse trees. The SSF rules are in a sense *object-oriented*, because in any part of a rule there is always an implicit *phrase (node) in focus* – the phrase being examined by that part of the rule. Selection operators need not mention the current phrase in focus because it is implicit, so they can be more succinct. There are convenient operators for shifting the focus, like going down to a modifier, which can proceed non-deterministically, depending on conditions placed on the modifier.

## 1 Introduction

The purpose of this report is to describe a formalism, **SGF**, for Slot Grammar (SG). SGF consists of three subformalisms, corresponding to different components of an SG: (1) The centerpiece is **SSF**, the SG syntactic rule formalism – the core of an SG. (2) An auxiliary formalism is **SFF**, the SG *feature formalism* – which allows the grammar writer to specify features and relationships among them. (3) **SLF** is the SG lexical formalism.

SLF is discussed in the companion report [9], "The Slot Grammar Lexical Formalism". This describes SG lexicons in enough detail that the reader should be able to create such a lexicon for a new language.

In the current report, we concentrate on SSF and SFF. With access to the SG "shell" (the general framework software for SG), which can interpret SGF, the reader should be able use the information given here to write the syntactic rule component for a SG for a specific natural language, and use it in parsing.

There is a second companion report, [10], "Using Slot Grammar". This contains a general description of SG and of an API for it, with an emphasis on how to use SG parsers in applications. The two companion reports and the current one form a sort of triad which should give a good picture of the current state of Slot Grammar, especially the syntax rules and the lexicon. The three reports complement one another, and one can get the full picture best by reading all three. Nevertheless, each of the reports is written in a fairly self-contained way.

Morphology is also an important language-specific component for a grammar. Generally Slot Grammar has less to say about morphology than it does about syntax and the lexicon. However, the features specified in the SFF formalism are often morphological features which a morphological analyzer produces, so SFF serves as a bridge between morphological analysis and syntactic analysis. The SG shell does contain a framework for doing morphology, but we will save a description of that till later. The report [11] contains a description of an earlier version. In any case, it is often feasible to interface an independent morphological analyzer to the rest of SG.

The syntax rule formalism SSF follows the central theme of Slot Grammar – that analysis (or parsing) consists of slot-filling. A typical rule of SSF deals with filling a slot $S$, where the proposed filler (or modifier) phrase $M$ and the higher phrase $H$ (the "owner" of the slot) have been tentatively chosen by the parsing algorithm. The *main* function of such a slot-filling rule is simply to decide whether $M$ can indeed fill $S$ in $H$. But the rule can do other things, such as setting features in $H$ or $M$, "raising" slots from $M$ into $H$, or affecting a numerical parse score for $H$ with its new modifier.

For doing these things, the SSF formalism makes it easy to "explore" the phrases $M$ and $H$ (which will normally have modifier structure of their own), examining features or other ingredients in any subphrases within these phrases, or even querying alternative parses for existing word spans. There is a rich and flexible set of operators for examining the components of phrases (parse trees).

The SSF rules are in a sense *object-oriented*, because in any part of a rule there is always an implicit *phrase (or node) in focus* – the phrase being examined by that part of the rule. Selection operators need not mention the current phrase in focus because it is implicit, so they can be more succinct. There are convenient operators for shifting the focus, like going *up* in the tree or *down* to a modifier. Going down can act non-deterministically, depending on conditions

placed on the modifier, including the condition that it fill a certain slot. The current focus can be named in a variable and returned to later by reference to that variable.

SSF is designed so as to abstract away as much as possible from the implementational details of the data structures used by the shell. For instance the current implementation, in C, represents morphosyntactic features as bit positions in fixed-length bit strings (for the sake of efficiency), and represents semantic features in an open-ended way as lists of atomic symbols in a Lisp-like data structure. But SSF makes no references to these matters of implementation, and all features are seen in the rules simply as atomic symbols in Lisp-like expressions used for SSF rules. This more formal, abstract view of the rules makes it easier to create different implementations of Slot Grammar.

The SG system expects the SSF rules for a language to be in a file named `X.gram`, where $X$ is the standard two-letter code for the language. So for example the syntax rules for English should be in `en.gram`, and those for German should be in `de.gram`. We will also use "`X.gram`" more abstractly to refer to the set of syntax rules for language $X$. The SFF rules should be in a file named `Xfeas.lx` – for instance `enfeas.lx` for English.

To get `X.gram` and `Xfeas.lx` used, one should have access to a *Slot Grammar system* **XSG** for language $X$ – normally in the form of an executable or a library file, like `XSG.exe` or `XSG.dll`. **XSG** contains the SG shell plus a few minor adaptations for language $X$. In naming these language-specific SG systems, we usually take $X$ to be the first letter of the language name, or more if necessary for disambiguation, so that e.g. we have **ESG** for English and **GSG** for German.

At run time, when **XSG** initializes, it finds the file `X.gram` and reads it in, storing the rules in a partially preprocessed form. During parsing, the parsing algorithm (contained in the module **XSG**) interprets the rules in `X.gram` (in their preprocessed form). There is also a compiler that converts `X.gram` to C code, which can then be compiled in with **XSG**, though that compiler is not currently up to date with the latest form of SSF (I will return to it to update it).

Also, when **XSG** initializes, it finds and reads in other language-specific files. These include, above all, the lexicons for language $X$. As indicated above, the format of SG lexicons is described in [9]. In addition, **XSG** reads in a general ontology lexicon `ont.lx`.

The syntax rule component `X.gram` and the lexicons have a similar basic syntax, based on Cambridge Polish (Lisp-like).

Although **XSG** uses language-specific rules and data like `X.gram` and the lexicons, some amount of **XSG** is language-universal and resides in the (SG) *shell*. Generally we try to share concepts and notations across languages as much as possible.

In Section 2 of this report we describe on a conceptual, implementation-

independent level the form of SG parse trees. Section 3 contains a general description of the SG parsing algorithm, and Section 4 discusses a specific aspect of the parsing algorithm – the parse scoring system. Section 5 describes features and the SFF formalism. In Section 6 we provide an overview of SSF and its basic characteristics. Then in Section 7 we give a detailed specification of SSF rules, going through the "built-in" operators and their meanings. Finally, in Section 8 we discuss tools available in the shell (in *X***SG**) for grammar development – tracing, debugging and testing.

## 2 Slot Grammar analysis structures

In this section we give a conceptual view of SG analysis structures (parse trees) – the objects that are explored by SSF slot-filling rules.

Slot Grammar is dependency-oriented, and each parse tree (node) has a headword together with information about the head or the phrase as a whole, plus a list of left modifiers of the head and a list of right modifiers. The modifiers are (recursively) trees of the same form. Let us call such SG trees *phrases* – although this term might also be used to mean a text string that the phrase analyzes.

The information in a phrase (node) includes the following ingredients:

1. The headword in its text form (which may include capitalization). Such "words" can be multiwords, as determined by the lexicon or other parts of the system.
2. The word number for the headword in the segment. We explain below just what word numbers are.
3. The citation form (or lemma) for the headword.
4. The sense name for the headword. This is by default the citation form followed by a number, but the SG lexicon can specify an arbitrary sense name.
5. The list of features of the node. These are identifiers that include part of speech, morphosyntactic features, and semantic features. Most of the features come from the headword and its morpholexical analysis – from the features attached to its chosen sense in the lexicon, and from its inflectional features. These are just viewed as features shared by the head and the phrase as a whole, in the spirit of dependency grammar. But some features provide information about the phrase as a whole.
6. The predicate argument list for the word sense. This is a list of pairs arising from the lexical slot frame of the word sense, each pair consisting of a complement slot from the frame and its filler, or nil if there is no filler in the parse tree. The slots are listed in the same order as in the lexical slot frame. Slot-filler arguments should be viewed as *deep* or *logical* arguments. Logical subjects and objects are shown for passive past participle

verbs, and remote fillers of slots are shown in the case of extraposition and coordination.

7. The lists of left modifiers and right modifiers of the head, given in left-to-right surface order. Each modifier is recursively a phrase itself.
8. The mother node of the current node – which is nil in case the current node is the top node, and otherwise is a phrase.
9. The surface slot (name) filled by the current node, or `top` if the current node is the top node.
10. The slot option for the slot-filling just named, or `nop` if the current node is the top node. See [10, 9] for the description of slot options. They are closely associated with the POS of the filler phrase.
11. The left and right boundary numbers of the current phrase. We explain below what boundary numbers are exactly.
12. A parse score (a real number) for the current node, representing roughly how likely it is that this phrase is a good analysis of the text string it spans.
13. A list of tests to be done on the node – its *phrase tests* – when it is used as a filler, or is taken as the top node. In most cases, this list will be empty. We describe this in more detail below.

A parse node *includes* these fields, but there are actually other fields that are relevant to the parsing algorithm or efficient implementation but are not necessary for SSF to work with. So we will not discuss all of the parse node fields.

Let us explain what word numbers and boundary numbers are. After doing tokenization of a segment, the SG shell organizes the tokens into two disjoint groups, with different data structures.

The first group is an array of the "word tokens" in the segment. These are tokens, like normal words, numbers, entity tags (e.g. of the form `&xxx;`) and other symbols that can be typical nodes of parse trees. If the headword of a parse tree is one these word tokens, then its word number is just the index of the word token in the array of segment word tokens, starting with 1 for the first word token. The shell can also agglomerate word tokens into multiwords which become headwords of parse tree nodes. This can happen because of multiword lexical entries or because of other rules in the shell. When a multiword is the headword of a phrase node, its word number is still based on the count in the *original* word tokens of the segment – it is the word number of the index word of the multiword. So if "data base" is a multiword, its word number will be that of "base", and for "attorney general" it will be the word number of "attorney".

Normal punctuation symbols and tags that act like punctuation or that bracket text (like font change tags) form the second group of tokens; we shall

just refer to these as "punctuation (tokens)" in this context. Punctuation tokens should be thought of as stored in the "interstices" between word tokens. The interstices are indexed by integers that run from 0 to the word number for the last word token of the segment. The interstice indexed by 0 holds the punctuation, as a list of tokens, that occurs before the first word. For $i \geq 1$, the interstice indexed by $i$ holds the list of punctuation tokens occurring right after word token $i$.

Parse trees do not normally exhibit punctuation tokens, but SSF allows one to access them, as we will see in Subsection 7.11 below. However, some punctuation tokens, like commas, can serve as coordinating conjunctions, and then they do show in parse trees (as well as in the interstices). We discuss this in Subsection 7.8.

The left and right boundary numbers of phrases are always interstice numbers. The left boundary is one less than the word number of the leftmost word token appearing in the phrase. The right boundary is the word number of the rightmost word token appearing in the phrase.

SG parse trees show both deep (or logical) and surface information in the same tree. Ingredients (4) and (6) in the preceding list provide deep structure information and the others provide surface structure information. This is illustrated in Figure 1, for the sentence *Mary gave a book to John*, where we show a standard SG parse tree display.

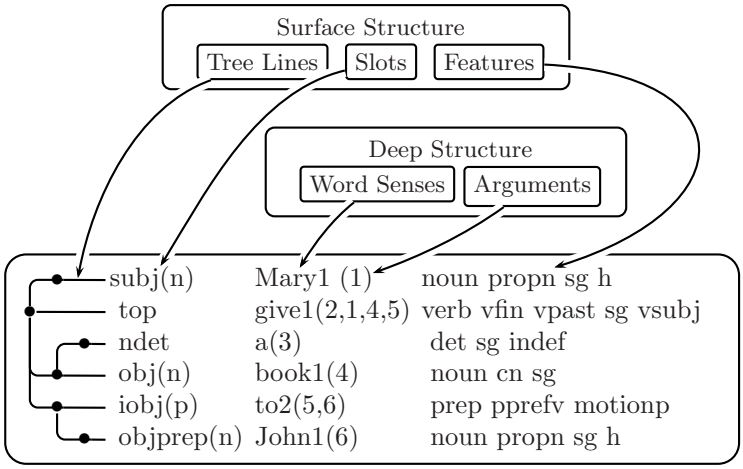



Figure 1: Ingredients of a Slot Grammar analysis structure

Note then that the surface structure of the sentence is shown in the tree lines and the slots on the left, and the features on the right. And the deep (or logical) structure is shown in the middle section through the word sense predicates and their arguments. In this display of predicate arguments, the first argument is

always the word number of the node (ingredient (2) above); this can be viewed like an event or entity argument for the predication. The remaining arguments are the word numbers of the complement slot fillers, or `u` if there is no filler, given in the order of the lexical slot frame.

