RC25340 (WAT1211-043) November 14, 2012
Computer Science
(Revision of RC23978)

IBM Research Report

Using Slot Grammar

Michael C. McCord
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

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

Using Slot Grammar

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

Abstract

This report describes how to use a Slot Grammar (SG) parser in applications, and provides details on an API. There is a companion report, "The Slot Grammar Lexical Formalism". Both reports are written in a fairly self-contained way. The current one can serve as an introduction to Slot Grammar and as a kind of user guide. Topics covered include: (a) An overview of SG analysis and parse displays. (b) The use of SG in interactive mode, including an editor-based interactive mode. (c) The use of SG on whole documents, or collections of documents. (d) Descriptions and inventories of SG slots and features – both morphosyntactic features and semantic types. (e) Controlling the behavior of the parser with flag settings. (f) Tag handling and annotation methods for named entities and other text chunks. (g) The parse tree data structures. (h) The API and compilation of SG-based applications. (i) Handling user lexicons. (j) Logical form production.

1 Introduction

In this report we describe how to use a Slot Grammar (SG) parser in applications. There is a companion report [19], "The Slot Grammar Lexical Formalism", which describes SG lexicons in enough detail that the reader should be able to create such a lexicon for a new language. Both reports are written in a fairly self-contained way.

There is also a report, [18], "A Formal System for Slot Grammar", which describes a formalism (SGF) for writing the syntax rules for a Slot Grammar. But this methodology is not currently used in SG.

The main current implementation of SG is in C. The SG parsers can be used in executable form, or via library (or DLL) versions, with an API described in this report. And there is a bridge of the C implementation to Java, developed by Marshall Schor in the context of UIMA (Unstructured Information Management Architecture) – see `http://www.ibm.com/research/uima`.

Currently there are Slot Grammars for English, French, Spanish, Italian, Brazilian Portuguese and German, which we call, respectively, **ESG**, **FSG**, **SSG**, **ISG**, **BPSG**, and **GSG**. There is a single syntactic component, `ltsyn`, for

the Latin-based languages (French, Spanish, Italian, Portuguese), with switches for the differences. We will use "*X***SG**" to refer generically to one of the Slot Grammars.[1]

The examples in this document are given mainly for English, but most of the material applies to all the languages for which we have Slot Grammars. Most of the slot and feature names are the same for all languages (are shared across languages), but we list most of the exceptions. The data structures and the API are the same for all languages.

*X***SG** runs on a variety of platforms, including Linux, Unix, AIX, all Windows 32 platforms, Solaris, HP, and IBM mainframes. Where it is relevant below, we will use terminology from Windows.

Both single-byte and Unicode versions of *X***SG** are available. The single-byte version uses ISO 8859-1 (which is a "subset" of Unicode).

The rest of the report is organized as follows:

- Section 2, page 3: "Overview of Slot Grammar analysis structures"
- Section 3, page 6: "Running the executable"
- Section 4, page 9: "Options for parse tree display"
- Section 5, page 10: "Processing a file of sentences"
- Section 6, page 11: "The `ldo` interface"
- Section 7, page 12: "Features"
- Section 8, page 25: "Slots and slot options"
- Section 9, page 33: "Flags"
- Section 10, page 39: "Tag handling"
- Section 11, page 40: "Multiwords, named entities, and chunks"
- Section 12, page 54: "The data structure for SG parse trees"
- Section 13, page 59: "Punctuation and tags in SG analyses"
- Section 14, page 63: "Compiling your own *X*`SG` application"
- Section 15, page 67: "Using your own lexicons"
- Section 16, page 71: "Lexical augmentation via WordNet"
- Section 17, page 75: "Logical Forms and Logical Dependency Trees"
- Section 18, page 82: "ESG parse trees in Penn Treebank form"

[1]The author developed initial versions of the Slot Grammars and continued to develop **ESG** through the Fall of 2012. Claudia Gdaniec took over **GSG** and developed most of it to its current state. Esméralda Manandise took over the Latin languages syntactic component `ltsyn`, with special emphasis on Spanish. Contributions to the lexicons and morphology have been made by Claudia Gdaniec and Marga Taylor for German, by Consuelo Rodríguez, Esméralda Manandise, Sue Medeiros and Joaquín Salcedo for Spanish, by Esméralda Manandise for French, and by the Synthema company for Italian and Portuguese.

2 Overview of Slot Grammar analysis structures

As the name suggests, Slot Grammar is based on the idea of *slots*. Slots have two levels of meaning. On the one hand, slots can be viewed as names for syntactic roles of phrases in a sentence. Examples of slots are:

`subj`	subject
`obj`	direct object
`iobj`	indirect object
`comp`	predicate complement
`objprep`	object of preposition
`ndet`	NP-modifying determiner

On the other hand, certain slots (complement slots) have a semantic significance. They can be viewed as names for argument positions for predicates that represent word senses.

Let us illustrate this. Figure 1 shows an example of slots, and the phrases that *fill* them, for the sentence *Mary gave John a book*. It shows for example that *Mary* fills the `subj` (subject) slot for the verb *gave*, *John* fills the `iobj` slot, etc. One can see then that the slots represent syntactic roles.

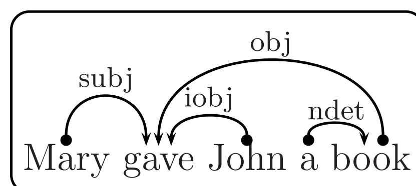


Figure 1: Slot filling for *Mary gave John a book*

To illustrate the semantic view of slots, let us understand that there is a word sense of *give* which, in logical representation, is a predicate, say $give_1$, where

(1) $give_1(e, x, y, z)$ means "e is an event where x gives y to z"

The sentence in Figure 1 could be taken to have logical representation:

(2) $\exists e \exists y(book(y) \wedge give_1(e, Mary, y, John))$

From this logical point of view, the slots `subj`, `obj`, and `iobj` can be taken as names for the arguments x, y, and z respectively of $give_1$ in (1). Or, one could say, these slots represent *argument positions* for the verb sense predicate.

Slots that represent predicate arguments in this way are called *complement slots*. Such slots are associated with word senses in the Slot Grammar lexicon – in *slot frames* for the word senses. All other slots are called *adjunct slots*. An

example of an adjunct slot is `ndet` in Figure 1. Adjunct slots are associated with parts of speech in the syntactic component of the Slot Grammar.

Natural language often provides more than one syntactic way to express the same logical proposition – where the variations can express extra ingredients of emphasis, topic, and the like.[2] Figure 2 shows an alternative syntactic way of expressing the same basic proposition as in Figure 1.

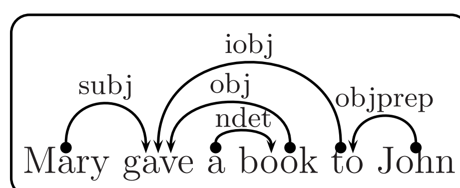


Figure 2: Slot filling for *Mary gave a book to John*

In fact, the logical representation for this sentence (not counting differences in emphasis, etc.) is the same as in (2) above. The sentence analysis in Figure 2 uses the *same* complement slot frame

(3) `(subj, obj, iobj)`

as that in Figure 1. In both cases, the frame comes from the same word sense entry for *give* in the lexicon. But the `iobj` slot is filled differently in the two sentences – in the first case by the NP *John*, and in the second case by the PP *to John*. And the orderings of the slot fillers are different in the two examples. But the syntactic component of the Slot Grammar knows about these alternative syntactic ways of using slots – alternatives that lead to the same basic predication in logical form.

Because of this dual role of slots, Slot Grammar parse trees show two levels of analysis – the surface syntactic structure and the deep logical structure. The two structures are shown in the *same* parse data structure. The full form of the SG parse tree is illustrated in Figure 3, for the sentence *Mary gave a book to John*.

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.

The lines of the parse display are in 1-1 correspondence with the (sub-)phrases, or nodes, of the parse tree. And generally each line (or tree node) corresponds to a word of the sentence. (There are exceptions to this when multiword analyses are used, and when punctuation symbols serve as conjunctions.) Slot Grammar is *dependency-oriented*, in that each node (phrase) of the parse tree has a head

[2]These extra ingredients should figure in the logical representation also.

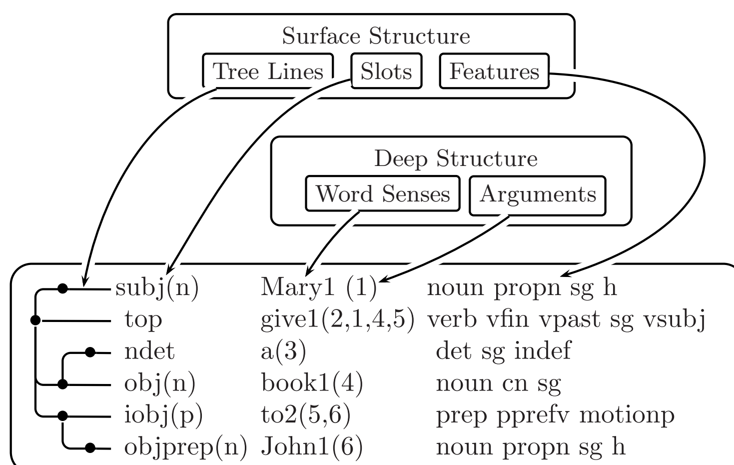


Figure 3: Ingredients of a Slot Grammar analysis structure

word, and the daughters of each node are viewed as modifiers of the head word of the node.

So on each line of the parse display, you see a head word sense in the middle section, along with its logical arguments. To the left of the word sense predication, you see the slot that the head word (or node) fills in its mother node, and then you can follow the tree line to the mother node. We describe the roster of possible slots in Section 8. To the right, you see the features of the head word (and of the phrase which it heads). The first feature is always the part of speech (POS). Further features can be morphological, syntactic, or semantic. We describe the possible morphosyntactic features in Section 7. The semantic features are more open-ended, and depend on the ontology and what is coded in the lexicon.

What are the arguments given to word sense predicates in the parse display? The first argument is just the node index, which is normally the word number of the word in the sentence. This index argument can be considered to correspond to the *event* argument e in (1) above (with a broad interpretation of "event"). The remaining arguments correspond to the complement slots of the word sense – or rather to the fillers of those slots. They always come in the same order as the slots in the lexical slot frame for the word sense. So for a verb, the first of these complement arguments (the verb sense's second argument) is always the *logical* subject of the verb. Generally, all the arguments are *logical* arguments. So passivized expressions are "unwound" in this logical representation. This is illustrated in Figure 4, the parse for the passive sentence *The book was given to John by Mary.*

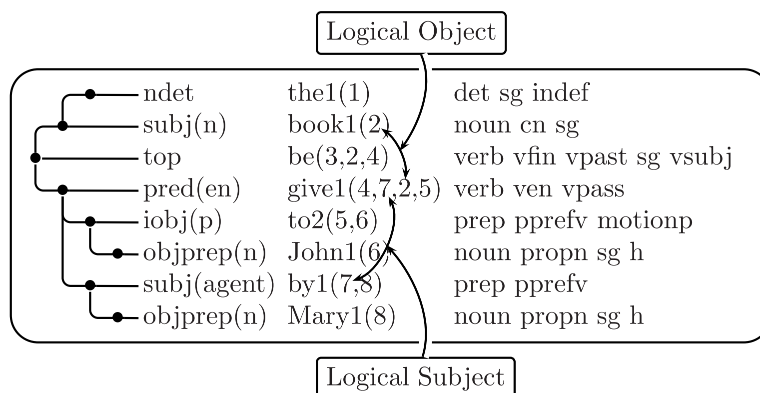


Figure 4: Parse of a passive sentence

Note that the word sense predication give1(4,7,2,5) appropriately has node 7, *by Mary* (and from this *Mary*), as its logical subject, and has node 2, *the book*, as its logical object.

The type of parse display shown in this section is close to the default display for *X***SG**. Actually, what we show here uses some special techniques (with PSTricks) in LaTeX, especially for drawing the tree lines. This special LaTeX form can be produced automatically by the *X***SG** parser when a certain flag is turned on, and this was done for the examples of this kind in this document. The default parse display mimics the nice tree lines with ASCII characters, but it is still readable.

There are several other options for parse displays (or parse output), and these are described in Section 4.

The actual C data structures for parse trees are described in Section 12.

3 Running the executable

The files needed to run *X***SG** are the executable `xsg` plus the lexical files. For **ESG** the lexical files are:

```
en.lxw     en.lxv
```

The lexical files for **GSG**, **SSG**, **FSG**, **ISG** and **BPSG** are similar to those for **ESG**, with `en` replaced by `de`, `es`, `fr`, `it` and `bp`, respectively.