The next example, in Figure 2, showing the parse for the passive sentence *The book was written by John*, illustrates deep arguments better. We have also applied a parse display control flag `predargslots` which causes the display to show both slots and filler numbers in predicate argument lists.

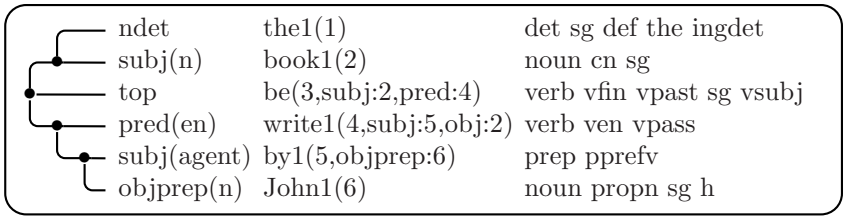



Figure 2: Parse of a passive sentence

Note that the word sense predication write1(4,subj:5,obj:2) appropriately has node 5, *by John* (and from this *John*), as its logical subject, and has node 2, *the book*, as its logical object. (In this display we have omitted some unfilled complement slots for *book* and *write*, for the sake of greater readability.)

## 3 Basic nature of the parser

The first step of parsing is morpholexical analysis. This is done in an initial pass through the word tokens of the sentence. Each such token is given all its possible morpholexical analyses, expressed as one-word *starter* phrases, using the phrase data structure described in the preceding section. The "one-word" heads of these phrases may be multiwords.

Then, for syntactic analysis proper, the parser makes a second pass, going through the starter phrases left-to-right, and combining them in the manner of a bottom-up chart parser. At each step, the parser builds up all possible phrasal analyses of substrings of the words that it has looked at so far. The *chart* is the collection of these built-up phrases. As a new starter phrase $Q$ is encountered, the parser attempts to *combine* $Q$ with an existing phrase $P$ in the chart such that the right boundary of $P$ is the left boundary of $Q$. This combination of phrases is done either by letting $P$ fill a slot in $Q$ or vice versa. (Both are tried, and both may be possible and be stored in the chart.) If a combination of $P$ and $Q$ is possible, then we have a new phrase $R$ in the chart whose left boundary is that of $P$ and whose right boundary is that of $Q$. Then the process of combination continues with $R$ recursively on to the left. We try to combine $R$ further with an existing phrase just to its left, and so on, recursively. When

all the starter phrases have been processed in this way, then a parse analysis of the whole segment is a phrase in the chart that spans the whole segment, and whose obligatory slots have been filled. There may be more than one analysis, of course.

The slots used in slot-filling are determined as follows. When a new starter phrase is formed, its *complement* slots, as determined from the morpholexical analysis of the headword, are stored in an *available slots* list of the data structure for the phrase. When it is time to try filling a slot of a phrase $H$ with an adjacent phrase $M$, one can choose a complement slot $S$ of $H$ from its available slots list and try that slot. For testing whether this slot-filling can succeed, the parsing algorithm consults $X$`.gram` for the *complement slot rules* associated with slot $S$. There are typically several rules for $S$, each indexed by the name of $S$. These rules are applied in the order they are listed in $X$`.gram`, but stopping when and if one succeeds. Each rule has a body which is applied as a test – plus actions it may have for setting new features and slots in the new version of $H$ with $M$ as a modifier, and new features of $M$. If this slot-filling succeeds, then the available slots list of the combined phrase is that of $H$, with $S$ removed, plus possibly new slots *raised* from $M$. (More on slot raising in Subsection 7.5.) Each complement slot owned by a phrase can be filled at most once.

On the other hand, we may try filling an *adjunct* slot of $H$ with $M$. In this case, the parser keys off the POS (part of speech) of the filler phrase $M$ and consults $X$`.gram` for the *adjunct slot rules* associated with that POS. There may be several rules associated with (and indexed by) that POS. The parser tries them all, non-deterministically, using each successful rule to build a new version of $H$ with $M$ as a modifier. It is up to each adjunct slot rule to specify an adjunct slot name and slot option name that are suitable for the filling. The rule can do this on the basis of not only the POS of $M$ (the index of the rule), but also other features of $M$ and $H$, and in general other characteristics of $M$ and $H$ obtained from exploring those phrases. Again, the rule acts as a test and can have actions on $H$ and $M$, as above for complement slots. Most adjunct slots can be filled multiple times for a given headword.

Although the parser deals with the starter phrases left-to-right as they occur, there is still a question of the order in which modifiers are attached to a higher phrase. It is good to think of attachment as proceeding "middle-out" from the headword. From this point of view, the parsing algorithm takes care to attach the right modifiers of the headword before it attaches the left modifiers. We will discuss this more in Subsection 7.6 below.

We mentioned in Section 2 that there are fields in a phrase data structure besides the ones listed, which are not crucial for SSF to deal with explicitly. Two such fields are for *proclitics* (clitics on the left of their mother words) and *enclitics* (clitics on the right). We use *clitic* here only for the cases of words that are attached to their mother words, making a single word (with no space between). Clitics are not used for English, but are relevant for some other languages.

Each of these clitic fields in a phrase is a list of one-word phrases. They are installed as lists of starter phrases by morpholexical analysis, and are dealt with by the parsing algorithm if they are present (non-null lists). If clitics are not present, the parsing algorithm will combine only adjacent phrases $M$ and $H$ in the chart. But the parsing algorithm can also combine a phrase with one of its clitics (again by slot-filling). The parser non-deterministically removes a clitic $C$ from the corresponding clitic list of its mother phrase $P$ and tries to combine $C$ with $P$ by slot-filling. The SSF rules just see $C$ as if it were a normal full-fledged phrase. There are different possible treatments of the clitics, depending on the language, according as they are allowed to modify their mother phrases $P$ or instead act as higher phrases with $P$ as modifier of the clitic.

The parser also tries normal slot-filling combinations of adjacent phrases $L$ and $R$ in the chart, with $L$ on the left of $R$, even though there may be clitics still present in the clitic lists of $L$ or $R$. But there are constraints:

- Clitics that modify ($L$ or $R$) must already be attached (not still in a clitic list of $L$ or $R$).
- $L$ must have no enclitics in its enclitic list, and $R$ must have no proclitics in its proclitic list.
- If $L$ modifies $R$, then proclitics of $L$ that can act as higher phrases are moved from the proclitic list of $L$ to that of $R$.
- If $R$ modifies $L$, then enclitics of $R$ that can act as higher phrases are moved from the enclitic list of $R$ to that of $L$.

# 4 Parse scoring and parse space pruning

In the basic description of the SG chart parsing process in Section 3, we ignored the idea of *parse space pruning*. The SG parser maintains *parse scores* for each partial phrasal analysis, where the score represents roughly how likely the phrase is to be a correct analysis. The general idea is that when a phrase gets too bad a score compared with competing analyses, then that phrase is discarded from the chart. This makes for much greater efficiency in parsing (in both space and time). The parse scores are used also to rank the final analyses, which are listed best-first. Most applications use only the first parse.

The parse score for a phrase is a real number and is one of the fields of a phrase structure, as mentioned in Section 2. Let us call the score field of a phrase `eval`. The SG shell does its bookkeeping on `eval` in the following way as new phrases are built.

The most general, top-level ideas are these: Positive contributions to `eval` are best thought of as *penalties*. Penalties are given when a construction seems unlikely or complex. A higher score is a worse score. But the parsing algorithm or *X*`.gram` can assign negative scores to reward constructions as being preferred.

If a partial analysis phrase gets too high a score during parsing, compared with "similar" phrases (to be explained below), then it will be pruned away from the chart. When parsing is completed, the parse analyses are ordered according to parse scores, lowest (best) score first.

There are three main ways that the `eval` component of a phrase can change in value:

1. When an initial one-word phrase is formed, its `eval` number is normally 0, but the lexical entry may explicitly assign a score through lexical components of the form `(ev ..)`, `(sa ..)`.
2. The shell does some maintenance of scores based on general linguistic heuristics. Two of these are:
   (a) Complement slots are preferred over adjunct slots. This is accomplished by adding 1 to `eval` for each case of slot-filling by an adjunct. (However, the results may be overridden in *X*`.gram` for particular adjunct slots.)
   (b) There are scoring heuristics which mildly prefer close attachment of modifiers. "Other things being equal", close attachment is preferred.
3. Operations in *X*`.gram` can change `eval`. This is very common, and is an important part of writing grammar rules.

Now let us look in more detail at the process of parse space pruning.

We said above that in pruning, phrases are compared with "similar" phrases. Let us define this more precisely. Two phrases are called *similar (with respect to parse scoring)* if they have the same left and right boundaries, and their headwords have the same word number and the same "eval-features". The *eval-features* of a phrase are normally just the part of speech of the phrase, but the bottom line is that they are returned by a function `evalfeature` in the shell. For the current shell, this consists just of the part of speech, except that for verbs it includes also the features `vsubj` (the verb has an overt subject) and `vpass` (the verb is a passive past participle) if they are present.

So the process of pruning is this: The shell keeps track of the best (lowest) parse score in each similarity class. When a new phrase $H$ is formed, its score is compared (by the shell) with the best score in its similarity class. If $H$'s score is higher than the best, then $H$ is not added into the chart. If $H$'s score is the same as the best, then $H$ is simply added into the chart. If $H$'s score is less (better) than the (existing) best, then all existing phrases in $H$'s similarity class are deleted from the chart, and $H$ is added.

The description of parse space pruning above is somewhat simplified over the actual algorithm in the shell. In actuality, the parse score is split into a vector of three subscores, and this vector is manipulated by the shell. But the description above provides the main ideas of the algorithm. The rules in SSF

(so far) deal with only one of the three components in the vector, but it is by far the most important component for what the syntax rules can do.

Another ingredient in the actual algorithm is that when a new phrase $H$'s score is compared against the existing best score in its similarity class, there is a "fuzz factor" in deciding whether to prune away something. There is a system variable `prunedelta`, which by default is 0. (And it is 0 for the English grammar.) In deciding to discard $H$, its score actually has to be higher (worse) than `prunedelta` plus the score of the best in its similarity class. Similarly for the case of discarding phrases whose scores are worse than that of $H$.

Parse pruning can be turned off totally by turning off the flag `prune`. When `prune` is off, the only effect of parse scoring is to order the final parses. All possible parses for a segment are obtained. This takes much more space and time, and often the parser will crash for longer sentences (unless *X***SG** is initialized with a lot of working memory). However, in debugging the grammar, it is often useful to do `-prune` on smaller sentences, or on reduced versions of larger sentences. This can be worthwhile when an expected parse is not being obtained, and you suspect that it is being lost due to pruning.

# 5 Features and the SFF formalism

Features for a language $X$ can be specified in a file named *X*`feas.lx`, written in the formalism SFF. The purpose of *X*`feas.lx` is twofold:

1. To declare features so that they get an efficient internal representation.
2. To specify relationships among these features.

Such declared features get two kinds of internal representations – as "atoms", represented as integers, and as bit positions in bit strings. Feature sets on parse nodes are represented internally as bit strings of a fixed length. (Lengths used currently in existing SGs are 64 for English, 96 for the other European languages, and 512 for Arabic.) In the feature bit string for a parse node, the bit positions for features that are present are turned on (assigned 1 instead of 0). Internally each individual feature $f$ gets an associated bit string with exactly one bit on via an array that uses $f$ as an index.

This internal representation of features from *X*`feas.lx` is invisible to the grammar writer. Instead, there are various operators in SSF that deal with features in perspicuous way.

The features specified in *X*`feas.lx` could be of any kind – morphosyntactic or semantic – but in practice they should be mainly be morphosyntactic, because of limited space available in the internal representations just described. SG also allows an open-ended set of features (with less efficient representation) – see [9] and [10]. Most semantic features should be represented in this way.

Entries in *X*`feas.lx` always start in column 1 with a string representing a feature – the *index* of the entry. More ingredients may follow, but if there are additional lines for the entry, the lines must begin with at least one whitespace character. When SG reads in *X*`feas.lx`, the index feature of each entry will be given its internal representation (an atom assignment and associated bit code), unless it has already been seen in a prior entry.

The simplest form of an entry is just an index feature by itself. For instance the following entries could declare four case features:

```
nom
gen
dat
acc
```

If there is more to an entry than the index feature $f$, then $f$ should be followed by a `<` sign, perhaps with surrounding whitespace. More features may follow after the `<`, separated by whitespace, and then each such feature $g$ is taken to be *implied* by $f$ (or higher in the feature hierarchy than $f$). This means that if $f$ is present on a parse node, then an SSF test for presence of $g$ will succeed. For example the entry

```
month < time advnoun
```

would say that the `month` feature implies the features `time` and `advnoun`. Feature implication is handled in internal representation (invisibly to the grammar writer) by bit string arrays.

Finally, an SFF entry may specify a *superfeature* relationship of the index feature $f$ to a set of other features $\{a_1, \ldots, a_n\}$. This happens when the entry is of the form

$f$ `<` $h_1, \ldots, h_m$ `>` $a_1, \ldots, a_n$

Here $m$ could be 0 – there are no implied features listed. Such a relationship means basically that $f$ is an abbreviation for the set of features $\{a_1, \ldots, a_n\}$, which are normally viewed disjunctively. For instance an SSF test

`(supf` $f$`)`

will succeed iff at least one of the features in $\{a_1, \ldots, a_n\}$ is present on the node in focus. And an SSF action

`(removef` $f$`)`

will remove all the features $a_1, \ldots, a_n$ from the node in focus.

Let us give an example for the German SG (Claudia Gdaniec) – for dealing with NP features for case, number and gender. We first include in `defeas.lx` the basic features for case, number and gender.

```
nom
gen
dat
acc
sg
pl
f
m
nt
```

A complexity of German NP feature markings is that words are often ambiguous with respect to these features. For instance, "Studenten" is masculine, but could have any combination of case and number except nominative singular. To handle this, the GSG morphology marks on "Studenten" the following features:

```
mgen mdat macc plnom plgen pldat placc
```

Here a compound feature like `mgen` means the conjunction of `m` and `gen`. The seven features marked are viewed as holding disjunctively, or ambiguously, on the node. In actuality, the GSG NP features include a distinction between weak and strong features, but we will ignore that in this discussion.

The total set of compound NP features (ignoring the weak/strong distinction) is:

```
fnom fgen fdat facc
mnom mgen mdat macc
ntnom ntgen ntdat ntacc
plnom plgen pldat placc
```

The ones on the first three lines have `sg` number, and the ones on the last line have `pl` number, where there is no distinction for gender. So in `defeas.lx` we enter these 16 compound features, one feature per line, in order to declare them. Although they are compound in intent, the SG system sees them as atoms.

Now we can enter superfeature rules in `defeas.lx`. Each of the simple NP features $f$ for case, number or gender will be a superfeature for the set of compound features that imply $f$. So we do this as follows:

```
nom < > fnom mnom ntnom plnom
gen < > fgen mgen ntgen plgen
dat < > fdat mdat ntdat pldat
acc < > facc macc ntacc placc
sg < > fnom fgen fdat facc mnom mgen mdat macc
       ntnom ntgen ntdat ntacc
pl < > plnom plgen pldat placc
f < > fnom fgen fdat facc plnom plgen pldat placc
m < > mnom mgen mdat macc plnom plgen pldat placc
nt < > ntnom ntgen ntdat ntacc plnom plgen pldat placc
```

In Section 7.3 we will describe an SSF operator `agreef` which can test feature agreement between a modifier phrase $M$ and a higher phrase $H$ along some "dimension" of features, such as case, number or gender. An example could be agreement between a determiner and a head noun for German. For this, the sequence of agreement tests could look like this:

```
(agreef nom gen dat acc)
(agreef sg pl)
(agreef f m nt)
```

We will describe the details in Section 7.3, but what happens basically is that `agreef` uses the superfeature expansions of its argument features to check agreement. So, for example, for `nom` it looks at the features `fnom`, `mnom`, `ntnom`, `plnom`, as marked on $M$ and $H$, to see if they both can be considered nominative (among other cases). The first `agreef` test will succeed if $M$ and $H$ have at least one case feature in common, in this sense.

To get *X*`feas.lx` used by *X***SG**, one should compile it, using the command:

```
Xsg -compilex Xfeas.lx -enc 1 -nosort
```

# 6 Overall form of SSF and *X*.gram

The syntax rules for a language $X$ should be put in the file $X$`.gram`, and should use the formalism SSF. We describe the overall form of $X$`.gram` in this section.

The rules in $X$`.gram` are slot (filler) rules, and these are of three types:

1. Complement slot rules
2. Adjunct slot rules
3. Special slot rules

Each rule is of the form:

```
Head < Body
```

For a complement slot rule, the `Head` is is the name of the complement slot (e.g. `subj`). For an adjunct slot rule, the `Head` is a POS – the POS of of the filler phrase $M$ (e.g. `noun`). For complement and adjunct slot rules, the same `Head` can occur in more than one rule, as described above in Section 3.

There are two kinds of special slot rules, and $X$`.gram` should have at most one occurrence of each. One uses the special slot name `arb` as index, and (if present) it is applied (and must succeed) every time there is a slot-filling. It is applied after the normal slot rule is applied. The second special slot rule uses

the slot name `top` as index, and it is applied (after the other slot rules) for a final top-level phrase analysis.

The `Body` components for the three types of slot rules all use the same syntax, which is described in Section 7.

The three types of slot rules can come in any order in *X*`.gram`, and can be intermingled. (This is possible because the SSF interpreter can recognize the type of the rule by the type of its `Head`.)

The overall basic rule format is "free-form" (whitespace can go anywhere except in the middle of tokens), except that each rule has to start (with its `Head`) at the beginning of a line, and continuations of rules must *not* start at the beginning of a line.

Comments with `/*` and `*/` are allowed, but with restrictions. The initial `/*` string must occur first on the line, or after initial blank characters. And then the comment extends only to the end of the line. Actually the closing `*/` is not required, but it is good practice to use it.

The overall syntax of the `Body` of any rule is Cambridge Polish, or Lisp-like, with one exception about the top level, which we will describe below. We will call expressions in Cambridge Polish *terms*. The general form of a term can be described quite simply. A *term* is either an *atomic term* or a *list term*. *Atomic terms* are sequences of characters not including whitespace or (round) parentheses, except that a backslash \ can be used as an escape character to allow those symbols also. A *list term* consists of a left parenthesis followed by possible whitespace, followed by any sequence of terms separated by whitespace, followed by possible whitespace, followed by a right parenthesis. (In SSF we do not use the dot notation sometimes used in Cambridge Polish.) The empty list can be denoted by either `()` or `nil`.

So an example of a term is:

```
(if (opt bfin) (mf vsubj))
```

Typically in an SSF list term

```
(a b c d ...)
```

the initial atomic term `a` is either a primitive operator, like `mf` in the preceding example, or a logical operator, like `if`. The subsequent members of the list term are operands for the operator, and quite a few of the SSF operators can take any number of operands.

The SG shell actually keeps track of runtime subdatatypes for atomic terms, distinguishing for example between string terms, integer terms, and real (double) terms. The SSF user does not need to be too aware of these distinctions, but can just use reasonable atomic terms as expected by the operators.

There are two special types of atomic terms though that the user should be aware of. These have to do with variables. Operators can set and evaluate variables (whose values are terms), and the SSF interpreter keeps track of variable

binding contexts – local to each rule application. Variables are recognized by their special syntax. Variables are of the following two kinds, with the indicated syntax:

| | |
|---|---|
| *Simple variable:* | `v1, v2, v3, ...` |
| *Put variable:* | `>v1, >v2, >v3, ...` |

So a variable begins with either `v` (specifically that letter) or `>v`, and is followed by a sequence of digits. Put variables are always assigned a value by an operator, and they can be reassigned. Simple variables are only evaluated. When a put variable `>v`*i* is assigned a value, then the corresponding simple variable `v`*i* gets that value. If a simple variable is evaluated but was not previously assigned a value, then its value will be taken as `nil`.

Several of the SSF operators do *(pattern) matching* with their arguments. If two terms are matched and they contain no variables, then the condition for success is just that the two terms (can) look alike in external syntax. One can give an obvious recursive definition of this, based on the recursive definition of terms. If one or both of the two terms being matched contain variables, then the idea is as follows:

- If a *simple* variable $v$ is to be matched against a subterm $y$, then $v$ is evaluated, and its value $x$ must match $y$. (The value $x$ could itself contain other variables, and so evaluation could be recursive, with a potential for infinite loops. But normally one can easily avoid such usages of variables.)
- If a *put* variable $v$ is to be matched against a subterm $y$, then $v$ is just assigned the value $y$, and this match always succeeds.

The simplest and most general operator that causes pattern matching is `=`. This takes two arguments, which must match for the operation to succeed. Note that because of the rules just listed for matching of variables, `=` can be used both for assigning and for evaluating variables. For instance,

```
(= >v5 (John sees Mary))
```

will assign the list `(John sees Mary)` to the variable `v5`.

We stated above that there is an exception about using the term notation on the top level of a rule body. It is that the body can be a sequence of one or more terms:

$b_1\ b_2 \cdots b_n$

But when $X$`.gram` is read in by the shell, this is in effect converted to the single term:

`(&` $b_1\ b_2 \cdots b_n$`)`

Here `&` is the conjunction operator of SSF, to be discussed in Section 7 below. As an example, the following is a very simplified (and inadequate) complement slot rule for the subject slot for English:

```
subj < le (agree sg pl)
```

The body here will be seen by the SSF interpreter as the term

```
(& le (agree sg pl))
```

The first condition tests that the (implicit) modifier phrase $M$ (the subject) is on the left of the (implicit) higher phrase $H$. The second condition tests that $M$ and $H$ agree with respect to the features `sg` and `pl` (singular and plural).

The same convention about leaving off the outer parentheses on the top level is used in the SG lexical formalism SLF.

Valid terms spanning several lines within the `Body` of a rule may be commented out by surrounding them with:

```
(# ... )
```

# 7 Slot rules

In this section we cover the form of slot rules.

## 7.1 General form of the rules

As indicated above, the three types of rules have the same syntax and allowable operators, and differ only in the following two ways:

1. The `Head` of a complement slot rule is a complement slot (name); the `Head` of an adjunct slot rule is a part of speech; and the `Head` of a special rule is one of the symbols `arb` or `top`.
2. The slot and its option are *given* for a complement slot rule, but they are *assigned* by an adjunct slot rule.

Currently the allowed parts of speech are as follows. We spell out the names or give an explanation in the cases where the meaning may not be obvious. See [9] for more details.

- `verb`.
- `noun`.
- `adj`. Adjective.
- `adv`. Adverb.
- `qual`. Qualifier.

- `det`. Determiner.
- `prep`. Preposition.
- `conj`. Coordinating conjunction.
- `subconj`. Subordinating conjunction.
- `thatconj`. Special subordinating conjunction like "that".
- `infto`. Preinfinitive like "to".
- `subinf`. Variants of infto.
- `forto`. The English "for" in for-to constructions.
- `incomplete`. The formal POS for an incomplete parse.

The allowed complement slot names are given as follows. In each item we first list a POS and then the complement slots that it may have. See [9] for conventional meanings (illustrated mainly for English).