When you run the executable `xsg` without any arguments, you will be in a loop with a prompt "`Input sentence:`". We call this *interactive mode*. From this you can type in sentences (spanning several lines if you want), or other special commands. All sentences have to end with a sentence terminator, and

all commands have to end with a period. Commands should be on one line. You can end the session by typing

```
stop.
```

or `Ctrl-C`. (Generally `stop` is better because it allows *X***SG** to release storage, and some operating systems may not do this automatically.)

Parse tree output may be seen directly on the console, or in various editors. There are interfaces on Windows to Notepad, Vim, Kedit, and Epsilon, on Linux/Unix to Vim and Emacs, and on VM to XEDIT. The version of *X***SG** that you get may be set by default to one or the other of these. If you want to direct output to the console, type:

```
-xout.
```

The command

```
+xout.
```

then causes the output to go to the file `sg.out` and then the chosen editor is invoked on that file. When you leave the file, you will be back in interactive mode.

If you have another editor, called by name E, then the command

`sgeditor` E.

will cause later output to go to that editor.

If you want to give an operating system command C while in the `Input sentence` loop, then you can type

/C.

For instance,

```
/dir /w.
```

would do a directory listing.

When the executable `xsg` is called, it can take command-line arguments that accomplish various things. These same command-line arguments can also be passed into an initialization function `sgInit` for the DLL form of *X***SG**, as described below in Section 14.

One kind of command-line argument is `-dofile`, described below in Section 5, which allows you to run *X***SG** on a text file.

Several command-line arguments allow various settings for *X***SG**. The most important are *flag* settings, for the flags discussed below in Section 9. These commands are of the form:

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

which, respectively, turn *Flag* on or off.

The remainder of this section is a bit more esoteric, and could be skipped on first reading.

The command-line argument pair

```
-sentlen N
```

where *N* is an integer, sets the maximum length of a segment (in words, not counting punctuation or tags) that *X***SG** will attempt to parse. The default is 60. It should not be set higher than 100.

The command-line argument pair

```
-timelimit N
```

where *N* is an integer, sets the maximum time in milliseconds that will be spent parsing a segment. The default is 15000.

One can also use command-line arguments to change default storage allocations for *X***SG**. These allocation commands are of the following four forms, where *N* is an integer:

```
-stStorage N
-ptStorage N
-PstStorage N
-PptStorage N
```

The first, `-stStorage`, determines the number of characters in the temporary string storage buffer used by *X***SG**. "Temporary" refers to the fact that the buffer is zeroed after each segment is parsed. The default value varies with the installation of *X***SG**, but is in a range like 1M to 4M. The second, `-ptStorage`, determines the number of bytes in a buffer used for temporary storage of pointers and numbers. Its default value is in the range 4M to 20M. The third and fourth commands are similar, but for the main "permanent" storage areas (persisting across parses of segments). Their default values are respectively 1.5M and 2M.

The most esoteric command described here is

```
-prunedelta P
```

where *P* is a real number or integer. This sets an internal field `prunedelta` to *P*. *X***SG** does parse space pruning during parsing (see the flag `prune` described in Section 9), to clear away unlikely partial analyses. The pruning is most vigorous when `prunedelta` is 0, and this is the default for most languages. When `prunedelta` is set higher, pruning is less vigorous and more parses are produced, but at the cost of efficiency (more space required and more time per parse).

4 Options for parse tree display

The default form for parse tree display has been described in Section 2. Parse trees can be displayed also in other forms. If you type

```
+deptree 0.
```

then you will see trees in a different kind of indented form, like so, for the sentence *John sees Mary*.

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

Typing either of the following gives the default form:

```
+deptree.
+deptree 1.
```

Typing `-deptree` is equivalent to `+deptree 0`. If you type this:

```
+deptree 2.
```

then you will see trees in an XML format that may be convenient to use for some interfaces, like so:

```
<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 m gname sname">
      <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 f gname">
      <hd w="Mary" c="Mary" s="Mary1" a=""/>
   </ph>
</ph>
</seg>
```

These are indented for easier readability. If you type

```
+deptree 3.
```

then the trees will still be in XML, but with no indentation whitespace, like so:

```
<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 m gname sname">
<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 f gname">
<hd w="Mary" c="Mary" s="Mary1" a=""/>
</ph>
</ph>
</seg>
```

5 Processing a file of sentences

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. (The `-dofile` keyword can also be preceded by the types of command-line arguments described at the end of Section 3.)

`xsg -dofile` *InFile OutFile*

The *InFile* argument 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 also give `xsg -dofile` optional arguments of the following forms, appearing *after* the `-dofile` keyword.

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

to turn a flag on or off or to set subject areas for the run. Available SG flags are described in Section 9. If you want to set a flag to a value *Val*, you can use an argument pair:

`-on "`*Flag Val*`"`

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"
```

6 The `ldo` interface

There is an editor-based interactive interface to *X***SG** which is useful in trying out variations of sentences when developing an application. It is called `ldo`. It works on Linux/Unix and Windows when Vim is the editor, and on Windows when Kedit is the editor. 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 `g` for Vim or F6 for Kedit.. Then *X***SG** will parse that sentence and show the results in `sg.out`. Then when you press `e` for Vim or F3 for Kedit, 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 `g` (Vim) or F6 (Kedit) 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).

When you are editing *File* in `ldo` mode, you can press `e` (Vim) or F3 (Kedit) 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. This is very useful for experimenting with variations in the form of the sentence. 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 `e` or 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.

7 Features

The features that appear in a Slot Grammar parse tree can be divided roughly into *grammatical* (or *morphosyntactic*) features and *semantic* features. In our formal data structure for a Slot Grammar phrase (which we describe in Section 12 below), all the features (grammatical or semantic) for each phrase are given in a single undifferentiated list of strings attached to the phrase. A phrase always has a head word, and a feature may refer only to the head word, or to the whole phrase.

Grammatical features may represent morphological characteristics of (head) words, or may have more to do with the syntax of the whole phrase. An example of the former is `vpast`, for a past tense verb; an example of the latter is `vsubj`, which means that a verb phrase has an overt subject (in that same phrase).

Semantic features (also called *semantic types* or *concepts*) are names for sets of things in the world, and they are organized into an ontology, which contains not only the set of types, but also a specification of the subset relation between the types:

$$Type_1 \subset Type_2$$

For instance the set `h` of humans is a subset of the set `liv` of living beings: `h` $\subset$ `liv`. These "things in the world" can include events, states, etc., and so for example an event referred to by a verb sense could have a semantic type. It is most typical (and useful) to mark semantic types on noun senses, but they could be marked on any part of speech.

There are two ways of dealing with semantic types in *X***SG**. One is to specify them in an *ontology lexicon*, which should be named `ont.lx`. For the form of ont.lx, see [19]. Currently the non-English *X***SG**'s use an ont.lx, and **ESG** can also. But the latest version of **ESG** uses a second method: The most basic types are encoded directly in **ESG** and its base lexicon; and then we use a method, described in Section 16 below, based on WordNet to augment the type marking.

For the sake of efficiency, in the current implementation the SG grammatical features, and some of the most basic semantic types, are represented internally during parsing by bit strings (or mappings onto bit positions). It is possible to do this because the grammatical features form a closed class. However, most semantic types are represented internally essentially as strings. In spite of the difference in internal (parse-time) representation of features, the SG API data structure for phrases, described in Section 12, represents both types of features as strings. This is done for the sake of simplicity of the API.

In the remainder of this section we first describe the grammatical features used in *X***SG**, and then describe the most basic semantic types used for **ESG**.

We try to use as uniform a set of grammatical features across the different languages as possible. Of course some features are not applicable to all the

languages. The SG system basically works with the union of all the possible features, and some of them just do not get used in all the *X***SG**.

As mentioned above, the part of speech feature comes first in the list of features of a phrase. The possible parts of speech are these:

```
noun, verb, adj, adv, det, prep, subconj,
conj, qual, subinf, infto, forto, thatconj,
special, incomplete
```

The first six of these have obvious meanings. Let us go over the others:

`subconj`. Subordinate conjunction, like *if*, *after*, ...

`conj`. Coordinate conjunction, like *and*, *or*, ...

`qual`. Qualifier, like *very*, *even*. Can modify adverbs and adjectives, not verbs.

`subinf`. For multiwords like *in order to* and *so as to* that act like preinfinitve *to* for infinitive verbs.

`infto`. The preinfinitive sense of *to*, or analogs in the other languages.

`forto`. The sense of *for* in *for-to* constructions like *for John to be there*.

`thatconj`. The subordinate conjunction sense of *that* (or *que, che, daß*).

`special`. A catch-all category for special "words" like apostrophe-s.

`incomplete`. Used as the part-of-speech category for the top node of an incomplete parse.

Now let us look at the other possible features of a phrase. We organize these mainly by part of speech of the phrase.

Shared features. There are some features that are shared across several parts of speech. These include: `sg` (singular), `pl` (plural), `sgpl` (singular and plural), `dual` (dual), `cord` (coordinated), `wh`, and `whnom`. The last is used on determiners, nouns, and verbs; its ultimate purpose is to mark clauses, like *what you see*, that can be used essentially anywhere an NP can.

Verb features. We list these in alphabetical order.

`badvenadj`. A past participle of such a verb does not easily fill the `nadj` slot. Example: *said*.

`gerund`. This gets marked on an *ing*-verb that has become a gerund, as in *the barking of the dogs*.

`ingprep`. A verb whose *ing* form behaves like a preposition. Example: *concern*.

`invertv.` A verb (like *arise* or *come*) that allows its subject on the right and a `comp` PP or *there* left modifier.

`npref.` Prefers to modify nouns over verbs, as head of non-finite VP.

`objpref.` Verb preferring NP object over finite clause or *that*-complement.

`q.` Question clause.

`relvp.` Relative clause.

`se.` German verb that takes *sein* for the present perfect.

`sta.` Stative verb.

`thatcpref.` Verb allowing both finite VP or *that*-complement but preferring the latter.

`transalt.` Verb (like *increase*) allowing transitivity alternation. The theme (the entity undergoing change) can be either the direct object or the subject (when no direct object is given).

`vcond.` Conditional mood.

`vdep.` Dependent clause.

`ven.` Past participle.

`vfin.` Finite verb.

`vfinf.` German infinitive verb modified by *zu*.

`vfut.` Future tense (for Latin languages).

`vimperf.` Imperfect (for Latin languages).

`vimpr.` Imperative VP.

`vind.` Indicative mood.

`vindep.` Indepedent clause.

`vinf.` Infinitive.

`ving.` Present participle.

`vlast.` VP whose last modifier is another VP.

`vobjc.` (For **GSG**) a verb with an overtly filled `obj` or `iobj`.

`vobjnom.` (For **GSG**) a verb with a nominative overtly filled `obj`.

`vpass.` Passive `ven` verb, as in *he was taken*.

`vpast.` Past tense.

`vpers1.` First person.

`vpers2.` Second person.

`vpers3.` Third person.

`vpluperf.` Pluperfect (for Latin languages).

`vpref.` Prefers to modify verbs over nouns, as head of non-finite VP.

`vpres.` Present tense.

`vrelv.` Verb that easily allows a relative clause modifier of its subject to right-extraposed.

`vsbjnc.` Subjunctive.

`vsubj.` VP with overtly filled subject.

`vthat.` Relative clause with *that* as the relative pronoun.

`whever.` A clause like *whatever you see* or *whichever road you take* that is modified by an extraposed `wh-ever` NP.

`postcomp.` Means that the verb cannot have a `comp` slot filler preceding a filler of `(obj n)`.

Noun features.

`acc.` Accusative.

`advnoun.` A noun that can behave adverbially. Main examples: time nouns and locative nouns.

`cn.` Common noun.

`cpropn.` Proper noun, like “Dane” or “Italian”, that is like a common noun in denoting a class.

`dat.` Dative.

`def.` Definite pronoun.

`detr.` Noun requiring a determiner when it (the noun) is singular.

`encprn.` Can be an enclitic (for the Latin languages).

`f.` Feminine.

`gen.` Genitive.

`glom.` A multiword noun obtained by agglomerating certain capitalized nouns in sequence.

`goodap`. Can easily be a right conjunct in comma coordination even though it itself is not coordinated.

`hplmod`. Noun that allows a plural nnoun modifier.

`indef`. Indefinite pronoun.

`iobjprn`. Can be non-clitic iobj (Latin languages).

`lmeas`. Linear measure.

`m`. Masculine.

`meas`. Measure.

`mf`. Masculine or feminine.

`mo`. Month.

`nadjpn`. Pronoun that can fill `nadj` and must agree with head noun (Latin languages).

`nadvn`. Noun that can fill `nadv` and must agree with head noun (Latin languages).

`nom`. Nominative.

`nonn`. A noun that cannot have an `nnoun` modifier.

`notnnoun`. A noun that cannot be an `nnoun` modifier.

`npremod`. Can be a noun premodifier of a noun even though it is also an adjective.

`nt`. Neuter.

`num`. A number noun.

`objpprn`. Can be object of preposition (Latin languages).

`objprn`. Can be non-clitic obj (Latin languages).

`oreflprn`. Can be use *only* as a reflexive (Latin languages).

`percent`. A percent number noun.

`perspron`. Personal pronoun.

`pers1`. First person.

`pers2`. Second person.

`pers3`. Third person.

`plmod`. Noun that can be an `nnoun` even when it is plural.

`posit.` Position (like *middle* or *end*).

`poss.` Possesive pronoun.

`procprn.` Can be proclitic (Latin languages).

`pron.` Pronoun.

`propn.` Proper noun.

`quantn.` Denotes a quantity (like *all* or *half*).

`reflprn.` Reflexive pronoun.

`relnp.` An NP that can be the relativizer of a relative clause.

`tonoun.` A noun, like *school*, that can by itself (without premodifiers) be the `objprep` of *to*, even though it is also a verb.

`uif.` Uninflected.

`way.` Manner/way.

`whevern.` An NP like *whatever* or *whichever road*.

Adjective features.

`adjnoun.` Adjectives like *poor* that can have a *the* premodifier and act like the head of an NP.

`adjpass.` An adjective like *delighted* that is also a past participle, but the past participle is not allowed to fill `pred(en)`.

`aqual.` (For German) an adjective, like *denkbar* that can premodify an adverb (filling `advpre`).

`compar.` Comparative.

`detadj.` Adjectives like *next* and *last* that have an implicit definite article.

`erest.` Can use *-er* and *-est* for comparative and superlative. Applies to adverbs too.

`lmeasadj.` Adjective like *high* allowing constructions like *three feet* <u>*high*</u>.

`noadv.` (For German) an adjective that cannot be used as an adverb.

`noattrib.` Not allowed as filler of `nadj`.

`nocompa.` Not allowed as filler of `comp(a)`.

`nocompare.` Not allowing comparison (for Latin languages).

`nopred`. Not allowed as filler of `pred`.

`nqual`. Adjectives like *medium* that allow constructions like *a medium quality car*.

`post`. Adjective like *available* that can easily postmodify a noun, as in *the first car available*.

`soadjp`. Gets put by the grammar on an adjective phrase like *so good* to allow that phrase to fill `nadv` in an NP like *so good a person*. This is done when the adjective (like *good*) is premodified by a qualifier marked `soqual`.

`superl`. Superlative.

`tmadj`. A time adjective like *early*.

`toadj`. Similar to `tonoun`.

Adverb features.

`badadjmod`. Preferred not to modify adjective.

`compar`. Comparative.

`detadv`. Can modify a determiner.

`interj`. An interjection.

`initialmod`. Modifies on left only as first modifier.

`introadv`. Adverb like *hello* that easily left-coordinates by comma-coordination.

`invertadv`. Adverb that allows certain constructions *Adv Verb Subj*.

`loadv`. Easily modifies a locational prep or adverb (particle).

`locadv`. Locative adverb like *above*.

`noadvpre`. Cannot have (qualifier) premodifier.

`nopadv`. Cannot modify preposition.

`notadjmod`. Cannot modify an adjective.

`notinitialmod`. Cannot appear clause-initially.

`notleftmod`. Cannot modify verb on left.

`notnadv`. Cannot premodify a noun.

`notrightmod`. Cannot modify verb on right.

`nounadv`. Can modify noun.

`nperadv`. Can fill `nper` slot for nouns, like *apiece* and *each*.

`npost`. Can postmodify a noun in slot `nadjp`.

`partf`. Adverb that can be a particle (also applies to prepositions that can be particles).

`post`. Can postmodify (like *enough*).

`ppadv`. Can modify preposition (with no penalty).

`prefadv`. Adverb analysis as `vadv` is preferred over noun analysis as `obj`.

`reladv`. For Spanish, an adverb like *cuando* that can create a relative clause.

`superl`. Superlative.

`thereprep`. Adverb like *thereafter, thereof, ...*.

`tmadv`. Time adverb like *before, early, ...*.

`vpost`. Cannot modify a finite verb on the left.

Determiner features.

`all`. Only for the determiner *all*.

`ingdet`. Marked on `possdet`s and *the*. Can premodify present participle verbs.

`possdet`. Possessive pronoun as determiner.

`prefdet`. Preferred as determiner (over other parts of speech).

`reldet`. For Spanish, a determiner like *cuyo* that can create an NP serving as relativizer.

`the`. Marked only on *the*.

Qualifier features.

`badattrib`. If it modifies an adjective, then that adjective cannot fill `nadj` slot.

`c`. Modifies only comparative adverbs.

`post`. Can postmodify.

`pre`. Can premodify (the default).

`soqual`. See `soadjp` for adjectives above.

Subordinate conjunction (`subconj`) features.

`assc`. For *as* (or analogs in other languages).

`comparsc`. For `assc` or `thansc`.

`finsc`. Allows only finite clause complements (filling `sccomp`).

`notleftsc`. A clause with this as head cannot left-modify a clause (as `vsubconj`). Example: *for*.

`okadjsc`. Allows adjective complement.

`oknounsc`. Allows noun complement.

`oknsubconj`. Can fill `nsubconj`.

`poorsubconj`. Preferred not as `subconj`.

`sbjncsc`. For the Latin languages: Suppose a `subjconj` S has a finite clause complement C (C is filler of `sccomp`). Then S must be marked `bjncsc` if C is marked `vsbjnc` (subjunctive) but not `vind` (indicative). And if S is marked `sbjncsc` and C is marked `vsbjc`, then remove `vind` from C if it is present.

`thansc`. For *than* (or analogs in other languages).

`tosc`. Allows `infto` complement.

`whsc`. A wh-`subconj` (like *whether*).

Preposition features.

`accobj`. (For German) allows the `objprep` to be accusative.

`adjobj`. (For German) allows the `objprep` to be an adjective or past participle phrase.

`asprep`. For *as* and analogs in other languages.

`badobjping`. Cannot have a `ving` `objprep` (under certain conditions).

`daprep`. For German. For PPs like *dabei* and *darauf*. A word *da+Prep* is unfolded to a PP with head *Prep* marked `daprep` and `objprep` filled by *es*.

`datobj`. (For German) allows the `objprep` to be dative.

`genobj`. (For German) allows the `objprep` to be genitive.

`hasadjobj`. (For German) the `objprep` is an adjective or past participle phrase.

`infobj`. (For Latin languages) has an `objprep` that is an infinitive VP.

`locmp`. Used for multiword prepositions that fill `comp(lo)`.

`motionp`. Prefers not to fill `comp` if the matrix verb is marked `sta`.

`nonlocp`. Cannot be a filler of `comp(lo)`.

`notwhpp`. Cannot have wh `objprep` (under certain conditions).

`pobjp`. The `objprep` can be a PP itself. Example: *from*.

`ppost`. The preposition can follow the `objprep`. If the preposition is marked `ppost` but not `ppre`, then the preposition *must* be on the right.

`ppre`. The preposition can precede the `objprep`. There is no need to mark this feature on the preposition, in order to allow the preposition to be on the left, unless the preposition is marked `ppost`.

`pprefn`. Prefers to modify nouns.

`pprefv`. Prefers to modify verbs.

`preflprn`. Marked by the grammar on a PP when the `objprep` is a reflexive pronoun.

`relpp`. A PP that can serve as a relativizer for a relative clause.

`staticp`. Preposition like *in* or *on* that is used normally to represent static location vs. goal of motion (as with *into* and *onto*).

`timep`. Common modifier of time nouns, as in *two days* <u>*after*</u> *the meeting*.

`timepp`. For certain PPs where the `objprep` is a time noun.

`woprep`. Like `daprep`, but with *wo* instead of *da*.

Basic semantic types.

Here we list and briefly describe the most basic semantic types used with **ESG**. They are listed in alphabetical order. Most of the types are marked on nouns, and just a few (named specifically) are marked on verbs.

`abst`. Abstraction.

`act`. Act by a human.

`ameas`. Unit of area measure.

`anml`. Animal (not including humans).

`artcomp`. Artistic composition – literary, musical, visual, dramatic, etc.

`artf`. Artifact.

`century.` Named century like "300 B.C."

`chng.` Event of change.

`cmeas.` Unit of volume (cubic) measure.

`cognsa.` Cognitive state or activity.

`coll.` Collection.

`collectn.` Collective noun dealing overtly with a set – like "set", "group", "family". Subtype of `coll`.

`cpropn.` Meant to suggest "common proper noun". A propn, like "German" that can name a class, and so behaves also like a common noun.

`cst.` State or province in a country.

`ctitle.` A postposed company title, like "Inc." or "Co.".

`ctry.` Named country (propn).

`cty.` Named city.

`discipline.` Branch of knowledge.

`doc.` Document.

`dy.` Day. Named weekdays like Tuesday or named holiday days.

`emeas.` Unit of electromagnetic measure.

`evnt.` Event.

`f.` Female.

`feeling.` Feeling, emotion.

`geoarea.` Geographical area.

`geoform.` Geological formation.

`geopol.` Named geopolitical unit – like ctry, cst, cty.

`gname.` Human given name.

`h.` Human individual.

`hg.` Human group.

`imeas.` Unit of illumination measure.

`inst.` Instrument – artifact used for some end.

`langunit.` Language unit, like word, discourse, etc.

`liv.` Living thing.

`lmeas.` Unit of linear measure..

`loc.` Pronoun for location, like “here”, “anywhere” or certain general locational common nouns like “outside”.

`location.` Point or region in space.

`locn.` Named location – like a geopol or an ocean.

`locnoun.` Noun like “here” or “east” that can function adverbially or fill comp(lo).

`m.` Male.

`massn.` Mass noun.

`meas.` Unit of measure.

`mmeas.` Unit of money.

`mo.` Named month, like “October”.

`name.` Usual sense.

`natent.` Natural entity (not made by humans).

`natlang.` Named natural language.

`natphenom.` Natural phenomenon.

`ntitle.` A postposed part of a human name like “Jr.” or “Sr.”.

`org.` A human organization. Subtype of hg.

`percent.` Percent expression.

`physobj.` Physical object.

`physphenom.` Physical phenomenon.

`process.` Sequence of events of change.

`professional.` Person with a profession requiring higher education.

`propcn.` A common noun, like “society” or “mathematics”, that can name a single entity and function like a propn – sometimes capitalized, especially in older writing.

`property.` (Synonym of “attribute”).

`ptitle.` A postposed human title, like “Ph.D.” or “M.D.”

`quantn.` Quantity pronoun like “more”, “half”, or ordinal number, or collectn noun.

`relig.` A named religion.

`rlent.` Role entity. An entity (usually person) viewed as having a particular role – includes many human roles like leader, artist, engineer, father, ...

`rmeas.` Relation between measures, like rate or scale.

`saying.` Expression or locution.

`sayv.` Verb of saying.

`sbst.` Substance.

`smeas.` Unit of sound volume measure.

`sname.` Human surname.

`socialevent.` Human social event.

`speechact.` Speech act, e.g. request, command, promise.

`strct.` Structure (thing constructed).

`title.` Human title, like “President” or “Professor”.

`tm.` Time noun, like “year” or “yesterday”.

`tma.` Time noun, usually a pron or propn, or behaving as such – e.g. “now”, “tomorrow”.

`tmdetr.` Time noun, like “season”, that requires a determiner in order to act adverbially.

`tmeas.` Unit of temperature measure.

`tmperiod.` Time period.

`tmrel.` Time noun allowing certain finite clauses as `nrel` modifiers.

`trait.` Distinguishing property (usually applied to persons).

`transaction.` Act in conducting business.

`ust.` U.S. state. ¡ cst

`vchng.` Change (verb type).

`vchngmag.` Change magnitude/size (verb type).

`vcreate.` Cause sth to exist or become (verb type).

`vmove.` Change locations (verb type).

`wlocn`. Named water mass, like "Atlantic Ocean".

`wmeas`. Unit of weight measure.

`yr`. Year. Specific year, usually given numerically.

8 Slots and slot options

The SG parse tree shows for each phrase (or tree node) the slot that is filled by the phrase. Slots are of two kinds – *complement* slots and *adjunct* slots. As indicated in Section 4, the complement slots correspond to logical arguments of the head word sense of the phrase; they are specified in the lexical entry for the word sense. The possible complement slots for verbs are these six:

`subj`	*subject*
`obj`	*direct object*
`iobj`	*indirect object*
`pred`	*predicate complement*
`auxcomp`	*auxiliary complement*
`comp`	*complement*

We discuss these in more detail below.

The fillers of adjunct slots are like "outer modifiers". In logical form, they often predicate on the logical form of the phrase that they modify. For instance the determiner slot `ndet` for an NP may contain a quantifier that is like a higher-order predicate on the rest of the NP. (See Section 17 for a description of ESG-based logical forms, with a treatment of quantifiers.) Adjunct slots are associated with parts of speech, and are specified in the syntactic component of *X*__SG__.

An SG phrase also shows the *slot option* chosen for a given slot. Slot options correspond roughly to the phrase category (part of speech) chosen for the filler of the slot. A given slot may be filled by phrases of various categories. For instance the `iobj` slot in English can normally be filled by either an NP or a *to-PP*, as in these two examples:

> *Mary gave John the book.*
> *Mary gave the book to John.*

The lexical entry for a word sense shows, for each of its complement slots, what slot options are allowed for that slot. The semantic idea is that (as mentioned) the slot corresponds to a logical argument, but the list of options for the slot indicates how the argument can be realized syntactically. (The slot itself also carries syntactic information though.) The current possible SG slot options are these:

```
a, agent, aj, av, bfin, binf, dt, en, ena,
```

```
fin, fina, finq, finv, ft, ger, gn,
impr, inf, ing, io, it, itinf, itthatc, itwh,
n, na, nen, nmeas, nop, nummeas, padj, pinf, pinfd,
prflx, prop, pthatc, pwh, qt, rflx, sc, so, thatc, v, whn
```

We will not describe these here because usually (though not always) the feature list of the phrase contains the information provided by the slot option – especially since the feature list includes the part of speech of the phrase. But see [19] for a description of the main options. Generally, one should think of slot options as providing control for parsing itself. There is one case at least where it is important to look at the slot option: The `comp` slot for verbs can have an `n` option, fillable by NPs, for object complements like this:

$$\textit{They elected } \underbrace{\textit{Ellen}}_{\texttt{obj(n)}} \ \underbrace{\textit{president of the company.}}_{\texttt{comp(n)}}$$

But `comp` also allows an option `io`, fillable by NPs, for an alternative form of the indirect object, like this:

$$\textit{They took } \underbrace{\textit{John}}_{\texttt{comp(io)}} \ \underbrace{\textit{the contract}}_{\texttt{obj(n)}}.$$

Since in both cases `comp` is filled by an NP – even a human NP – it is worth looking at the distinction of options `n` vs. `io`.

Now let us look at the SG slots. We organize the description by part of speech. It is worth noting that most slots – especially complement slots – allow several options. In some of the description, we will be a bit imprecise by not distinguishing between a slot and the filler of the slot. For instance, in a statement like "The `subj` agrees with the finite verb", we really mean: "The filler of `subj` agrees with the finite verb".

Verb complement slots. We will elucidate these in part in terms of the thematic roles THEME, LOCATION, GOAL, and AGENT, in the sense of Gruber [6] and Jackendoff [9]. In verbs that describe a change, the THEME is what changes, and it changes to the GOAL state. The AGENT (if present) is the participant that carries out the change. In verbs that describe a state, the THEME is what the state is predicated of and the LOCATION is that state.

`subj`. Most readers of this document will be familiar with the subject slot, but let us make a few comments. If a verb has an AGENT, then this will normally fill `subj`. The subject may also be THEME. The default position of the subject in our European languages is before the verb in an active declarative clause. The subject is filled overtly (in these languages) only in finite clauses, and then it must agree in person and number with the finite verb. In non-finite VPs, we often show the logical subject in the word sense predication. The `subj` slot may be filled by several different categories of phrases besides noun phrases, as in:

> *Seeing him was difficult.*
> *That he did that is amazing.*
> *Whether he can go is an open question.*

`obj`. If a verb has a THEME and a filled `obj`, then that `obj` is normally the THEME. The `obj` may be filled by several different categories of phrases, as in:

> *He believes John's story.*
> *He believes in John.*
> *He believes that John was there.*
> *He believes what John said.*
> *John said, "I was there".*
> *He likes seeing John.*
> *He believes John to be honest.*

`iobj`. The indirect object normally has a GOAL role. In English, it can most typically be filled by both an *NP* and a *to-PP*. But some verbs, like *ensure*, can take only an *NP* `iobj`. And others, like *confess*, take only a *to-PP* `iobj`. The `iobj` sometimes allows other prepositions, as in:

> *She bought some books for me.*
> *She bought me some books.*

An *NP* `iobj` normally takes the dative case in our European languages.

`comp`. In the thematic roles, the basic idea of `comp` is that it is the GOAL in verbs of change and the LOCATION in verbs that describe a state. The `iobj` slot could actually be folded into the `comp` slot, and in fact the option `io` for `comp` is more or less an alternative way of expressing `iobj(n)`, as mentioned above. But we use `iobj`, partly for the sake of familiarity. As with the other verb complement slots, `comp` allows several options. We gave *NP* examples above. The most typical `comp` is a *PP*. Examples:

> *Alice drove* *Betty* [`obj(n)`] *to the store* [`comp(lo)`].
> *Alice drove* *Betty* [`obj(n)`] *to distraction* [`comp(lo)`].
> *Alice drove* *Betty* [`obj(n)`] *crazy* [`comp(a)`] .

`pred`. In English, `pred` is a slot only for the verb *be*. It can be filled by a very wide range of categories of phrases, illustrated in part by the following:

> *Bob is a teacher.*
> *Bob is happy.*

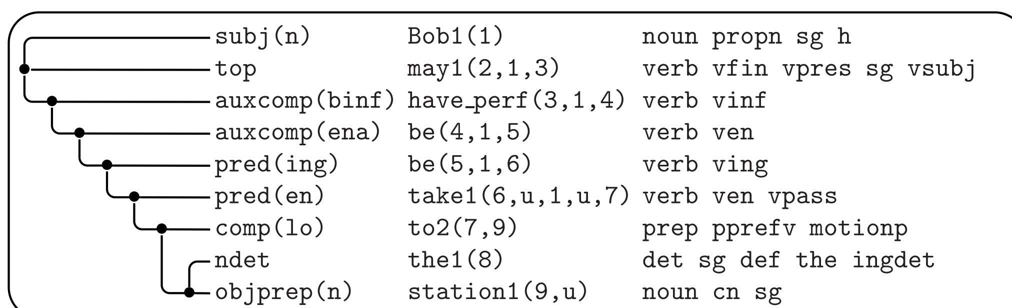


Figure 5: *Bob may have been being taken to the station.*

Bob is in love.
Bob is to leave tomorrow.
Bob is leaving tomorrow.
Bob was taken to the station.

In our other European languages, `pred` is used for some more verbs besides the analogs of *be*; for instance for German it is used for *werden* as well as *sein*.

`auxcomp`. This slot is used in English mainly for the modal verbs, where `auxcomp` is filled by a bare infinitive, for the auxiliary *do*, which also takes a bare infinitive, and for the perfect sense of *have*, where the filler is an active past participle. Two of these cases are illustrated in Figure 5, which also contains the use of `pred` for the progressive and the passive.

Verb adjunct slots. The names of most of these slots begin with "v". For most of these, instead of giving a general definition, we will just give one or more examples.

`vadv`. *He always likes chocolate.*

`vprep`. *In that case, please buy some chocolate.*

`vsubconj`. *If the shoe fits, wear it.*

`vnfvp`. The `nfvp` stands for "non-finite verb phrase".

Using the mouse, select the chocolate icon.
Made in Switzerland, this chocolate is sold in many countries.
To select the chocolate icon, use the mouse.

`vfvp`. The `fvp` stands for "finite verb phrase".

However you do it, select the chocolate icon.
Had you selected this kind of chocolate, you would be happier.

`vforto`. *For John to be there on time, we'll have to call.*

`vcomment`. *This chocolate, I think, is best.*

`vadjp`. *Happy with the results, she left.*

`vvoc`. *John, can you hear me?*

`vadj`. *happy-seeming*

`vdet`. *his seeing the car*

`vnoun`. *user-defined*

`vrel`. *Someone appeared who I didn't expect.*

`vextra`. *It was assumed that she would arrive Tuesday.*

`vpreinf`. *zu sehen*

`vnp`. *Votre ami, aime-t-il le chocolat?*

`vinfp`. *Trouver ce chocolat, c'est très difficile.*

`vdat`. *Mi mangio una mela.*

`vnthatc`. *Er hat keine Erkenntnis gehabt, daß sie da war.*

`whadv`. *When did you eat?*

`whprep`. *At which table did you sit?*

Noun complement slots.

There are four of these. The most important are:

```
nsubj, nobj, ncomp
```

For deverbal nouns, these slots correspond to the verb slots:

```
subj, obj, comp
```

– allowing for the view that `iobj` is a kind of special case of `comp`. These noun slots have roughly the same range of possible slot options as the corresponding verb slots do. But the `n` option for the noun slots is filled by an *of*-PP instead of an NP.

For example we have the correspondence:

John (`subj(n)`) *knows* *that Bill was there* (`obj(thatc)`)

John's (`nsubj(n)`) *knowledge* *that Bill was there* (`nobj(thatc)`)

Filling of the `nsubj` slot by *John's* here is seen only in the deep structure, in the predicate arguments of *knowledge*. In the surface structure, *John's* fills the adjunct slot `ndet` of *knowledge*.

The `nsubj` slot can be filled directly in postmodification by *by*-NPs, *of*-NPs, or *from*-NPs, depending on the head noun and the modifier context.

There are relational nouns that are not deverbal, but still have some of the slots `nsubj`, `nobj`, `ncomp`. Example:

A poem $\underbrace{\textit{by Smith}}_{\texttt{nsubj(agent)}}$ $\underbrace{\textit{about nature}}_{\texttt{ncomp(p)}}$

There are many nouns that have only the slot `nobj`, typically with option `n`, for example in:

Department $\underbrace{\textit{of Mathematics}}_{\texttt{nobj(n)}}$

The last noun complement slot is `nid` ("noun identifier"), which is illustrated in `nid` slots for *page* and *Appendix*, in:

It appears on page $\underbrace{\textit{235}}_{\texttt{nid(n)}}$ *in Appendix* $\underbrace{\textit{B}}_{\texttt{nid(n)}}$.

Noun adjunct slots. The names of these all begin with "`n`". For English, the first four premodify the noun, and the remaining postmodify.

`ndet.` <u>*the*</u> *house*
<u>*her*</u> *house*
<u>*my mother's*</u> *house*

`nadj.` *the* <u>*large*</u> *house*
the <u>*Chicago*</u> *house*
<u>*three*</u> *houses*

`nnoun.` *the* <u>*boat*</u> *house*

`nadv.` <u>*only*</u> *the house*

`nposs.` *John* <u>*'s*</u>

`nadjp.` *anyone* <u>*available for the job*</u>

`nper.` *\$2.00* <u>*per gallon*</u>

`nprop.` *company* <u>*X*</u>

`nappos.`

John, my brother
Paris, the capital of France

`nloc.` *Paris, France*

`nprep.` *the man in the house*

`ngen.` *ein Freund meines Vaters*

`nrel.` *the man that I saw yesterday*

`nrela.` *the man I saw yesterday*

`nnfvp.` *anyone seeing John*
anyone seen by John
the first person to see John

`nsubconj.` *the situation as described by John*

`ncompar.` *more money than he has*

`nsothat.` *such a good speaker that he gets many invitations*

Adjective complement slots. There are two. The main one is `aobj`. Examples:

happy with the result
happy to see the result
happy that the result was good

The second one is `asubj` and is filled by a noun that the adjective modifies.

Adjective adjunct slots.

`adjpre.` *very happy*

`anoun.` *user-friendly*

`adjpost.` *happy enough*

`acompar.` *happier than John*

`aprep.` *red in the face*

`asothat.` *so red that he glowed*

`arel.` This is like the noun adjunct `nrel` but is used for `adjnoun` adjectives like “poor”, “rich”, “latter”, that can be used like nouns when modified by “the”, as in: *the latter, which was discussed in the preceding section*

`anfvp.` Similar to `arel`, but is analogous to `nnfvp` as in:
the latter, discussed in the preceding section

Adverb complement slots. There are two. The main one is `avobj`. Examples:

> *enough for me*
> *enough to accomplish the result*
> *enough that the result was good*

The second one is `avsubj` and is filled by a verb that the adverb modifies.

Adverb adjunct slots.

`advpre`. *very quickly*

`advpost`. *quickly enough*

`advinf`. *how to do it*

`avsothat`. *so quickly that it smoked*

`avcompar`. *more happily than John*

Preposition complement slots: There are two. The main one is `objprep`. Examples:

> *in the house*
> *in seeing the house*
> *in what you see*

The second one is `psubj`, and is filled by the noun or verb modified by the PP headed by the preposition.

Preposition adjunct slots.

`padv`. *out in the yard*
expressly for John

`pvapp`. *daran, daß er geht*

Determinar adjunct slot: `dadv`. Example: *almost all people*

Complement slot for `subconj`: `sccomp`. Example: *if the shoe fits*

Adjunct slot for `subconj`: `scadv`. Example: *only while she eats*

Complement slot for `infto`: `tocomp`. Example: *to eat chocolate*

Adjunct slot for `infto`: `toadv`. Example: *only to eat*

Complement slots for `forto`.

`forsubj.` *for John to write the book*

`forcomp.` *for John to write the book*

Complement slot for `thatconj`: `thatcomp`. Example: *that he wants to go*

Complement slot for `subinf`: `subinfcomp`. Example: *in order to see the show*

9 Flags

The SG system has several *flags* that can be set to control various aspects of parsing. Each flag has an integer value, which is normally either 1 (flag is on) or 0 (flag is off), but some flags use other integer values. For instance the flag `deptree`, discussed above, can have values 0, 1, 2, or 3. Default values for all flags are set by *X***SG** at initialization time.

To set a flag F on or off in interactive mode, you can type `+`F or `-`F respectively. Or you can type `+`F n to set the value to any non-negative integer n. (So `-`F is equivalent to `+`F 0.)

You can also set flags when you compile your own *X***SG** application, by using the API function `sgSetflag`, described on page 67 in Section 14.

Here is a list (in alphabetical order) of the most useful flags and their meanings. We will usually explain what the effect of having the flag *on* is. If the description includes a phrase like "if this flag is turned on", then this means that it is off by default. Some of the flags, especially ones that trace the parsing process, have effect only when *X***SG** is compiled in a way that allows more tracing.

`adjustchunks.` When chunk-lexical processing is being done (see Subsection 11.4), this enables **ESG** to apply distribution in coordinated chunks – so that e.g. "neck and back pain" shows like "neck pain and back pain".

`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 hook functions.

`allcapprop.` The general idea of `allcapprop` is that a word that consists of all capital letters will be treated as a proper noun. But more precisely it is this: Suppose (1) the flag is on, (2) W is a word consisting of all capital letters and having at least two letters, and (3) the containing segment does not consist totally of capital letters. Then the system will use only W itself in morphology and lexical look-up; it will *not* do morphology and look-up on the lower-case form of W. The latter action is the default. For instance, if the input segment contains "MAN", the system will look up both "MAN" and "man". If `allcapprop` is on (and the segment is not all caps), then it will look up only "MAN". This flag is off by default.

`allcappropl`. This has the following effect. Suppose (1) the flag is on, (2) W is a word consisting of all capital letters and having at least two letters, (3) the containing segment does not consist totally of capital letters, and (4) there is no lexical analysis of W itself. Then the system will create a proper noun analysis for W. And, if `allcapprop` is off, the system will add any lexical analyses it can obtain for the decapitalized form of W. This flag is off by default.

`allmorph`. For **ESG** (English *X***SG**), morphological analysis is not done on a word that appears directly in the lexicon. But if this flag is turned on, such analysis is done, as well as returning the analysis from the lexical entry.

`allnodes`. If this is turned on, then the tree lines in the default parse display show a circle on each nodes's treeline.

`aotrace`. When this is turned on, the effects of affix operations in morphology are shown.

`capng`. This causes capitalized noun groups to be agglomerated into multi-word proper nouns. The noun group can contain (capitalized) common nouns, proper nouns, or adjectives. The agglomeration will not take place unless the segment is an *lcseg*; this means that the segment contains some "content words" (nouns or verbs) in lower case. The flag `capng` is off by default.

`capprop`. This is similar to `capng` but does not produce as many proper noun agglomerations. If the segment is an *lcseg* then it agglomerates sequences of capitalized words that have some proper noun analysis. Else it agglomerates only capitalized words that have a *unique* analysis and this analysis is a proper noun. This flag is on by default.

`captoprop`. This is the strongest of schemes for turning capitalized words into proper nouns. The basic idea is that every capitalized word W becomes a proper noun, but there are several exceptions, as follows: (1) The segment is not an *lcseg*; (2) W is the first word of the sentence, or the first word after a quote; (3) W is a pronoun or already has a proper noun analysis; (4) W has one of the semantic types:

```
tma, propcn, meas, title, hlanguage, st_people,
st_religion, st_discipline, mmeas, wmeas, emeas, cmeas
```

This flag is off by default.

`chunkstruct`. See Subsection 11.4. Has same effect as flag `hststruct`.

`colonsep`. Basically this causes colons to be treated as segment terminators. It is on by default. But if in addition the flag `linemode` is on, then the flag `lncolonsep` must be off in order for colons to be treated as terminators.

When colons are not terminators, they will be treated as punctuation conjunctions.

`ctrace.` When this is turned on, the process of coordination analysis in parsing is shown.

`deptree.` Controls type of parse tree display. Discussed above.

`derivmorph.` Enables derivational morphology. On by default.

`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.

`doLDT.` See Section 17.

`doLF.` See Section 17.

`doLFsense.` When this is turned on, the word predicates in LFs are shown in *X***SG** sense form. Otherwise citation forms are used.

`doshowstat.` Show parsing statistics for a file. Output goes at the end of the output file. On by default.

`doUTF8.` Enables processing of UTF-8 texts. Off by default.

`dolangs.` Enables use of lexicon enlang.lx, which has entries like

```
academia < es pt lt
```

showing that the English headword (`academia` in this case) is also a word in the indicated languages. Allows better recognition of (multi-)words in other languages. Off by default.

`dolatrwd.` Enables use of lexical features of the form `(latrwd R)` that reward the headword as a LAT (Lexical Answer Type) by real number `R`. Off by default.

`dopostag.` If this is turned on, *X***SG** does POS tagging of the input sentence, e.g.

```
The[det] cats[cnpl] leapt[vpastpl3] up[prep]
```

Off by default.

`dotime.` Do accounting on times spent on the various parts of the parsing process. 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`. Enables detailed tracing of the parsing process. Off by default.

`forcenp`. This is used in **ESG**. When it is on, only the NP analyses of a segment will be produced, as long as there exists at least one NP analysis. Off by default.

`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, it shows only the strongest one – the one with greatest n which the phrase has as a feature. Similarly for the `ri`n. This flag is off by default.

`glomnotfnd`. Enables agglomeration of adjacent non-found words into multiwords. On by default.

`hstlex`. Enables chunk-lexical processing – see Subection 11.4. Same as `usechunklex`. Off by default.

`hststruct`. See Subsection 11.4. Has same effect as flag `chunkstruct`. Off by default.

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

`hyphenblank`. See Subsection 11.4. Off by default.

`ibmiddoc`. Enables processing of IBMIdDOC documents (used in some IBM manuals). Off by default. The flag `sgml` should also be set on when `ibmiddoc` is on.

`idtext`. Stands for "identificational text". Helps parsing of Jeopardy!-style segments. Special heuristics are used for the occasionally non-standard uses in Jeopardy! questions of "this" and similar words, and also for the greater use of NPs as complete segments.

`ldtsyn`. See Section 17.

`limitall`. Suppose the value of this flag is the integer N. If the flag `all` is on, then only the first N parses of the current segment will be processed (if $N \leq 0$, this means that no parses will be processed). If `all` is off, then exactly the first parse will be processed, no matter what `limitall` is set to. The default value of `limitall` is a "large integer" (like 1000000).

`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.

`lncolonsep.` It is on by default. See flag `colonsep` for the effect of this flag.

`loadlangs.` This flag should be turned on at initialization time in order to load the lexicon `enlang.lx` – see `dolangs` above.

`ltrace.` When this is turned on, *X***SG** will trace the lexical analyses it finds for each (single) word w in the segment, including multiword analyses with w as head.

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

`onlyLDT.` See Section 17. Turning this on has the effect of turing on `doLDT` and `ldtsyn`, but turning off any other parse displays.

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

`penntags.` In parse trees, parts of speech get displayed in Penn Treebank form.

`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. This is implemented in `topeval` by decrementing the `eval` field by an amount that increases with shorter segments. This flag is off by default. It can also be turned on with a value different from 1 (or 0). When it is given a non-zero value, that value will be used as a multiplier in a certain formula that rewards NP analyses. So for example a value of 3 will produce 3 times the reward of a value of 1.

`printinc.` Causes the segments with incomplete parses to be printed out in a file with the same filename as the main output file, but with extension `.inc`.

`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.` When this is on (and it is on by default), the parsing algorithm prunes away (discards) partial analyses whose parse scores become too bad. We will not describe the details of parse scoring in this report, but see [18]. If you turn this flag off, the parser generally produces many more parses. If not enough space is allocated for *X***SG** and `prune` is off, then there may be problems for parsing longer sentences. An alternative to turning `prune` off is to increase the *X***SG** field `prunedelta`, as described in Section 3.

`ptbtree.` When this is turned on, a Penn Treebank (PTB) form of the parse will be displayed. (It uses Cambridge Polish syntax.) If the flag value is 1, then the PTB tree will be displayed on one line. If the value is 2, the tree will be displayed in a nice indented form.

`ptbtreeF.` If this is on, the PTB tree will be added (as a string) to the features of the top node of the parse tree. This allows a way of communicating the PTB tree through the existing *X***SG** API.

`qtnodes.` On by default. Allows quote node analyses – see Subsection 11.3.

`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.

`showLF.` See Section 17. Off by default.

`showaopts.` This is off by default, but when it 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`.

`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.

`showopts.` When this is turned on, the chosen option *o* for each complement slot *s* will be shown in parses in the form *s*(*o*). On by default.

`showposonly.` In parse tree displays, the only features shown are parts of speech.

`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.

`showslots`. Show the available (unfilled complement) 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 of whitespace will break the current segment. Off by default.

`sgsyn`. 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 display of the tokens for a segment that are produced by the tokenizer. See Section 13 for the description of the `token` data type.

`usechunklex`. Enables chunk-lexical processing – see Section 11.4. Same as `hstlex`.

`wstrace`. If turned on, causes morphological analysis structures to be displayed.

`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, except on VM.

This completes the inventory of flags.

10 Tag handling

*X***SG** can read texts in various formats. One of these of course is plain text (without any tagging). In addition, *X***SG** can handle various forms of SGML or XML and the IBM BookMaster document format. The main two forms of SGML known to *X***SG** are HTML and IBMIdDoc. See the flags in Section 9 for switching to these tag systems. Adapting to a new form of SGML is a matter of specifying for *X***SG** certain classes of tags for that DTD (this currently cannot be done directly by the user).

Generally *X***SG** does better with tagged text than with plain text, for at least three reasons:

(1) Sometimes in plain text it is not clear to *X***SG** where segments end, because segments like section titles are usually written without end punctuation. Tags will usually show *X***SG** where such segments end.

(2) In tagged text, tag pairs within segments like those that indicate font changes, italics and boldface are often useful to *X***SG** in delineating subphrases of segments.

(3) Some tags can provide clues to *X***SG** as to the kind of segment that follows the tag. For example, headings or titles are often noun phrases as opposed to sentences, and *X***SG** can pay attention to heading tags like the HTML `<H1>` and parse the following segment with a preference (though not strict requirement) for an NP analysis.

(4) Sometimes parts of documents are not meant to be processed by the parser or NLP system. For example an MT system should not try to translate a Javascript section of an HTML document. In tagged text, such sections are usually delimited by certain tags, which *X***SG** can pay attention to.

11 Multiwords, named entities, and chunks

*X***SG** has several strategies for handling multiwords, named entities, and special kinds of tokenization. We discuss these in this section.

11.1 Normal lexical multiwords

The *X***SG** lexicons contain many multiword entries, which can be of any part of speech. Multiwords always have a *headword*, which can be inflected. The headword is by default the last word in the multiword, but this can be overriden by putting an `=` sign in front of the desired headword, as in this entry for "attorney general":

```
=attorney general < n nobj h title
```

By default, a multiword entry, if it applies, is "irrevocable" in the sense of not allowing other analyses of the string it matches. But if a multiword frame has the feature `sepmw` ("separable multiword"), then the parser is allowed to create alternative analyses. For instance the **ESG** lexicon has a `sepmw` entry for "in that" as a `subconj`, and this allows alternative parsing, as needed for "in that house".

11.2 Structured lexical multiwords

Certain multiwords, which we call *structured multiwords* or *star multiwords*, can exhibit a "have-it-both-ways" scheme, where syntactic structure is shown for the multiword string, but the multiword still behaves basically as a unit, or is "insular", in parsing: Proper subphrases of the multiword cannot modify, or be modified by, phrases outside the multiword.

Example: If "blue sage" has a lexical entry that makes it a star multiword, then for the sentence "We saw some blue sage", we get the parse:

```
subj(n)  we(1)          noun pron pl pers1 nom h perspron
top      see1(2,1,5)    verb vfin vpast pl vsubj thatcpref perceive
ndet     some1(3)       det sg indef prefdet
nadj     blue1(4)       adj erest
obj(n)   sage2(5)       noun cn sg physobj plnt liv (* blue sage)
```

The semantic types `physobj plnt liv` come from type markings on the whole multiword "blue sage", not just the headword "sage". They come ultimately from the WordNet-based lexical augmentation described in Section 16 below, where there is a star multiword entry for "blue sage" with such types marked. And yet the parse shows syntactic structure of "blue sage", helping one to know that blue sage is a kind of sage (in a limited sense of "sage"). But note that the parse also shows the string for the multiword, via the feature `(* blue sage)` of the headword.

The multiwords that can get structured in this way are restricted as follows: They must have exactly one lexical sense, which is a common noun, and the headword is the last word of the multiword. Call these "nouny multiwords". Note that "attorney general" wouldn't satisfy the last condition. Nouny multiwords can be designated as structured by marking a star (`*`) as a feature in their sense frame.

In the WordNet-based lexical augmentation described in Section 16, approximately 28,000 multiwords in the WordNet-based augmentation get starred. The criteria for marking a multiword W in this way are: (1) W is a common noun and has an **ESG** nouny analysis (**ESG** parsing is used to check this). (2) There are no punctuation marks in W. (3) The components of W are all common nouns, adjectives, or verb participles that are found in the ESG base lexicon.

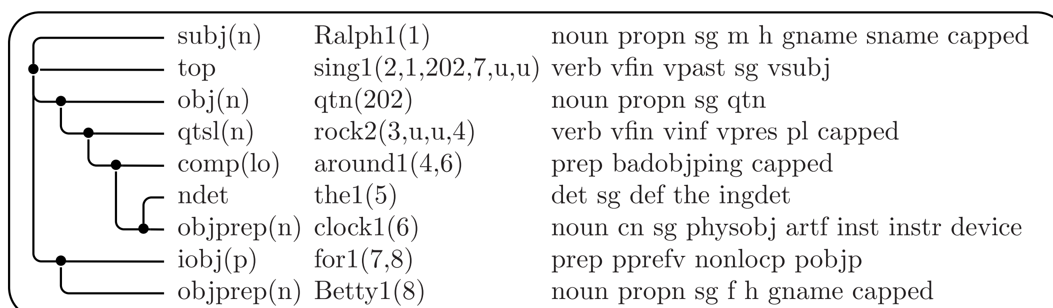
The structuring of star multiwords happens at parse time, but it is done in an efficient way – it doesn't add significantly to parse time. It is done at a stage before real parsing starts – at the stage where multiwords are being glommed and dealt with. But it does use regular parsing, just for those multiword strings.

There are two flags that control how the multiwords are handled: `parsemw` (on by default), and `parsemwa` (off by default). If both flags are off, then starred multiwords are treated just as normal (non-structured) multiwords. If `parsemw` is on, then all nouny multiwords with a star (*) in their lexical features will get structured. If `parsemwa` is on (the "`a`" suggests "all"), then *all* nouny multiwords will get structured.

11.3 Quote nodes

Another way *X***SG** can deal with chunks, in this case for named entities, is through *quote node*, or *qtnode* analyses in parse trees. Here is an example:

"Ralph sang Rock Around the Clock for Betty."



Note that there is a proper noun analysis above the subphrase "Rock Around the Clock", which has sense name `qtn` and feature `qtn`. This is a qtnode. This node has one slot `qtsl`, which is filled with a VP analysis of "Rock Around the Clock". So we "have it both ways", where the NE is clearly a proper noun chunk, but underneath there is an appropriate syntactic structure.

Qtnode analyses are based not on lexical entries, but are built at syntactic analysis time mainly through attention to capitalization and possible quoting. The NE may be actually quoted, and that helps, but quotes may be absent, as in the above example, and then capitalization is the main clue. Of course certain "little words" (like prepositions and articles) may be in lower case, also as in this example. But the needs of slot filling also play a role. Since prepositions may be in lower case, what is to prevent the whole string "Rock Around the Clock for Betty" being taken as the qtnode? What happens in this example is that "sing" allows a "for"-PP `iobj`, and the filling of this gets rewarded.

Qtnode analysis is turned on or off by the flag `qtnodes`, which is on by default.

11.4 Chunk lexicons

A way of dealing with named entities and terminology on a large scale is through *X***SG**'s *chunk lexicons*. Entries are typically for multiwords (hence the name "chunk"), but they can also be single words. A chunk lexicon can have a very large number of entries (though it has to fit in RAM).

When a chunk lexicon is used with *X***SG**, look-up is done before syntactic analysis starts, and the chunks found become the head words of nodes in the parse tree. Any features specified in the chunk lexicon for such a chunk are marked on the corresponding parse node, and this may be useful for downstream components of the NLP system.

The associated lexical look-up is done on top of – after – normal morpholexical processing. Chunk-lexical multiword agglomeration is done after regular lexical analyses have been converted into "starter" phrases prior to syntactic analysis. Inflectional variants of chunks are handled on the basis of morphological processing already done on their single-word components during the normal

lexical processing step.

Here are sample entries for a chunk lexicon:

```
=attorney general < n
economic growth < n
Federal Reserve < propn
parent company < n
net income < n
short sell < v obj < n obj
```

Entries are normally written one per line, and must start in column 1. But entries may be spread over many lines as long as continuation lines have some initial whitespace. It is better style to write one entry per line.

The entries do not have to be in alphabetical order. They are sorted during the process of compiling, described below.

The *index word*, or *chunk*, for an entry is the first part, up to the first occurrence of the separator symbol `<`, or up to the end of the line if there is no `<` on the line, and with trailing whitespace removed. The index word is usually a multiword, but may be a single word.

Punctuation symbols are allowed within index words, although it is typical not to have them. The *components* of an index word are determined by blank-separated tokenization of it.

Every index word has a *headword* component. The headword can be inflected. By default, the headword is the last component of the index word (or the index word itself it is a single word). But the default can be overridden by placing an `=` sign right before the headword. This is the case for the first entry in the examples above. Note that the plural of "attorney general" is "attorneys general".

The part of an entry after the first `<` is its *value*, or *sense frame* list. The value may have more than one sense frame, as for `short sell` above, and these are separated by `<` symbols.

If there is no `<` in the entry at all, then the value is taken to be `propn`, meaning that the index word is a proper noun. For example the entry could be simply

```
Federal Reserve
```

and then this chunk is taken as a proper noun.

The first token in a sense frame is normally the part of speech (POS) for that sense frame. The POS should be one of the symbols:

```
n (common noun)
propn (proper noun)
v (verb)
```

```
adj (adjective)
adv (adverb)
```

If the POS symbol is omitted, then the POS is taken to be `propn`.

After the POS, a sense frame may have slots and features, of the same form described in Section 8 and Section 7. One can often get by without specifying slots, for two reasons:

1. The most typical entries in a chunk lexicon are multiword nouns, and for these, the parsing often works rather well without using complement slots for the nouns.

2. For chunk lexicon index words that also have entries in the *X***SG** base lexicon (true most often for single words), there is a way, described below, of separating out such entries from the chunk lexicon and just letting their features augment the corresponding base lexicon entries. In this case, any slots specified in the chunk-lexical entry are ignored, and only the slot frames (usually well-thought-out) in the base lexicon are used.

But for verb entries in the chunk lexicon that are not in the base lexicon (like maybe for `short sell` in the above examples), it is good to specify at least the most basic slots, especially `obj` (direct object).

The features most often used in chunk-lexical entries are semantic features (semantic types). These may be ones known to *X***SG**, or may be freely added to sense frames by the user – e.g. if they are semantic types specific to a domain and an ontology for that domain. The special feature `sepmw` ("separable multiword"), described above for ordinary *X***SG** lexicons, can also be used.

This completes the description of the format of chunk lexicons.

Any number of chunk lexicons can be used at once with *X***SG**. They are specified in an ordered way. We describe below how to control this.

The look-up algorithm, given an input text segment, works basically as follows. It proceeds across the blank-separated tokens of the segment. For each token T, the goal is to find a chunk starting with T. All the chunk lexicons are tried in order, and the longest chunk match (starting with T), if any are found, is used as the looked-up chunk. That chunk is used as the head word for a "one-word" phrase that feeds into the syntactic analysis. If such a chunk is found, then the algorithm goes to the end of it, and continues the look-up with the token after that (if the end of the segment has not been reached). If none is found, then the algorithm simply goes on to the token after T (if the end of the segment has not been reached) and continues.

Before a chunk lexicon, say `abc.lx`, can be used with *X***SG**, it must be compiled into a binary form. The command to do this is:

*X***SG** `-compilechunklex abc.lx`

(The alternative command name `-compilehst` can also the used.) This produces two binary files

```
abc.lxw, abc.lxv
```

which are used at initialization time to read in the chunk lexicon quickly. The compiled form also reflects the internal data structure for the chunk lexicon. In this structure, the index words are stored as string arrays instead of simple strings. This makes look-up more efficient, and also facilitates the handling of inflections of headwords.

In order to use a (compiled) chunk lexicon `abc.lx` with *X***SG** parsing, the simplest command is to call *X***SG** like so:

```
XSG -chunklexf abc.lx [any other arguments]
```

(The alternative command name `-hstlexf` can also the used.) Note that one refers simply to `abc.lx`, even though it is the pair of binary files `abc.lxw`, `abc.lxv` that get used. If one wants to use, say, two chunk lexicons `abc.lx` and `def.lx`, then the command would be:

```
XSG -chunklexf "abc.lx def.lx" ...
```

Any number of chunk lexicons can be specified in this way.

There is a binary setting `lclookup` ("lower-case look-up") associated with any chunk lexicon, which affects look-up with that lexicon (`lclookup` is an integer field in the `chunklex` datatype). When `lclookup` is **on** (value `1`), look-up of each chunk component C is attempted only for the all-lower-case form of C. When `lclookup` is **off** (value `0` – the default) look-up of C is first attempted for C as-is (even if it has some upper-case letters). If this is not found, and it does have some upper-case letters, then look-up is attempted for the lower-cased form of C.

It makes sense to turn `lclookup` on only when all the index words of the chunk lexicon are in lower-case. And then it is more efficient to turn `lclookup` on, so that only one look-up is ever attempted for a component C. It is sometimes safe enough to keep all the chunk-lexical entries in lower-case, because there is no ambiguity between upper- and lower-case forms of the index words.

The setting of `lclookup` for a lexicon can be done in the call to `-chunklexf` by appending `:1` or `:0` to the lexicon name, according as you want `lclookup` on or off for that lexicon. So for example in

```
XSG -chunklexf abc.lx:1 ...
XSG -chunklexf abc.lx:0 ...
```

the first call turns `lclookup` on for `abc.lx` and the second call turns it off. Of course there is no point in turning it off, since it is off by default.

We have described the command option `-chunklexf` as used with the executable form of *X***SG**. But when the library form of *X***SG** is being used, either with the C-based API described in Section 14 or with the Java-UIMA API, there are similar ways of passing such arguments. See e.g. the API function `sgInit` in Section 14, which has arguments that are like the command-line arguments for an executable.

There is an *X***SG** flag `usechunklex` (alternative name `hstlex`) which controls whether any chunk-lexical look-up is actually done during parsing. This flag is off by default. But a call

*X***SG** `-chunklexf ...`

turns it on. Then the chunk lexicons named in this call will be used. However, you can keep the lexicons loaded and turn chunk-lexical processing off or on any number of times during a run by setting `usechunklex`. But it is not normal to need to do this switching, and you normally don't need to be concerned with `usechunklex`.

When index words of a chunk lexicon are also index words in the *X***SG** base lexicon, it is worth separating these out of the chunk lexicon, as mentioned above, and letting their chunk-lexical value features augment the corresponding base lexicon entries. *X***SG** comes with a way of doing this automatically, which we now describe. The process works for any language version of *X***SG**, but let us illustrate it for **ESG**, with executable `esg`.

ESG comes with a lexicon `enbase.lx` which is a special version of its base lexicon, where multiwords are kept as index words instead of being indexed under their headwords. The following assumes presence of `enbase.lx` in binary compiled form (`enbase.lxw`, `enbase.lxv`).

Suppose then that the chunk lexicon is `abc.lx`. The process will produce the following lexical files:

- `abcM.lx`: Its entries are those in `abc.lx` whose index words are not in `enbase.lx` (`M` stands for "Main" – the main part of `abc.lx`). Produced in both source and binary form.
- `abcE.lx`: Its entries are those in `abc.lx` whose index words are in `enbase.lx` (`E` suggests "ESG base lexicon"). Produced in both source and binary form.
- `enaug.lx`: A version of `enbase.lx` where the entry sense frames are augmented with the features marked in the corresponding entries from `abcE.lx`. The feature augmentation respects parts of speech; e.g. a *noun* sense frame in `enbase.lx` is augmented only by the features of *noun* sense frames for the corresponding entry in `abcE.lx`. Any slots specified in `abc.lx` are ignored in this augmentation process. This is because the slot frames specified in `en.lx` are carefully coded and sometimes complex. `enaug.lx` is produced only in binary compiled form.

The commands for carrying out this process are:

```
esg -lexdiffa abc.lx enbase.lx abcM.lx abcE.lx
esg -compilex abcE.lx
esg -lexaug abcE.lx enbase.lx enaug
esg -compilechunklex abcM.lx
```

It could be worth putting these in a script file, perhaps parameterized, at least for the chunk lexicon name `abc`.

For use of this transformation, the only resulting lexical files needed are:

```
(abcM.lxw, abcM.lxv)
(enaug.lxw, enaug.lxv)
```

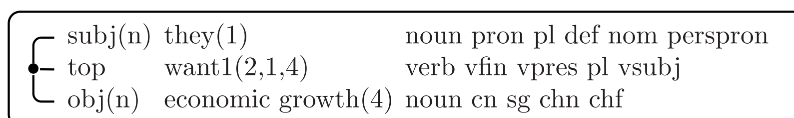
Then to call **ESG** and use the results, you would use the first or second of the calls:

```
esg -chunklexf abcM.lx -mlex enaug.lx
esg -chunklexf abcM.lx:1 -mlex enaug.lx
```

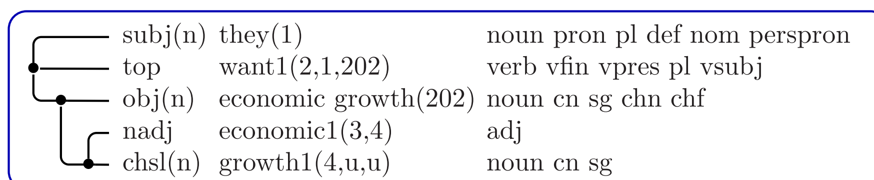
according as you want `lclook` off or on. The argument pair `-mlex enaug.lx` tells **ESG** to use `enaug.lx` as its base lexicon instead of `en.lx`. (For more details on `-mlex` and similar keywords, see Section 15.)

When a lexical analysis coming from a chunk lexicon is used in a parse, certain general features are marked on the parse node (in addition to features that actually appear in sense frames of the chunk lexicon). The feature `chf` is marked, no matter whether the analysis comes directly from the chunk lexicon specified with `-chunklexf` or from `enaug.lx` as described above. The feature `chn` is marked if and only if the analysis comes directly from the `-chunklexf` lexicon.

If the flag `chunkstruct` (alternative name `hststruct`) is turned on (it is off by default), then for each chunk node in a parse, an internal parse analysis of the chunk itself will be shown underneath the chunk, filling the slot `chsl`. Using the small sample chunk lexicon at the beginning of this section, the parse tree for "They want economic growth" with `chunkstruct` off would be:



But with `chunkstruct` on, the parse is



When the flag hyphenblank is turned on (it is off by default), chunk lexical look-up allows alternation between hyphens and blanks is allowed. I.e., a hyphen in the text can match a blank in the lexicon, or vice versa.

11.5 Creating chunks at tokenization time via tags

Normally, text tokenization by *X***SG** follows the usual rules for tokenization, paying attention to runs of letters, delineation by punctuation symbols, effects of abbreviations, and the like. But some special *X***SG** tag pairs – let us call them "chunking tags" – can be used to delineate larger tokens, or "chunk" tokens, so that any characters, even blanks and newlines, can appear in the chunk (aside from the end-tag of the tag pair).

Chunking tags can be used to delineate named entities, making multiword (or single-word) named entities into chunks, and even specifying semantic types for the chunks, which will figure in the parsing. Such tagging of named entities might be done by a preprocessor for *X***SG**, which passes the tagged text to *X***SG** for better parsing. Actually, the *X***SG** chunking tags can be used to provide much more general lexical specifications for chunks than are normally considered for "named entities", as we will see below.

In the next subsection we will describe an additional method for providing "chunking control" for *X***SG** at tokenization time. Instead of using tags, this method involves function calls to a hook (call-back) function that is monitoring the text for the existence of chunks at particular positions in the input document.

There are two classes of chunking tags known to *X***SG**: (a) *literal* or *delimiter* tags, and (b) *lexical* tags.

When a chunk xxx is surrounded by a *literal* tag <ttt> (and its end tag) as in

```
<ttt>xxx</ttt>
```

the chunk xxx will be treated like any other word that is looked up in the *X***SG** lexicon (with morphology) for parsing. If the chunk is not found in the lexicon, then it will be given the default lexical analysis for all unfound words – a singular proper noun. Currently, *X***SG** knows only one literal chunking tag, named lit.

The basic idea of a *lexical* tag <ttt> is that some of the lexical and semantic feature control lies with the tag itself. The chunk xxx surrounded by the tag and its end-tag may or may not be actually looked up in the lexicon, but even if it is looked up, the tag may filter out lexical analyses and may specify additional semantic features of the chunk. What is done depends on the attributes within the tag <ttt>, to be described below.

Three particular lexical tags – <ch> (for "chunk"), <ne> (for "named entity"), and <pn> (for "proper noun") – can contain attributes that specify lexical features of the chunk xxx. In spite of the different names for these three tags, they are all treated the same by *X***SG**. You might as well use <ch> in all cases.

I have kept `<ne>` and `<pn>` for compatibility reasons. In the discussion below, I will use only `<ch>`, but you could use either of `<ne>` and `<pn>` as well.

Here is an example of use of a lexical tag:

```
<ch lex="propn Company">International Business Machines</ch>
```

This will make "International Business Machines" into a chunk which is a proper noun (`propn`) and has the semantic type `Company`.

There are two types of attributes a lexical tag can have – `lex` and `sem`, which should be used separately (not both together in the same lexical tag).

Let us look first at `lex`. With the `lex` attribute, one completely specifies the lexical analysis of the surrounded chunk `xxx`, and `xxx` will not be looked up in the lexicon. In general, the form of a `lex` attribute-value pair in a `<ch>` tag is:

```
lex="LexicalAnalysis"
```

Note: No spaces are allowed on either side of the `=` sign. The same is true for other attribute value pairs below. The `LexicalAnalysis` is in general what can appear as a lexical analysis in an *X*SG lexicon (see [19]). (There is an exception in the notation, described below.) Such lexical analyses can specify not only proper noun frames, as in the above example, but frames of any part of speech, and even more than one frame.

Here is a more complex example:

```
<ch lex="n (p for) ~~ v obj">fixem up</ch>
```

This would give the multiword `fixem up` both a noun analysis `n (p for)` (a common noun with a `for`-prepositional complement slot) and a verb analysis `v obj` with a direct object slot. The symbol `~~` is used to separate lexical analyses in the `lex` attribute instead of the usual `<` in *X*SG lexicons. These lexical analyses can express anything that normal *X*SG lexical entries can – any parts of speech, slot frames, semantic types, subject area tests, etc.

One constraint, when the `lex` attribute is used, is that no morphology is done on the chunk. If the chunk appears in an inflected form, then the `lex` attribute must itself specify the morphological features. So for a noun plural `fixem ups`, you might write something like:

```
<ch lex="n (p for) pl">fixem ups</ch>
```

thus showing the noun plural feature `pl`.

Actually, for the LMT machine translation system, the `lex` attribute can show even more than just described. It can show complete LMT bilingual lexical entries for chunks, and LMT will use those in parsing (with *X*SG) and translation. In the `lex` attribute notation, the symbol `$$` is used instead of the symbol `>` that is normally used in LMT lexicons to introduce transfers.

Now let us look at lexical chunking tags where the `sem` attribute is used. Here is an example:

```
<ch sem="n PERSON MALE">fathers</ch>
```

When the `sem` attribute is used, the surrounded chunk *is* looked up in the lexicon, with morphology. Initially, in this example, the lexical analysis shows `fathers` both as a plural noun and as a third person singular present tense verb. However, with `sem`, only one lexical analysis will be used, which in this example will be the noun analysis, because of the initial POS symbol `n` in the value of `sem`. In addition, the semantic features `PERSON` and `MALE`, included in the value of `sem`, are added to the lexical analysis, and they will show in the parse tree for this node.

In general, the value of the `sem` attribute should be a string of the form

```
[POS] SemFea1 SemFea2 ...
```

The optional part of speech `POS` should be chosen from these part of speech symbols:

```
n, propn, pron, num, v, adj, adv,
det, prep, subconj, conj, qual
```

These are parts of speech that can appear in *X***SG** lexical entries. The first four are used for subcategories of nouns – common noun, proper noun, pronoun and number, respectively. The meanings of the others should be clear. If the `POS` is omitted, the default `propn` is used. The subsequent items $\texttt{SemFea}_i$ in the value of `sem` are semantic features, and will be added to the semantic features of the lexical analysis. These features need not be "known" to *X***SG**. They will appear in the parse tree on the node corresponding to the chunk.

So, given a `ch` tag with a `sem` attribute, *X***SG** will do morphology and lexical look-up on the surrounded chunk. But *X***SG** will select only one lexical analysis. If there is an analysis whose POS matches that of the `sem` tag, then *X***SG** uses the first such. Otherwise, if the POS of the `sem` is `propn` and there is an `n` (common noun) lexical analysis, *X***SG** will use the first such. Otherwise, *X***SG** will take the analysis to be `propn`. In all cases, any semantic features listed in the value of `sem` will be added to the lexical analysis.

If the `<ch>` tag has no `lex` or `sem` attribute, as in

```
<ch>International Business Machines</ch>
```

then the chunk will be given the default lexical analysis of a singular proper noun.

So there is a great deal of generality in the lexical specification allowed with the `<ch>` tag; it includes more than is usually encompassed under the notion of "named entity". I wish to thank Nelson Correa for a useful discussion about using *X***SG** chunking tags to specify properties particularly of named entities, which led me in part to the general formulation described here with lexical chunking tags.

*X***SG** knows several other lexical chunking tags. When HTML tag reading is turned on, the additional tags of this sort (besides `<ch>`, `<ne>` and `<pn>`) that are recognized are:

```
<code>, <samp>, <var>, <l>
```

*X***SG** does not recognize `lex` or `sem` attributes in these, and the surrounded chunks just get the default lexical analysis of singular proper noun.

11.6 Creating chunks at tokenization time via call-back functions

Now let us look at the second way of specifying chunks and their lexical analyses for *X***SG** at tokenization time (not using tags). This involves a "chunk hook function" (defined by the user of *X***SG**) which knows about document positions and tells *X***SG** where the chunks start and end (in terms of document positions), and what their lexical analyses are. This methodology is used when *X***SG** is embedded in UIMA (see the Introduction). I wish to thank Marshall Schor for discussions that led to this formulation.

To explain the form and use of chunk hook functions, I need to say a bit about the way *X***SG** handles document positions.

When *X***SG** is processing a file, the system keeps track of line numbers and column numbers in the file. Column numbers are like byte offsets on a line, so a column number of 0 is used for the position right before the first character. (You can tell *X***SG** to process a file either by calling the API function `sgFile` described below in Section 14, or by using the command `do` in interactive mode, as described above in Section 5.) So in the case of file input, document positions are specified by ordered pairs ($LineNo, ColumnNo$), and the hook function needs to deal with these.

On the other hand, *X***SG** can process a document given to it directly as a string, with the *X***SG** output also produced as a string. (You can tell *X***SG** to do this via the API function `sgSt`, described below in Section 14.) In this case, all that counts for giving a document position is the byte offset in the document string. Newlines are treated as normal characters that count in the byte offset. But *X***SG** still deals formally with line numbers and column numbers in this scenario, and the chunk hook function must also. The "line number" is always just kept at 0, and the "column number" is the byte offset in the whole document string.

Given this explanation of line numbers and column numbers for specifying document positions, we can now describe the form of a chunk hook function for specifying chunks. Such a function (let's call it `chhook` here) should have the following type:

```
sint chhook(ptr id, sint slineno, sint scolno,
            sint *elineno, sint *ecolno, st *annotation)
```

Here the types are defined by:

```
typedef void * ptr;
typedef long int sint;
typedef unsigned char * st;
```

(The type `st` has a different definition in case *X***SG** is compiled in Unicode mode – see Section 12.) The arguments of chhook are as follows:

1. `id` is the SG handle, as described in Section 14. You obtain this by calling the SG initialization function `sgInit`, described in that section.

2. `slineno` and `scolno` give the starting line and colmun number pair of the chunk. These are passed by the *X***SG** tokenizer to `chhook`.

3. `*elineno` and `*ecolno` give the ending line and colmun number pair of the chunk. These are passed back to the tokenizer by `chhook`.

4. `*annotation` is a string that is an attribute-value list. This is returned by `chhook` to *X***SG**. This list can contain one of the attribute-value pairs

   ```
   lex="LexicalAnalysis"
   sem="SemAnalysis"
   ```

   of just the same form allowed as `lex` or `sem` attribute-value pairs on the chunking tag `<ch>`. Then *X***SG** will associate a lexical analysis with the chunk, in just the same way it would if `<ch>` had been used. Actually, *X***SG** stores the whole `*annotation` list with the chunk, but currently *X***SG** does not use any more of it than a `lex` or `sem` attribute-value.

During tokenization, *X***SG** calls the `chhook` function at every spot where a possible new token could begin. It passes in the current document position to `chhook` in the pair `(slineno, scolno)`. If `chhook` *succeeds*, that is, returns a non-zero integer, then *X***SG** will create the chunk from all characters in the document stretching from the current position up to the ending position `(elineno, ecolno)` returned by `chhook`, or up to the end of the document, whichever comes first. And it will associate the `*annotation` list with the chunk, as indicated.

In order for *X***SG** to call a chunk hook function `chhook` during tokenization, you have to make `chhook` known to *X***SG** as such, and you do this by calling the API function

```
void sgSetChhook(ptr id,
                 sint (*chhook)(ptr, sint, sint,
                                sint *, sint *, st *))
```

where the first argument `id` is the SG handle and the second argument is (a pointer to) the hook function. So if the hook function is named `chhook`, you would call:

```
sgSetChhook(id, chhook);
```

Figure 6 shows a simple example for `chhook`, used with the file-mode of processing (where the line number coordinates vary), which for each line will cause the string stretching from column position 3 to column position 14 to be a common noun with semantic type `Weird`.