- `verb: subj, obj, iobj, comp, auxcomp, pred`
- `noun: nobj, nid`
- `adj: aobj`
- `adv: avobj`
- `prep: pobj, objprep`
- `conj: pconj, lconj, rconj, postconj`
- `subconj: sccomp`
- `thatconj: thatcomp`
- `infto: tocomp`
- `subinf: subinfcomp`
- `forto: forsubj, forcomp`

The grammar writer can create new slot option names, but the following are commonly used ones:

```
a, agent, aj, av, bfin, binf, dt, en, ena,
fin, fina, finq, finv, ft, ger, gn, inf, ing, io, it,
itinf, itthatc, itwh, lo, n, na, nen, nmeas, nop, nummeas,
p, padj, pinf, pinfd, prflx, prop, pt, pthatc, pwh,
qt, rflx, sc, so, thatc, v, wh
```

The grammar writer can also choose adjunct slot names, but the following ones are commonly used names. Again, each item has a POS followed by adjunct slots for that POS.

- `verb: vpreinf, vadv, vprep, vsubconj, vnfvp, vfvp, vforto, vcomment, vcompar, vsothat, vadjp, vnp, vinfp, vvoc, vdat, vacc, vadj, vdet, vnoun, vrel, vextra, vnthatc, proclitic, enclitic, whadv, whprep`
- `noun: nadv, ndet, nadj, nnoun, nposs, nadjp, nper, nprop, nappos, nprep, ngen, nrel, nrela, nnfvp, nsubconj, ncompar`
- `adj: adjpre, anoun, adjpost, asothat, acompar, aprep, arel`
- `adv: advpre, advpost, advinf, avsothat, avcompar`
- `det: dadv`
- `prep: padv, pvapp`
- `subconj: scadv`
- `infto: toadv`

When a slot rule is applied, several arguments are given implicitly to the rule. In effect, this means that the SSF interpreter uses these arguments in functions that recursively interpret the rules. The arguments (or implicit data) for the rules are as follows. Of course the SSF implementation in the shell has its own data structures for the arguments, but these details are hidden from the external view of SSF, so as to make it more abstract and implementable in different ways. Ultimately the rule writer can use SSF operators to access what is necessary in the data structures.

- The modifier phrase $M$.
- The higher phrase $H$.
- The slot $S$ of $H$ that $M$ fills. This is *given* for a complement slot rule, and is *assigned* by an adjunct slot rule.
- The option of $S$ used in the slot-filling. Again, this is given for a complement slot rule, and is assigned by an adjunct slot rile.
- The set of features of $M$. This can be tested or modified by the slot rule. Of course such a modification does not change the features of $M$ in the occurrence of $M$ as a chart element, but it changes the features of $M$ as $M$ is viewed in the new version of $H$ with $M$ as a subnode (if the slot-filling succeeds).
- The set of features of $H$. This can be tested or modified by the slot rule. A modification occurs in the new version of $H$ with $M$ as a subnode.
- A boolean that indicates whether $M$ is on the left or the right of $H$.
- An identifier that specifies any *separator* punctuation between $M$ and $H$. More on this below.
- An assignable list of slots that are *raised* by the slot filling. They are added to the available slots in the new $H$. Primitive operators of SSF can add to this list. More on this below.

- An assignable list of all the available slots of the new $H$. This item is rarely used. More below.
- The current *evaluation* or *parse score*, a real number, for the new $H$. This number can be tested of modified. The main SSF operator `eval` dealing with it increments it by the amount of the argument of `eval`.
- The *node in focus*. At the top level of interpretation of the `Body` of the rule, the node in focus starts out as $M$ (the modifier). But SSF operators can shift the node in focus, allowing this node to wander to any node in $M$ or $H$. Many operators of SSF use this node as an implicit argument. The device greatly simplifies the use of SSF and creates more readable and more compact expressions.

Now let us proceed to the specification of the `Body` of a slot rule. The `Body` is an *SSF test*, and we define this recursively. Such a test should be thought of on one level like a proposition – a term that has a truth value. For the slot-filling to succeed, the top-level SSF test (the `Body` of the slot rule) must be true. The slot rule interpreter evaluates SSF tests recursively, and truth values of embedded SSF tests are of course crucial to determining truth values of higher tests. But evaluation of SSF tests can also have side effects, like setting the implicit arguments of the rule described above. Some SSF tests always succeed and are of use only for their side effects.

An *SSF test* is a term that is either a *basic (SSF) test* or a *compound (SSF) test*. A *basic test* is an atomic term which is among those listed in the following subsections (with explanations of their truth conditions). A *compound test* is a list term where the first element, the *operator* for the test, is one of the operators listed in the subsections below, and the succeeding members of the list – the *arguments* of the operator – are SSF tests.

Now we go through the basic tests and operators. These are grouped in the following subsections, each of which deals with related tests. In this material, we always use $M$ to mean the (implicit) modifier phrase for the rule, and $H$ to mean the (implicit) higher phrase for the rule.

## 7.2 Logical operators

The logical operators are as follows. Each one except `=` can take any number of arguments, and these arguments should be SSF tests. For each operator we describe the conditions under which a test based on that operator will succeed.

- `&` (and). All argument tests must succeed. Evaluation of the arguments stops, with failure, if any fails.
- `|` (or). At least one argument test must succeed. Evaluation of the arguments stops, with success, if any succeeds.
- `^` (not). All argument tests must fail. Evaluation of the arguments stops, with failure, if any succeeds.

- `^&` (nand). It is not the case that all the argument tests succeed. Evaluation of the arguments stops, with success, if any fails.
- `if` (conditional). This behaves like an if-then-else series. Suppose first that there are an even number of arguments. The interpreter evaluates in succession the odd-numbered arguments. If one of them succeeds, then the interpreter evaluates the succeeding even-numbered argument, and returns the truth value of that. If no odd-numbered argument is true, so that the evaluation "runs off the end", then the `if` test succeeds. If there are totally an odd number of arguments, then evaluation behaves as if a `T` test were inserted before the last argument. Thus in this case the last argument is like an `else`.

  The `if` test

  ```
  (if t (& a b c)
      u d
      v (& e f)
  )
  ```

  would then be like the following boolean expression in C:

  ```
  t ? (a && b && c) :
  u ? (d) :
  v ? (e && f) :
  1
  ```

  And the `if` test

  ```
  (if t (& a b c)
      u d
      v (& e f)
      (& w z)
  )
  ```

  would be like

  ```
  t ? (a && b && c) :
  u ? (d) :
  v ? (e && f) :
  (w && z)
  ```

- `=` (equality). This takes two arguments, and succeeds iff they match – according to the rules for pattern matching described in Section 6.

There are two basic SSF tests `T` and `F` which act like `true` and `false` respectively; `T` always succeeds and `F` always fails.

There is also a logical operator `try`, allowing any number of arguments, where

`(try` *a b c* `...)`

is equivalent to:

`(| (&` *a b c* `...) T)`

This of course always succeeds, but it can be useful for evaluating its arguments (until one fails) because the arguments may have side effects.

## 7.3 Testing and changing features

The operators described in this section take features as arguments. Any number of arguments are allowed, unless it is stated otherwise. Features can be parts of speech, morphosyntactic features, or semantic features. In the implementation, these different kinds of features could be represented in different ways. For instance in the current C implementation of SG, morphosyntactic features are implemented by bit positions in bit strings, and operations on them are done with machine logical operations on bit strings. But conceptually in SSF, one can view all of the features as being specified by identifier strings.

The operators are as follows. For several of the items, we list three operators, testing respectively for the node in focus, for $M$, and for $H$.

- `f`, `mf`, `hf`. This tests that each of its arguments is a feature of the phrase (where we include morphosyntactic features, parts of speech, and semantic features). The first operator name is another overloaded symbol! When it occurs as a feature, it conventionally means feminine gender.
- `of`, `omf`, `ohf`. This tests that at least one of its arguments is a feature of the phrase.
- `nf`, `nmf`, `nhf`. This tests that none of its arguments is a feature of the phrase.
- `supf`, `supmf`, `suphf`. This is like `f`, but its arguments can be superfeatures, and their expansions through superfeature rules (see Section 5) get checked for.
- `osupf`, `osupmf`, `osuphf`. This is like `of`, but for superfeatures.
- `st`. This tests that its argument is a semantic type of the phrase in focus.
- `pos`, `mpos`, `hpos`. This allows any number of arguments, which are matched against the POS of the phrase. The testing is disjunctive. Evaluation stops, with success, when an argument matches.
- `setmpos`, `sethpos`. This takes one argument, which is either a simple variable or a constant, with value a POS, and sets the POS of $M$ or $H$, respectively, to that POS.

- `addf`, `addmf`, `addhf`. This adds each of its arguments as a feature of the phrase. (The first operator is not implemented yet, but the other two are.)

- `delf`, `delmf`, `delhf`. This deletes each of its arguments as a feature of the phrase. (The first operator is not implemented yet, but the other two are.)

- `removef`, `removemf`, `removehf`. This is similar to `delf`, but its arguments can be superfeatures, and their expansions through superfeature rules (see Section 5) get removed.

- `agree`. This succeeds iff at least one argument feature is a feature of both $M$ and $H$. If it does succeed, then every argument feature that is not a feature of both $M$ and $H$ is removed from the features of $M$ and $H$. Thus for example `(agree sg pl)` would test that $M$ and $H$ agree in number.

- `agreef`. This is similar to `agree`, but allows the arguments to be superfeatures, and it uses both the actual superfeatures and their expansions through superfeature rules (see Section 5). The arguments can also be pairs of features, where the first member of the pair is checked for $M$ and the second member is checked for $H$.

- `raisef`. This actually appears (currently) like a basic test, with no arguments. It simply replaces the features from $H$ with a copy of those of $M$. It is useful in handling coordination. The right conjunct (slot `rconj`) of a coordinating conjunction should be filled before the left conjunct (slot `lconj`). In the process of filling `rconj`, it would be normal to call `raisef`, so that the larger phrase inherits the features of the right conjunct.

- `raisesf`. This is also a basic test. It adds to the semantic features of $H$ those of the phrase in focus.

- `coordf`. This is also (currently) used only as a basic test, with no arguments. Its purpose is to coordinate the features of $M$ and $H$. It should be called when the left conjunct (slot `lconj`) of a coordinating conjunction $C$ is being filled. So $H$ would have head $C$, would have `rconj` filled by the right conjunct $R$, and would have inherited the features of $R$ via `raisef`. So `coordf` is really coordinating the features of $M$ (the left conjunct) with those of $R$. Currently, `coordf` requires that $M$ and $H$ have the same POS, and it gives $H$ each of its existing features that are also features of $M$. It also requires (to succeed) that if the POS is `verb`, then $M$ and $H$ have at least one of the features `vfin` (finite verb), `vinf` (infinitive), `ven` (past participle) or `ving` (present participle) in common. So these requirements are now in the shell. A facility for expressing and specifying the details of `coordf` should be put in the SSF formalism instead of being in the shell. This will be done at a later point.

## 7.4 Testing headwords

The operators that test the head (multi-)word of a phrase can test the textual form of the headword (with whatever mixed case is in the text), the lower case form of the headword, the citation form (lemma), or the sense name. Each of the operators can take any number of arguments, and succeeds iff the corresponding headword-related field *matches* at least one of the arguments. The operators are as follows. For the first four items we list three operators, testing the head (multi-)word of respectively the node in focus, for $M$, and for $H$. The most commonly used operators are `mcite` and `hcite`.

- `cite`, `mcite`, `hcite`. This tests that the citation form of the head matches one of the arguments.
- `word`, `mword`, `hword`. This tests that the textual form of the head (with possible mixed case) matches one of the arguments.
- `lcword`, `mlcword`, `hlcword`. This tests that the lowercase form of the head matches one of the arguments.
- `sense`, `msense`, `hsense`. This tests that the sense name of the head matches one of the arguments.
- `mwcite`. This is a basic test, which checks that the citation form of the phrase in focus is a multiword.

Note that the uses of pattern matching can both test for specific strings and set variables to strings. Thus in the following, the first test would test that the citation form of $H$ equals one of the eight words listed, but the second test would always succeed and would set the variable `v1` to the citation form of $H$, whatever it is.

```
(hcite be do have will can may shall must)
(hcite >v1)
```

## 7.5 Testing and manipulating slots and options

The most basic operators involving slots and options are the following.

- `slot`. This can have one or two arguments. It matches its first argument against the *current slot*. The default current slot is the slot being filled by $M$ in $H$. For a complement slot rule, this slot is chosen by the shell. For adjunct slot rules, it is set by the rule itself, using `setslot`. However, operators in the `mod` family (see Section 7.9) create an environment with a different current slot – the slot filled by the modifier. If there is a second argument, it is matched against the current slot option.
- `opt`. This is similar, but deals only with the current option.

- `setslot`. This sets the slot (name) to the argument. This would normally only be used for adjunct slot rules, but can also be used for complement slot rules.
- `setopt`. Similar, for options.
- `iscomp`. This tests that its argument is a complement slot name.
- `ob`. When this occurs as a basic test, it says that the current (complement) slot to be filled is obligatory.
- `eslot`, `emslot`, `ehslot`. This tests for "empty slots" – or the current available complement slots – in the phrase. It can have from 0 to 4 arguments. With no arguments, it tests that there are some available slots. The allowable arguments are as follows, and they can come in any order.

  1. A non-list argument (atomic or a variable) matches against the name of the slot you're looking for.
  2. An argument of the form `(opt OptionName)` matches `OptionName` against the option name of an available slot (which must be the slot in (1) if given).
  3. An argument of the form `(ob N)` matches `N` against 1 or 0 according as the slot is obligatory or not (and this must be for the slot and/or option in (1) or (2) if given).
  4. An argument of the form `(cite [C])` matches `C`, if given, against a required filler citation form (and this must be for the slot and/or option in (1) or (2) if given). If an argument `C` is not given, it means that there are no required citation forms.

  For example,

  ```
  (ehslot obj (opt n) (ob 1))
  ```

  would test for an available obligatory `obj` slot having `n` among its options.

There are several operators that concern what happens to the available slots (unfilled complement slots) of $M$ when $M$ becomes a modifier of $H$. For instance, obligatory unfilled slots of $M$ should generally not be ignored (though they can be in some cases). A phrase is called *satisfied* if it has no obligatory available slots. One thing that can be done with available slots of $M$ (whether or not they are obligatory) is to *raise* them to become available slots of $H$. They might get filled on the level of $H$, or they might get raised still higher when $H$ itself becomes a filler. Filling of raised slots happens with both extraposition and coordination, as in:

*Who did you say she tried to find?*
*He looked for and then read the book.*

The following basic tests and operators deal with satisfied phrases and raising of slots from $M$ to $H$. Normally only one of them would be used at a given spot in the grammar. Currently the definitions of these are all in the shell. Probably later SSF will be expanded to allow more language-specific versions of them to be written in $X$`.gram`.

- `satisfied`. This is a basic test (no parentheses, no arguments). The main idea is that the node in focus has no obligatory available (unfilled) slots. For English, it fails also if the node in focus has an extraposed `subj` slot. (For cases like *Who did you say he thought was there?* the rules should not call `satisfied` at the spot where the `subj` is raised.)

- `raiseslots`. Again a basic test. This copies typical raisable slots like `obj` and `objprep` from the available slots of $M$ to those of $H$ and checks that the non-raised slots are optional (non-obligatory). Thus `satisfied` need not be called.

- `satorfill`. This is like `satisfied`, but it can exempt one obligatory object-type slot.

- `satisfill`. This can be used either as a basic test or an operator (with arguments). It is like `satisfied`, but it exempts an object-type slot $S$ in a passivized past participle phrase filler where one considers that $S$ is "filled" implicitly by the passivization. It can be used an operator also with any number of slot arguments, and it tests that the exempted slot $S$ (if any) is not among those arguments.

- `satfillraise`. The idea of the "`fill`" part of the name is that a passivized object-type of slot is "filled" implicitly by the passivization. The idea of the "`raise`" part is that other raisable slots of $M$ are raised to $H$. The idea of the "`sat`" part is that otherwise the unfilled slots of $M$ are not obligatory.

- `raised`. This is a basic test meaning that the slot of the current slot-filling has been raised.

In addition to these operators and tests dealing with satisfied phrases and slot raising, there is a basic test that deals with slots in coordination:

- `coordslots`. This is a basic test which, like `coordf`, should be called when $M$ is a left conjunct of a coordination and the `lconj` slot is being filled. It matches up the slots of $M$ with those of the right conjunct in $H$ (which must be present), factoring out basically the ones in common, and adding these to the available slots of $H$.

## 7.6 Sides and order of slot-filling

The basic test `le` succeeds iff $M$ is on the left of $H$. The basic test `ri` succeeds iff $M$ is on the right of $H$.

Recall from Section 3 that the parsing algorithm takes care that right modifiers are attached before left modifiers.

In controlling the order of modifiers *among* the left modifiers, it is common to use in $X$`.gram` the features:

```
le1, le2, le3, le4, xtra
```

These represent increasing degrees to the left, away from the headword, and `xtra` can be used when there is an extraposed left modifier. The feature lexicon $X$`feas.lx` should show the hierarchy among the `le`$i$ features like so.

```
le2 < le1
le3 < le1 le2
le4 < le1 le2 le3
```

The shell takes care of adding the feature `le1` for any left modification. But $X$`.gram` should take care of adding other `le`$i$ features as desired. For instance for the `ndet` slot (for determiners), `en.gram` does:

```
(addhf le3)
```

Similarly, for right modifiers, it is common to use the features:

```
ri1, ri2, ri3
```

And the shell takes care of adding the feature `ri1` for any right modification.

## 7.7 Segment positions

The operators of this section test for segment positions – for instance left and right boundaries of phrases – expressed as interstice numbers, as described above in Section 2.

- `lb`, `mlb`, `hlb`. This takes a single argument which is matched to the left boundary of the phrase (as an integer term). Thus for example `(mlb >v1)` would assign to `v1` the left boundary of $M$.
- `rb`, `mrb`, `hrb`. Similar, for right boundaries.
- `wordno`. This matches its argument to the word number of the (headword of) the phrase in focus.
- `wlb`. This matches its argument to the left boundary of the headword of the phrase in focus.

- `wrb`. Similar, for the right boundary. Note: The difference between these two boundary numbers could be greater than 1 because the headword could be a multiword.
- `len`. This matches its argument to the length of the phrase in focus (right boundary minus left boundary).
- `segrb`. This matches its argument against the right boundary number of the whole current segment. (The left boundary is always 0.)

## 7.8 Punctuation

Certain punctuation symbols, or tags acting as such, existing textually between $M$ and $H$ may be taken by the shell as a *separator* for the slot-filling. These include the following:

```
,   ;   -   ...   --   ---   /   \
dash   emdash   mdash   endash   ndash
slash   slr   bslash   bsl
```

For the backslash, one should write "\\" (without the quotes), because \ is an escape character in the syntax of SSF. There is also a separator consisting of a hyphen with a blank on either side, and this should be written as "`\ -\ `".

The operator `sep` tests that at least one of its arguments matches the existing separator. Example:

```
(sep , \ -\ )
```

This tests that the separator is either a comma or a hyphen with spaces around it.

The symbol `sep` can also be used as a basic test, in which case it means that there is some separator between $M$ and $H$.

For convenience, there is a negative version `nsep` of `sep`. As a basic test, this means there is no separator between $M$ and $H$. As an operator, it means that none of its arguments is the separator.

The separators listed above can also serve as coordinating conjunctions (except that there are constraints on a hyphen when it is not preceded by a blank). The parsing algorithm takes care of "promoting" these symbols to be conjunctions, and giving them `lconj` and `rconj` slots. It also does this for the left bracketing symbols

```
(   [   {   lpar   lbrk   lbracket   lbrc
```

The matching right brackets are required by the parser shell to be on the right side of the `rconj` filler.

When a punctuation token is promoted in this way to be a regular phrase node (as a coordinator), it gets a word number that is equal to its interstice number plus a certain SG system constant `sentlenmax`, which is an upper bound on the number of word tokens in a segmemt that the system will try to parse. The constant `sentlenmax` is set by default to 100. Example:

```
The goat, the cow and the horse.
```

Here **ESG** will take the comma, at interstice 2, as a coordinator (and as the top node of the parse tree). Its node number will be 102.

There are several operators for testing the existing punctuation at other positions in the segment.

- `punc`. The first argument should be an interstice number, given as a constant of a simple variable. The remaining arguments should be for punctuation symbols, given as constants, simple variables, or put variables. The test is that at least one of those remaining arguments matches a punctuation symbol at the given interstice, and evaluation stops when there is a match. If there is no match (in particular if no punctuation arguments are given), then the test fails. For example,

  ```
  (punc 3 , -)
  ```

  would test that there is a comma or a hyphen at position 3.

- `spunc`. This is similar to `punc`, but it tests a fuller version of the punctuation symbols. Each token has a field `spword` that includes any space characters before it in the segment, and this operator tests `spword` fields.

- `nopunc`. This takes one argument, which should be an interstice number, and it checks that there are no punctuations there.

- `lbpunc`, `mlbpunc`, `hlbpunc`. This is like `punc`, but it does not take the initial interstice argument. Instead, it checks its arguments against the left boundary of the phrase. For example, `(mlbpunc , -)` checks that there is a comma or a hyphen at the left boundary of $M$.

- `rbpunc`, `mrbpunc`, `hrbpunc`. Similar, but for the right boundary.

- `lbrbpunc`. The first two arguments should be constants or simple variables whose values are interstice numbers $m$ and $n$ respectively, with $m < n$. The remaining arguments should represent punctuation symbols – as constants, simple variables, or put variables. The test checks whether one of those punctuations occurs at some interstice $i$ strictly between $m$ and $n$, i.e. $m < i < n$. This would be equivalent to calling `punc` at every such $i$ and with the same punctuation arguments – succeeding and stopping when a match is found, otherwise failing. Thus e.g. `(lbrbpunc 2 6 ,)` would check that there is a comma punctuation somewhere strictly between positions 2 and 6.

- `phpunc`. This is similar to `lbrbpunc`, but where the first two arguments are replaced by the left and right boundaries of the phrase in focus. Thus e.g. `(phpunc ,)` would check that there is a comma somewhere in the interior of the phrase in focus.

- `phspunc`. This is similar to `phpunc`, but it checks the interior of the phrase for the `spword` forms of punctuation tokens.

- `phnopunc`. This is a basic test that checks that the interior of the phrase in focus is free of all punctuation.

- `quoted`. This is a basic test that checks that the phrase in focus is quoted by any of several types of matching quotes.

- `segend`. This takes one argument, which is matched against the segment terminator punctuation, if such is present. It fails if none is present. For example, if the sentence is

  John asked, "Where are you going?"

  then `(segend >v1)` would bind `v1` to a question mark (term).

## 7.9 Exploring parse trees

Exploration of parse trees is accomplished by using *focus operators*, which can set the node in focus. Many SSF operators have an implicit node in focus, so that they are then dealing with that phrase. A focus operator can have any number of arguments, and these should be SSF tests. The new node in focus is local to these arguments. Of course further (local) shifts of focus may occur within an argument whose operator is itself a focus operator. Recall that for the top level test of a slot rule (its `Body`), the node in focus is $M$ (the modifier phrase for the rule). The focus operators are as follows:

- A simple variable `v`$i$. The variable should be bound to a phrase (node) $P$, and then the node in focus becomes $P$ for the arguments of this operator. One can *bind* a variable `v`$i$ to a phrase $P$, when this variable appears in put form `>v`$i$ as a basic test argument in another test. Such binding of variables is more than local to the current test. It persists for the remainder of the SSF interpretation of the rule, unless it is rebound.

- `h`. This sets the focus to $H$ (the higher phrase for the slot-filling). The symbol `h` is overloaded, because when it occurs as a feature, it conventionally means the feature "human".

- `m`. This sets the focus to $M$. This symbol is also overloaded, because when it occurs as a feature, it conventionally means the feature "male".

- `lmod`. The interpreter goes through the left modifiers of the current node in focus in left-to-right sentence order, and for each such modifier $x$ tests whether all the arguments of `lmod` are true with $x$ as node in focus. When the arguments of `lmod` are being tested, the operator `slot` will match its argument against the slot being filled by the modifier, and the operator `opt` will match its argument against the slot option for the modifier. The `slot` operator can also be given two arguments, where the first is the slot and the second is the option. If a `slot` test is used, a slight efficiency is gained by making it the first argument of `lmod`. If all the arguments of `lmod` are true for some modifier $x$, then the `lmod` test succeeds; otherwise it fails.

- `rmod`. This is similar to `lmod`, but using right modifiers. And the right modifiers are visited in right-to-left sentence order.

- `mod`. The test

  (`mod` *args*)

  is equivalent to

  (`|` (`rmod` *args*) (`lmod` *args*))

- `hlmod`. This takes $H$ as the node in focus and applies `lmod`. So

  (`hlmod` *a b c ...*)

  is equivalent to:

  (`h` (`lmod` *a b c ...*))

- `hrmod`. Similarly, this takes $H$ as the node in focus and applies `rmod`.

- `firstlmod`. This is like `lmod` but it checks only the first left modifier. Similarly there are `lastlmod`, `firstrmod`, and `lastrmod`, where in all cases, `first` and `last` refer to left-right sentence order.

- `onlylmod`. This is like `lmod` but it checks there is only one left modifier, and then tests with that as the node in focus. Similarly there is `onlyrmod`.

- `alllmod` (to be read as "all lmod"). This does a universal quantification over the left modifiers of the node in focus. (The operator `lmod` does an *existential* quantification.) Namely, the test

  (`alllmod` *a b c ...*)

  succeeds iff, for each left modifier $P$ of the node in focus,

  (`&` *a b c ...*)

holds with $P$ as the node in focus – assuming that $a$ is not a slot test. If the first argument is a slot test, like so:

`(alllmod (slot` $S$`)` $b$ $c$ `...)`

then this succeeds iff, for each left modifier $P$ filling slot $S$ of the node in focus,

`(&` $b$ $c$ `...)`

holds with $P$ as the node in focus. So the slot condition constrains the modifiers being quantified over instead of being required as a condition that has to hold for each modifier. It's the same as if you had:

`(alllmod (if (slot` $S$`) (&` $b$ $c$ `...)))`

Variants are allowed where the slot test specifies an option also.

- `allrmod`. Similar to `alllmod` but quantifies over right modifiers.
- `allmod`. Similar. Quantifies first over left modifiers, then over right.

## 7.10 Alternative analyses

For handling disambiguation during parsing, it is quite useful in grammar rules to be able to examine alternative analyses of the current phrase or word being dealt with. There are several SSF operators for doing this conveniently. They examine the *starter phrases* that are built before parsing proper, as described in Section 3 above. When a word has several morpholexical analyses, it will give rise to corresponding starter phrases. The operators described here can take any number of arguments, which should be tests – except in the case of `altf` and its variants.

- `alt`, `altm`, `alth`. These operators look at the current phrase $P$ (phrase in focus, $M$ or $H$, respectively) and go through the starter phrases $Q$ that have the same span as the headword of $P$. For each such $Q$, the argument tests of the operator are applied with $Q$ as the phrase in focus. Any success of such an application results in success of the original test.

  Here is an example. Suppose that we want to say that if a word can be both an adjective and a noun, then only the adjective form can be used to premodify another noun. We can implement this condition as follows: In the rule that attaches noun premodifiers, we can add the constraint:

  `(^ (altm (pos adj)))`

  And then of course the rule that attaches adjectives has to be there.

- `altf`, `altmf`, `althf`. This is similar, but the arguments should be features, and these are tested against the features of the alternative analyses. There is the equivalence:

```
(altf a b a ...) = (alt (f a b c ...))
```

- `altlbrb`. This should have an initial two arguments that represent the left and right boundaries of the starter phrases to be examined. The remaining arguments are applied as tests to such starter phrases (as phrase in focus).

- `lbalt`, `lbaltm`, `lbalth`. These examine starter phrases whose left boundaries coincide with the left boundary of the current phrase, and apply the argument tests to those phrases. So they are basically alternative analyses of the first word of the current phrase.

- `rbalt`, `rbaltm`, `rbalth`. Similar, but the starter phrases have to have right boundaries that coincide with the right boundary of the current phrase.

- `prevalt`, `prevaltm`, `prevalth`. Similar, but the starter phrases have to have right boundaries that coincide with the left boundary of the current phrase. So we are looking at alternative analyses of the word just before the current phrase.

- `nextalt`, `nextaltm`, `nextalth`. Similar, but the starter phrases have to have left boundaries that coincide with the right boundary of the current phrase. So we are looking at alternative analyses of the word just after the current phrase.

## 7.11 Arithmetic operations

Arithmetic expressions can be built up recursively using the usual SSF syntax, and using the four operators:

```
+  -   *  /
```

Each arithmetic operator can take any number of arguments, which recursively are arithmetic expressions. Base arithmetic expressions are integer constants, `double` constants, and simple variables. Example:

```
(+ 3 v2 v5 v7 (* 0.15 v8 (- v3 6.0)))
```

is like

```
3 + v2 + v5 + v7 + 0.15 * v8 * (v3 - 6)
```

in normal infix notation.

All the arithmetic is done on the level of `double`. Integer constants are converted to `double`s, and variables with integer (term) values are converted to `double`s. The value returned is a term of type `double`.

If an arithmetic operator has no arguments, then the value of the expression is `0.0` (in the intermediate computation).

If `+` or `*` has one argument, then its value is the value of that one argument. If `-` has one argument, then its value is the negative of the value of that argument. If `/` has one argument, then its value is the reciprocal of the value of that argument.

For two or more arguments, the result is as if you inserted the operator as an infix operator between the arguments (with the usual left associativity).

Variables whose values are not numeric (or without a value at all) are treated as if the value were `0.0`.

Division by `0.0` is generally "safe" (shouldn't cause a crash) and the result is `0.0`. (The test for `0.0` is `x == 0.0`. There might be some overflow complications if the thing is very close to `0.0` but not equal.)

There are four test operators for numerical comparisons (besides the general equality operator `=` described above):

```
<=  <   >=  >
```

Each takes two arguments, which can be any arithmetic expressions, as just described. These expressions are evaluated to `double`s, and the resulting numbers are compared in the usual way.

Assignment of the value of an arithmetic expression can be done using the operator `calc`. This takes two arguments, the first being a put variable and the second an arithmetic expression. The second argument is evaluated, and its value (as a term) is assigned to the variable of the first argument. The operation always succeeds. Example: If the variable `v1` has value `5` and we call

```
(calc >v2 (/ v1 2))
```

then `v2` will be assigned `2.5`.

## 7.12 Parse scoring operations

In Section 4 we described the parse scoring system. There are three SSF operators that affect parse scoring.

- `eval`. This takes one argument, which should be an arithmetic expression. That expression is evaluated, and its value is *added* to the current parse score of $H$. (Thus a negative argument for `eval` creates a reward.) The change is only for the original $H$ of the slot-filling rule, even if `eval` is called from a context with a different node in focus.

- `prunediff`. In Section 4 we mentioned the "fuzz factor" `prunedelta` for loosening the effects of parse space pruning (and for English this has the default value of 0). The operator `prunediff` allows for a "local" or

temporary effect like that of `prunedelta`. It takes one argument, which should be an arithmetic expression, and the parser takes the maximum of `prunedelta` and the value of that expression as the fuzz factor for the current slot-filling. This operation should only be called from the bodies of the special slot rules for `arb` and `top`. Here is an example from `en.gram`, within the `arb` rule:

```
(if (^ (slot top))
       (if (& (hf vfin)
              (hcite be do have will can may shall must) )
           (prunediff 1.00000001) ))
```

This loosens up pruning in cases where we are modifying a finite auxiliary verb.

- `ceval`. This should be used in handling coordination, and called when filling the left conjunct slot `lconj`. It causes certain general parse score changes to be done to $H$ (the conjoined phrase), having mainly to do with parallelism in coordination. Currently it uses no arguments, and is then just called like so: `(ceval)`.

## 7.13 String operations

The operations described in this section deal witb manipulating strings. The first one, `steq`, is quite general, and can be used for both analyzing and synthesizing strings via pattern matching. We have described pattern matching above on the level of lists (as terms), but this is on the level of the list of characters in a string.

- `steq`. This is called like so:

  ```
  (steq Arg Pattern)
  ```

  For *analysis* of strings, `Arg` should be a simple variable or a string (or atom) constant, and `Pattern` should be a list of terms of the following form:

  1. a string (or atom) constant
  2. a simple variable whose value is a string or atom
  3. a put variable, which matches and is assigned to a single character of `Arg`
  4. a "put sublist variable", of the form `>*v`*i*, which matches and is assigned to a substring of `Arg`

  Example:

```
(&
   (= >v1 constitution)
   (steq v1 (>v2 >*v3 tut >*v4 >v5))
)
```

This assigns:

```
v2 = c, v3 = onsti, v4 = io, v5 = n
```

For *synthesis* of strings, `Arg` should be a put variable, and `Pattern` should consist of ingredients of type (1) or (2) above (no put variables). We can then expand the preceding example to:

```
(&
   (= >v1 constitution)
   (steq v1 (>v2 >*v3 tut >*v4 >v5))
   (steq >v6 (v5 v4 v3 v2))
)
```

which assigns `v6 = niioonstic`.

Here is a sequence of tests from en.gram that handles possessive plurals as in "managers' offices":

```
(lcword >v1) (steq v1 (>*v2 s))
(rb >v3) (punc v3 ')
(lb >v4) (^ (punc v4 '))
```

It tests that $M$ (like `managers'`) ends in `s`, is followed by `'`, and is not preceded by `'`.

- `haschr`. A call `(haschr x y)` tests that the string `x` contains at least one of the characters in string `y`. The arguments can be constants or variables. This operation can be expressed in terms of `steq`, but for the purpose, this one is simpler and more efficient to use. One could test that the phrase in focus has a multiword headword via these two tests:

  ```
  (cite >v1) (haschr v1 \ -/)
  ```

- `allcaps`. This takes one argument, which should be a constant or simple variable, and it tests that the value, as a string, consists of all capital letters.

- `capfirst`. This is called similarly, and it checks that the string begins with a capital letter and any remaining characters are lower case.

- `lcseg`. This is a basic test that checks that the current segment is an "lc (lower case) segment", which basically means that is not like a headline or header phrase, where "content words" are capitalized. It checks that at least one noun or verb in the segment is in lower case.

## 7.14 Flags

- `flag`. This takes one argument which should be an *X***SG** system flag name, and it checks that this flag is turned on.
- `sa`. This takes any number of arguments that should name subject areas (like `entertainment`), and it checks that at least one of these is active.

Flags and subject areas can be controlled in the API to *X***SG** – see [10], "Using Slot Grammar".

## 7.15 I/O

- `prt`. This takes any number of arguments, which are evaluated, and the values are printed in succession on the standard SG output. The operation always succeeds. This is useful for debugging rules by inserting such print commands in the rule bodies.
- `prtnl`. This does the same as `prt`, and then prints a newline at the end. (Note then that `(prtnl)` would simply print a newline.)

## 7.16 Postponed tests

As mentioned above, the right modifiers of a phrase are attached before the left modifiers. And yet, in attaching a right modifier, occasionally one needs to know what kinds of left modifiers are (eventually) present. For example, in the phrase

*as happy as John*

the right modifier PP *as John* fills an adjunct slot (called `acompar` in `en.gram`) for the adjective *happy*. And yet we would want to know that *happy* has a left modifier of *as* or *so*. We can accomplish such a test on the current $H$ with the operator `htest`.

This operator can take any number of arguments, which should be tests. But the tests are not applied for the current slot-filling. They are *postponed* until the current $H$ phrase itself (hence the name "htest") becomes a filler, or is to be taken as the top node, perhaps after receiving more modifiers than the current one. This is accomplished in the system by storing the argument tests of `htest` in the **phrase tests** field of the current $H$, mentioned above in Section 2. As the current $H$ is built up by receiving more modifiers, any calls to `htest` will add more phrase tests to $H$. Then they get applied, and must succeed, when $H$ is finally *used* as a modifier (or top node) itself. A call to `htest` itself always succeeds in the slot rule that calls it. But the new $H$ created by that slot-filling will never "live" unless the argument tests succeed when $H$ is finally used.

When the argument tests of `htest` are finally applied, there is a context shift, because what *was* a higher phrase ($H$) will now be a modifier ($M$). And there

will be a still higher $H$ now (unless we are at a top node). So in the argument tests of `htest`, operators that apply to current $H$ should be written as if that phrase is a modifier, etc.

Here is an example extracted from the `acompar` adjunct slot-filling rule in `en.gram`.

```
prep <
   (hpos adj)
   ri
   (mf asprep)
   satisfied
   (htest (lmod (cite as so)) (eval -1))
   (setslot acompar p)
```

The first argument test of `htest` checks that $H$ has a left modifier with citation form `as` or `so`, and the second argument provides a reward for this.

# 8 Tracing, debugging and testing

In this section we discuss some tools for grammar development. To start off, we look at flags that can be set to control various aspects of parsing.

To set a flag $F$ on (or off) in interactive (`Input sentence`) mode, one can type `+`$F$ or `-`$F$ respectively. Or one can type `+`$F$ $n$ to set the value to any non-negative integer value $n$. (`-`$F$ is equivalent to `+`$F$ 0.) One can also set flags on or off when invoking the *X***SG** executable by giving it arguments like so:

`-on` *Flag*
`-off` *Flag*

Here is a list (in alphabetical order) of some relevant available flags and their meanings. We will usually explain what the effect of having the flag *on* is.

`all`. Process all of the top-level parses. When `all` is off, only the first (best-ranked) parse is processed. *Processing* a parse means displaying it, if the flag `syn` is on, and calling any post-parse functions that one sets up through the *X***SG** API. But for developing a grammar it is usually best to have `all` on.

`aotrace`. Trace the action of affix operations in morphology (for versions of *X***SG** that use the SG morphology system).

`deptree`. Controls the basic form of parse tree displays. We describe this below.

`derivmorph`. This enables derivational morphology for versions of *X***SG** that use SG morphology. It is on by default. If it is off, and a word structure has a derivational (non-inflectional) affix, then morphology skips that structure.

`displinecut`. If the current input is within a *display tag* (see `dodisplays`), then newlines break segments, i.e., segments cannot span across lines. On by default.

`dodisplays`. Certain tags are designated *display tags* in the shell (and which ones they are depends on the formatting language). The flag `dodisplays`, which is on by default, enables processing (parsing, etc.) of text within display (begin and end) tags.

`doshowstat`. Show parsing statistics for a run of *X***SG** on a document. Output goes at the end of the output file. On by default.

`echoseg`. Echo (print out) the input segment for each parse. On by default.

`echosegext`. Echo segment-external material in output. Off by default.

`fftrace`. Activates strongest (most detailed) of parse tracing. Discussed below.

`ftrace`. Activates next strongest of parse tracing (after `fftrace`). Discussed below.

`fullfeas`. Activates display of additional phrase features in parse output, namely the features:

```
le1, le2, le3, le4, ri1, ri2, ri3,
xtra, cord, comcord, unitph, lcase, glom
```

Among the `le`$n$, which are customarily used to indicate presence of left modifiers, it shows only the strongest one – the one with greatest $n$ which the phrase has as a feature. Similarly for the `ri`$n$, for right modifiers. This flag is off by default.

`html`. Enables processing of HTML documents. On by default. The flag `sgml` should also be set on when `html` is on.

`linemode`. Causes newlines to break segments (so that segments cannot span across lines). Useful for regression testing. A segment (on a line) need not have a segment terminator (like a period, question mark, etc.). Off by default.

`linesyn`. Causes the parse display to be written all on one line. Useful for regression testing. Off by default.

`longlineout`. When the input segment is echoed, it is written all on one line. Useful for regression testing. Off by default.

`ltrace`. Enables tracing of the results of morpholexical analysis. Often very useful for debugging *X*`.gram`.

`net`. Enables building of the *SG network* representation of each parse tree. On by default.

`nettrace`. Enables tracing output of the SG network.

`noparse`. If this flag is turned on, then only the morpholexical part of parsing will be done; the chart parsing will be skipped.

`otext`. Enables processing of Otext documents. The flag `sgml` should also be turned on when `otext` is on. Off by default.

`ptrace`. This enables the weakest form of parse tracing. Discussed below.

`predargs`. This is on by default, and in parse displays, it enables word senses to be shown with their arguments, like:

```
believe2(2,1,3,4)
```

When it is off, you would see just `believe2` in the head sense position.

`predargslots`. When this is turned on, it causes word sense predications to be displayed with slot names attached to the arguments, like so:

```
believe2(2,subj:1,obj:3,comp:4)
```

`prefnp`. This is used in **ESG**. When it is on, short top-level segments that have *some NP* analysis are preferred as *NPs* – the shorter, the more preference. Off by default.

`printinc`. When this is turned on, and a file *fn.ft* is parsed, the segment strings that produce an incomplete parse will be output to the file *fn*.`inc`. This is quite useful for developing *X*`.gram`; it is a "free" way of finding bad parses that should be taken care of.

`printsentno`. When this is on (and it is on by default), and a file is parsed, the segment number of each segment will be printed before the segment string in the output file.

`prune`. Enables parse space pruning. On by default.

`semicolonsep`. Causes semicolons to be treated as segment terminators. It is on by default. When it is off, semicolons will be treated as punctuation conjunctions.

`sgml`. Enables processing of SGML texts. It is on by default. Even when it is on, plain text is handled because plain text is just viewed as SGML text with no tags.

`showaopts.` When this is turned on, the chosen option $O$ for each *adjunct* slot $S$ will be shown in parses in the form $S$`(`$O$`)`. So for example for the determiner `the`, one would see the slot `ndet(dt)` instead of just `ndet`. Options are always shown for *complement* slots.

`shownumparses.` In parse output, shows the total number of parses obtained. On by default.

`shownumsent.` When this is on and a file is parsed, the segment number of each segment will be printed to the console. On by default.

`showsense.` In word sense predications in parse displays, this causes sense names of the words to be used as the predicates. It is overridden by `showssense`. If both `showsense` and `showssense` are off, citation forms will be used for the predicates. On by default.

`showsf.` Show the semantic types for each phrase in the parse tree. Off by default.

`showslots.` Show the available slots of each phrase in the parse tree. Off by default.

`showssense.` For the predicate of each word sense predication in parse displays, this shows a deeper kind of sense expression for some word senses (details not given here). Off by default.

`spacelinecut.` If this on, then each line of input text that consists only whitespace will break the current segment. Off by default.

`stortrace.` Causes tracing of storage used for analyzing each sentence. Off by default.

`syn.` Causes parse trees to be printed out. On by default.

`timit.` Causes tracing of time used for analyzing each sentence. On by default.

`toktrace.` Causes tracing of tokenization.

`toktrace0.` Causes tracing of tokenization at a stage prior to that for `toktrace`; the state of tokenization is shown before the call to a function `adjustsegment`, which is responsible for various adjustments to the token list, such as splitting of contractions.

`wstrace.` Causes tracing of affix stripping in SG morphology (the "`ws`" is for "word structure").

`xout.` When it is on, results go, in interactive mode, to the file `sg.out`. When it is off, they go to the console. Off by default.

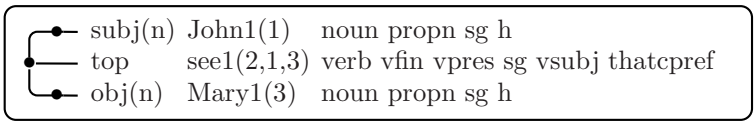
The flag `deptree`, which controls parse tree display, can have any of the values 0, 1, 2, or 3. The default value is 1. Let us show what the displays look like for **ESG** and the sentence *John sees Mary*. You can experiment with these by typing

`+deptree` $n$`.`

in `Input sentence` mode, where $n$ is the desired value. For value 0, the display is:

```
top verb vfin vpres sg vsg vsubj thatcpref
   subj(n) noun propn sg h
      John1(1)
   see1(2,1,3)
   obj(n) noun propn sg h
      Mary1(3)
```

For value 1 (the default), it is the standard SG display:




For value 2, we get an XML-tagged form:

```
<seg start="0" end="15" text="John sees Mary.">
<ph id="2" slot="top" f="verb vfin vpres sg vsg vsubj thatcpref">
   <ph id="1" slot="subj(n)" f="noun propn sg h">
      <hd w="John" c="John" s="John1" a=""/>
   </ph>
   <hd w="sees" c="see" s="see1" a="1,3"/>
   <ph id="3" slot="obj(n)" f="noun propn sg h">
      <hd w="Mary" c="Mary" s="Mary1" a=""/>
   </ph>
</ph>
</seg>
```

For value 3, we get the same tagged display without indentation:

```
<seg start="0" end="15" text="John sees Mary.">
<ph id="2" slot="top" f="verb vfin vpres sg vsg vsubj thatcpref">
```

```
<ph id="1" slot="subj(n)" f="noun propn sg h">
<hd w="John" c="John" s="John1" a=""/>
</ph>
<hd w="sees" c="see" s="see1" a="1,3"/>
<ph id="3" slot="obj(n)" f="noun propn sg h">
<hd w="Mary" c="Mary" s="Mary1" a=""/>
</ph>
</ph>
</seg>
```

This completes the inventory of flags relevant for grammar development. Let us look now at ways of invoking the parser.

The most immediate method of invoking the parser is just to call the executable *X***SG** with no arguments, so that it enters *interactive mode*, in a loop that asks for input with an `Input sentence` prompt. In this mode you can type in sentences, perhaps across several lines, until the segmenter breaks the segment. The output goes to the console if the flag `xout` is off; otherwise the output goes to the file `sg.out`, and the current `sgeditor` is called on that file.

On Windows this editor is set by default to `notepad` or `kedit` (depending on the compilation) and on AIX to `emacs`. You can set the name of the SG editor in `Input sentence` mode if you type:

`sgeditor` *NameOfEditor*`.`

Or when you invoke the *X***SG** executable, you can give it the parameters:

`-sgeditor` *NameOfEditor*

In interactive mode, when you are typing in various sentences and you want to repeat the parsing of the most recent sentence (perhaps after setting some flags or giving other commands), you can type:

`redo.`

In interactive mode you can make *X***SG** process a whole file by typing:

`do` *InFile OutFile*`.`

If you omit the *OutFile*, then output will go either to `sg.out` or to the console according as `xout` is on or off. It will send results to `sg.out` a segment at a time.

Another way to process a whole file is to invoke the *X***SG** executable with arguments as follows:

*X***SG** `-dofile` *InFile OutFile*

The *InFile* can actually be a file pattern, like `\texts\*.htm`, and it will process all the files matching that pattern, sending all the output to the same output file. You can omit the *OutFile*, in which case the output will all go to `sg.out` (all the results in one step). Or you can specify *OutFile* as the empty string `""`, in which case the output will go to `sg.out` one segment at a time if `xout` is on, or to the console otherwise. You can give *X***SG** `-dofile` optional arguments of the form:

`-on` *Flag*
`-off` *Flag*
`-sa` *SubjectArea*(*s*)

to turn a flag on or off or to set subject areas for the run. If there is more than one subject area, you should use `-sa` just once, and let its "argument" contain all the desired subject areas, enclosed in double quotes and separated by blanks, like so:

```
fsg -dofile -sa "computers instructions"
```

Probably the most useful kind of interface in grammar development is `ldo`, which works only with certain editors. In interactive mode with *X***SG** you can type:

`ldo` *File*`.`

This will open up *File* with the editor. Then you can place the cursor on the first line of any sentence (the cursor need not be at the beginning of the sentence) and press F6. Then *X***SG** will parse that sentence and show the results in `sg.out`. When you press F3 in the editor, you will return to editing *File* in the same spot you were.

You can edit and change sentences in *File*, in order to experiment with variations of them, and again press F6 to parse them. As long as you do not save the file, when you return from `sg.out`, *File* will be in its original state (before you made a variation of the test sentence). This convenient way of trying variations of sentences is one thing that makes `ldo` so valuable in debugging the grammar. When you have a problem sentence to work on, you can reduce it to the simplest form where the problem still exists. For instance if the parse of a sentence is incomplete, there may be some subclause that does not parse, and the problem lies there.

When you are editing *File* in `ldo` mode, you can press F3 to return to the *X***SG** command line. This will leave the file in its original state, even if you have made changes – as long as you do not save the file. Of course you may want to save the file, because you have found some useful variations of sentences to work on later.

Once you have used the above `ldo` command with a given file argument, you can just type

```
ldo.
```

even after leaving *X***SG** and reinvoking it, and it will return to the same file you were working on, and on the same line where the cursor was when you left the file with F3.

In preparing a test file for use with `ldo`, you should arrange it so that no two sentences share any of the same line. Start each new sentence on a new line.

The `ldo` software exists partly in the C code of the SG shell and partly in the macro language of the editor. Currently there are editor macro programs in Kedit on Windows and Xedit on VM.

As you can see from the above inventory of flags, the main flags for enabling parse tracing are these:

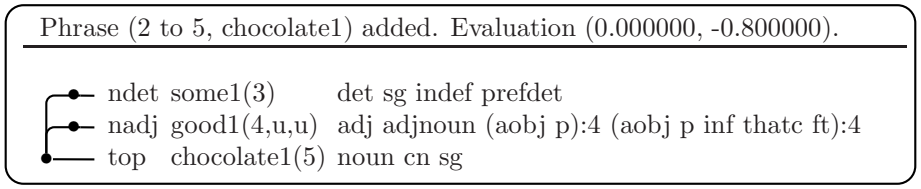
```
ptrace, ftrace, fftrace
```

They show increasing levels of information about slot-filling attempts.

With `ptrace`, you see basic information about which phrase analyses are added into the chart during parsing, and which ones are not added or are deleted because of parse space pruning. For instance with the sentence

*I ate some good chocolate.*

one of the outputs of `+ptrace` would be:




Note the "2 to 5". These are the left and right boundaries of the added phrase. In searching for the analyses for a given subphrase of the sentence, you can use such boundary numbers and the form "*m* `to` *n*". The pair of numbers after "Evaluation" are the `reward` and `eval` fields of the added phrase. Looking at the output from `ptrace` is a good way of learning the basic steps of the chart parsing.

With `ftrace`, you see additional information – about attempts to fill specific slots. Consider the situation for *I ate some good chocolate* where the noun phrase *some good chocolate* has been formed and the parser tries to attach it to *I ate* by filling the `obj` slot of the latter. Then `ftrace` causes the message:

```
slot = obj, mod = 2 5 chocolate1, matrix = 0 2 eat1 (complement)
```

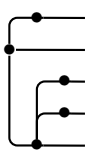
We see here (a) the slot being tried, (b) the (possible) modifier's boundaries and headword sense, (c) similar things for the matrix phrase, and (d) the type of

slot. After such a slot message, we see the options of that slot being attempted, in order. For `ftrace`, each option is displayed only by its name, an arrow, and the part of speech of the filler. If an option rule succeeds, we see a message "`option matched`". Then if the slot rule succeeds, we see messages for that also. In our example then we see:

```
n ==> noun
option matched
slot rule or 'satisfied' succeeded
slot = obj, filled
```

And when a slot-filling succeeds, we see a message (as for `ptrace`) for adding the resulting new phrase, in this case:

Phrase (0 to 5, eat1) added. Evaluation (0.000000, -0.780000).



| | | |
|---|---|---|
| subj(n) | I1(1) | noun pron pl pers1 nom h perspron |
| top | eat1(2,u,u,u) | verb vfin vpast pl vsubj (comp pt pt):2 |
| ndet | some1(3) | det sg indef prefdet |
| nadj | good1(4,u,u) | adj adjnoun (aobj p):4 (aobj p inf thatc ft):4 |
| obj(n) | chocolate1(5) | noun cn sg |

With `fftrace`, we get additional information about features. After a line about a particular slot, we see also the features of the candidate modifier and matrix, like so:

```
slot = obj, mod = 2 5 chocolate1, matrix = 0 2 eat1 (complement)
   mf: noun cn sg
   hf: verb vfin vpast pl vsubj
```

And for an attempted slot option, we see the whole representation of the slot option rule, with all the features, like so:

```
n ==> noun (^ wh) < ri (^ vpass ri3) (+ ri2)
```

If no slot option succeeds, then `ftrace` and `fftrace` will just print all the options for that slot, with no message about failure. For example, for the subphrases *I ate* and *some good chocolate* of our sample sentence, **ESG** will also try to fill the `comp` slot of the verb sense `eat1`, because the lexical entry for `eat` begins as:

```
eat
   < v obj (pt out up away)
   ...
```

For this, `fftrace` prints out:

```
slot = comp, mod = 2 5 chocolate1, matrix = 0 2 eat1 (complement)
   mf: noun cn sg
   hf: verb vfin vpast pl vsubj

   pt ==> prep (^ ri1) ctest=out;up;away < ri (^ ri3) (+ ri2)
   pt ==> adv partf (^ ri1) ctest=out;up;away < ri (^ ri3) (+ ri2)
```

No option succeeds because the filler is an $NP$.

For debugging a grammar, it is also useful to insert print statements in slot rules using operators `prt` or `prtnl` (see Subsection 7.11).

*Regression testing* is an extremely important part of grammar development. The SG shell provides convenient tools for doing this.

To make the regression comparisons convenient, the system needs to output the results of parsing a test file in lines of the form

$Segment_1$
$Parse_1$
$Segment_2$
$Parse_2$ ...

where of course $Parse_i$ is the parse tree of text $Segment_i$. Each text segment and each parse tree need to be written on a single line.

To get this set up, it is best to get each test file in a form where there is exactly one segment per line. There is a "built-in" SG utility for doing this. If $InFile$ is your raw test file (where segments may span several lines and where there may be lots of tags), you can type:

*X***SG** `-segfile -notags` $InFile$

The resulting one-per-line segments will be put in a file which has the same name as $InFile$ but has extension `.seg`. The file will go in the same directory as $InFile$. The $InFile$ name can be a file pattern instead of the name of a single file, and then `-segfile` processes all the files matching that pattern and puts all the segments in a single file. In this case it is best to give a file name to the output file, and this is done with an extra pair of arguments:

`-outfname` $FName$

So for example we might do:

```
ssg -segfile -notags \texts\*.htm -outfname htmtest
```

Then all the segments from files of the form `\texts\*.htm` will put (one per line) in the output file `\texts\htmtest.seg`.

As with `-dofile`, the command `-segfile` allows arguments

`-on` *Flag*
`-off` *Flag*

for turning on or off flags.

The next step in regression testing is to run *X***SG** on such segment files in order to get output in the desired form of alternating lines of text segments and parse trees, described above. You can do this with a command that is very similar to `-dofile` in its form and options:

*X***SG** `-sgtest` *InFile OutFile*

Again, the *InFile* string can name a single string or a pattern of files. Again, you can specify arguments that turn flags on or off, or set subject areas, in the same forms as for `-dofile`. If the *InFile* has been created by `-segfile`, then you should give `-sgtest` the additional arguments `-on linemode` (see page 39 for the flag `linemode`).

The next step, after a running a regression test more than once on the same input, is to compare the results with those of the previous run. For this, you can use:

*X***SG** `-compare` *OldFile NewFile* `-compfile` *CompFile*

Here the *OldFile* is the previous test result file, *NewFile* is the new one, and *CompFile* is where the comparisons (differences) go. In the comparison file, `-compare` will show only the test segments where there is a *different* parse. For each of these, it prints out the segment itself, the old parse, and the new parse. Even though the parses in the test result files are all on one line, here they will be displayed in the usual form (according to the current setting for `deptree`).

Here is a more specific example for the whole process, illustrated for Windows. Suppose you have a test file `stest1.seg` of Spanish segments in the desired form – one segment per line. Since you do this testing regularly, you make a `.bat` file `stest1.bat` with the three lines:

```
copy stest1.new stest1.old
erase stest1.new
ssg -sgtest stest1.seg stest1.new -on linemode
```

So we run `stest1`, make some improvements to **SSG**, and run `stest1` again. For the comparison, we have another `.bat` file `scomp1.bat` with the two following lines in it:

```
ssg -compare stest1.old stest1.new -compfile stest1.cmp
kedit stest1.cmp
```

It is possible to run `-compare` on a whole collection of test result files. The comparisons get put in a single comparison file. Here is how you do it. Suppose we have several test files, as follows, maybe representing our whole regression test suite:

```
test1.seg
test2.seg
...
```

Assume that we just run `-sgtest` separately on these files. To automate it, we could have a `.bat` file `regtest.bat` with the contents:

```
copy test1.new test1.old
erase test1.new
XSG -sgtest test1.seg test1.new -on linemode
copy test2.new test2.old
erase test2.new
XSG -sgtest test2.seg test2.new -on linemode
...
```

We also create a *file list file* for `-compare` with contents:

```
test1.old test1.new
test2.old test2.new
...
```

Call this file `regcomp.fl` (the name is arbitrary). Finally, we make another `.bat` file, `regcomp.bat`, with the two following lines in it:

```
XSG -compare -flist regcomp.fl -compfile reg.cmp
kedit reg.cmp
```

Then when we execute `regcomp`, we will be editing a single file `reg.cmp` that shows all the parse differences for our whole regression test suite. So regression testing becomes easy and automated, consisting just of calling `regtest` and `regcomp` and looking at the comparisons in a single file. Each comparison will be labeled with the name of the test file that it originated in.

The reader should be able to see the general rule about the `-flist` version of `-compare`. Instead of giving `-compare` a single pair $(OldFile, NewFile)$, of file names, we put all those pairs of names in a file $Pairs$, and then call

*X*`SG -compare -flist` $Pairs$ `-compfile` $CompFile$

In the preceding examples, we have omitted directory paths, but any of the file names can be qualified by such paths.

# References

[1] Veronica Dahl and Michael C.McCord. Treating coordination in logic grammars. *Computational Linguistics*, 9:69–91, 1983.

[2] Shalom Lappin and Michael C. McCord. Anaphora resolution in Slot Grammar. *Computational Linguistics*, 16:197–212, 1990.

[3] Shalom Lappin and Michael C. McCord. A syntactic filter on pronominal anaphora for Slot Grammar. In *Proceedings of the 28th Annual Meeting of the ACL*, pages 135–142, 1990.

[4] Michael C. McCord. Slot Grammars. *Computational Linguistics*, 6:31–43, 1980.

[5] Michael C. McCord. Using slots and modifiers in logic grammars for natural language. *Artificial Intelligence*, 18:327–367, 1982.

[6] Michael C. McCord. Modular logic grammars. In *Proceedings of the 23rd Annual Meeting of the ACL*, pages 104–117, 1985.

[7] Michael C. McCord. Slot Grammar: A system for simpler construction of practical natural language grammars. In R. Studer, editor, *Natural Language and Logic: International Scientific Symposium, Lecture Notes in Computer Science*, pages 118–145. Springer Verlag, Berlin, 1990.

[8] Michael C. McCord. Heuristics for broad-coverage natural language parsing. In *Proceedings of the ARPA Human Language Technology Workshop*, pages 127–132. Morgan-Kaufmann, 1993.

[9] Michael C. McCord. SLF: The Slot Grammar lexical formalism. Technical report, IBM T. J. Watson Research Center, 2006. RC 23977.

[10] Michael C. McCord. Using Slot Grammar. Technical report, IBM T. J. Watson Research Center, 2006. RC 23978.

[11] Michael C. McCord and Susanne Wolff. The lexicon and morphology for LMT, a prolog-based MT system. Technical report, IBM T. J. Watson Research Center, 1988. RC 13403.

[12] Adrian Walker, Michael C. McCord, John F. Sowa, and Walter G. Wilson. *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Addison-Wesley, Reading, MA, 1987.