```
sint chhook(ptr id, sint slineno, sint scolno,
            sint *elineno, sint *ecolno, st *lex) {
   if (scolno != 3)
      return 0;
   *elineno = slineno;
   *ecolno = 14;
   *lex = "lex=\"n Weird\"";
   return 1;
}
```

Figure 6: Example of a chunk hook function

So what would this chunk hook function do for the following text?

```
In tagged text, tag pairs within segments like those that
indicate font changes, italics and boldface are often useful
to XSG in delineating subphrases of segments.
```

First `chhook` is called at position `(1, 0)`, because a token is starting there, but `chhook` fails. It is next called at position `(1, 3)`, and here it succeeds, creating the chunk `"tagged text"` (up through position `(1, 14)`). It is called several times more on the first line, but without success. On the second line, the only chance for a successful call would be at position `(2, 3)`, but the tokenizer is in the midst of doing the token `"indicate"` at that point, and no call is made. On the third line, there is a successful call at position `(3, 3)`, which creates the token `"XSG in deli"` (through position `(3, 14)`). The next token on that line (recognized by normal tokenization, not by `chhook`) begins immediately, and is `"neating"`.

We have discussed two ways of specifying chunks of text for *X***SG** and giving them lexical analyses. There is a third way that will work for most kinds of chunks – simply to put them in a user lexicon for *X***SG**. User lexicons are described below in Section 15, and they use the general SG lexical format specified in [19]. This will work in many cases, but there are advantages to using chunking tags or a chunk hook function:

- Named entities may be determined on-the-fly by a named entity recognizer that uses general rules instead of depending on a known list.

- Chunks (as specified in this section) can consist of *any* characters, whereas multiwords in *X***SG** lexicons are more restricted.
- If an *X***SG** lexicon has too many multiwords (e.g. in the hundreds of thousands), the processing may not be efficient enough. (This problem does not exist for single-word entries.)

12 The data structure for SG parse trees

The *X***SG** API described in this report uses a certain data structure for parse trees that we call `sgph` (for "**S**lot **G**rammar **ph**rase"). We describe this data structure in the current section. In the next section, we describe the representation of punctuation and tags in *X***SG** analysis structures.

The `sgph` data structure is considerably simpler than the data structure for phrases that SG uses internally during parsing. The definition of `sgph` involves only a total of six datatypes (including itself), but the definition of the internal phrase structure involves about eighteen datatypes. The external API datatype `sgph` contains essentially all the same applicable information as the internal one, but the internal one has finer distinctions for the sake of efficiency and control of the parsing process itself. For instance, as mentioned in Section 7, grammatical features are encoded via bit string positions in the internal representation.

In the `sgph` structure, features, slots, and slot options are represented as strings. The string datatype is named `st`. For the single-byte case, `st` is defined by:

```
typedef unsigned char * st;
```

For the double-byte (Unicode) case, `st` is defined by:

```
typedef wchar_t * st;
```

The list of features for an `sgph` structure is represented by a string list, `stlist`, defined by:

```
typedef struct Stlist {st hd; struct Stlist * tl;}
   * stlist;
```

We also find it convenient to have a separate name, `sint`, for long signed integers, defined by:

```
typedef signed long int sint;
```

And we define `ptr` (for "pointer") by:

```
typedef void * ptr;
```

```
typedef struct Sgph {
   sint wordno; sint lb; sint rb;
   stlist f;
   st word; st lcword; st cite;
   st sense; st ssense;
   intlist osenses; dbllist oweights;
   stlist compslots; struct Sgphlist * frame;
   struct Sgphlist * lmods; struct Sgphlist * rmods;
   st sl; st opt;
   struct Sgph * mother;
   double eval;
   ptr ref;
} * sgph;
```

Figure 7: Definition of the `sgph` data structure

```
typedef struct Sgphlist {
   sgph hd;
   struct Sgphlist * tl;
} * sgphlist;
```

Figure 8: Definition of the `sgphlist` data structure

Now we can define the phrase structure `sgph` itself. It is given in Figure 7. To be complete, this requires only the definition of `struct Sgphlist` (and the type of a pointer to it), which is given in Figure 8.

Notice that these two data structures are mutually recursive. If we express them totally in terms of the pointer types, we see that an `sgph` is a pointer to a structure with the fields:

```
sint wordno; sint lb; sint rb;
stlist f;
st word; st lcword; st cite; st sense; st ssense;
intlist osenses; dbllist oweights;
stlist compslots;
sgphlist frame;
sgphlist lmods; sgphlist rmods;
st sl; st opt;
sgph mother;
double eval;
```

And a phrase list, or `sgphlist`, is a pointer to a structure with the two fields:

```
sgph hd;
```

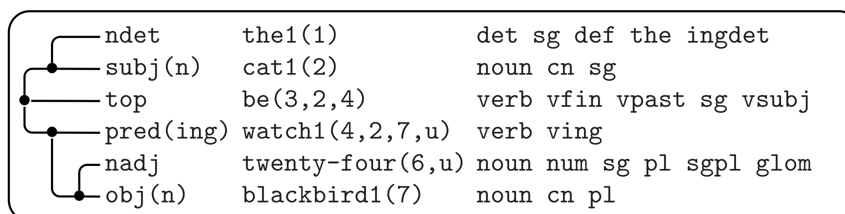
```
sgphlist tl
```

It is best to view `sgph` and `sgphlist` that way as we look further at what these structures represent.

The idea of the data structure `sgphlist` should be clear; it is just a list of `sgph` objects. So it remains to describe `sgph`.

Let us go through the different fields of an `sgph` and describe what they mean. We will look at an example parse tree to illustrate each point, the parse for the sentence:

The cat was watching twenty-four blackbirds.



So here are the individual fields of an `sgph` and what they mean:

`wordno`. The word number in the sentence of the head word of the phrase. In our example, for the subphrase for *the cat*, it is 2. This number shows in the parse tree display as the first argument of the word sense predication.

`lb`. The left boundary of the phrase in the sentence. It is one less than the sentence word number of the leftmost (single) word in the phrase. In our example, for *twenty-four blackbirds*, it is 4. In this example, the tokenizer sees *twenty* and *four* as separate tokens, and the word number of *twenty* is 5.

`rb`. The right boundary of the phrase in the sentence. It is the sentence word number of the rightmost (single) word in the phrase. For *the cat*, it is 2.

`f`. The feature list of the phrase, as a list of strings. For the whole sentence in the example, then, it is the list consisting of:

```
verb vfin vpast sg vsubj
```

`word`. The head word, exactly as written in the input, including capitalization. For the `ndet` filler, it is `The`.

`lcword`. The lower case form of the head word. For the `ndet` filler, it is `the`.

`cite`. The citation form (after inflections are "undone") of the head word. For the `obj` filler, it is `blackbird`.

`sense`. The sense of the head word. For the `obj` filler, it is `blackbird1`. For *twenty-four*, it is `twenty-four`. In parse tree displays, the default is to show the `sense` string as the predicate portion of each word sense predication. One can see this in the display of our example. However, flags can be set to show either the `cite` field of the `ssense` field instead; see Section 9, page 39.

`ssense`. A variant of the sense of the head word. In most cases, it will be exactly the same as `sense`, but for example for literal numbers, a numeric representation of the number is shown. For *twenty-four*, the `ssense` looks roughly like this:

```
(lnum 24.00)
```

For ordinal numbers like *twenty-fourth*, the `lnum` would be replaced by `olnum`.

`osenses, oweights`. These have to do with word sense disambiguation, and we omit the discussion here.

`compslots`. The list of names of the complement slots of the head word sense. The list is given in the order of the complement slots specified in the lexical entry for the word sense, or derived from such a lexical frame.[3]

`frame`. This is the list of filler phrases of the complement slots of the head word sense. They are in 1-1 correspondence with the slots (slot names) in the `compslots` field. I.e., the nth member of `frame`, if it is non-null, is the filler of the slot named by the nth member of `compslots`. The `frame` field is reflected in the arguments of the word sense predication in the parse tree – the arguments after the first one (which is the `wordno`). This is explained in part on page 5 above in Section 2. In our example, the word sense predication for *watching* is:

```
watch1(4,2,7,u)
```

In this form of the display, each member of the `frame` list is exhibited only by its `wordno`. The `2` is for the logical subject of *watching*, which is *the cat*. The `7` is for the logical direct object, which is *twenty-four blackbirds*. If a slot of the complement slot frame is not filled, then the corresponding member of `frame` is `nil` (the null pointer), and is indicated in the parse display by `u` ("unfilled" or "unknown") in the word sense predication. If the flag `predargslots` is turned on (see Section 9), then the slot from `compslots` corresponding to each `frame` member is also shown in the parse display, like so:

[3]Some complement slots may not be required in lexical entries (and so are implicit); for example the `subj` slot of a verb need not be listed, unless it has special conditions on its fillers. A word sense may be obtained by derivational morphology; in this case the morphology will create a derived complement slot frame from that of the base word.

```
watch1(4,subj:2,obj:7,comp:u)
```

The slots and fillers in `compslots` and `frame` are the *logical* slot-fillers for the word sense. For example a `subj` slot is the *logical* subject.

`lmods`. The list of left modifiers of the head word (each member of which is an `sgph` object). They appear in left-to-right sentence order in the list. Both they and the right modifiers are shown in the obvious way in the tree display.

`rmods`. The list of right modifiers of the head word. They also appear in left-to-right sentence order in the list. In the internal phrase data structure, the right modifiers are shown in the reverse order of this, for efficiency reasons.

`sl`. The slot filled by the phrase. This is shown in the parse tree display just to the left of the word sense predication.

`opt`. The option chosen for the slot filled by the phrase. For (most) complement slots, this is shown in parentheses after the `sl` field. It will be shown also for adjunct slots if the flag `showaopts` is turned on (see Section 9). For the top-level phrase, the slot is `top`.

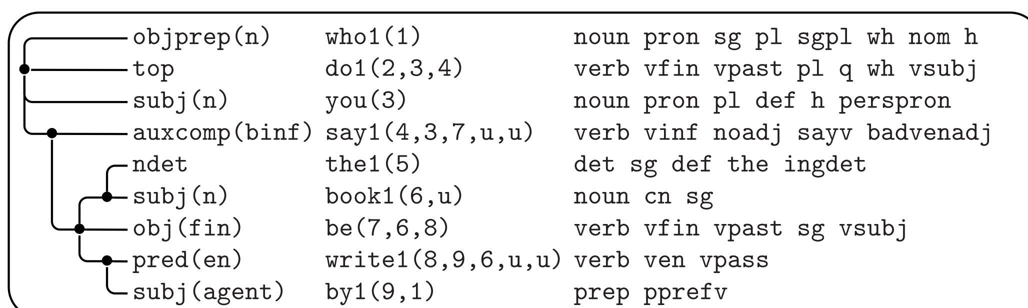
`mother`. The mother phrase (again an `sgph`) of the phrase. It is the null pointer for the top-level phrase, but is non-null for all subphrases.

`eval`. This floating point number (a `double`) gives a parse score for this phrase. We will not try to describe parse scoring in this report, but one thing that should be noted is that a higher score is a worse score.

Note that the `sgph` parse tree is "well-pointered". There is the upwards pointer provided by the `mother` field, and the downwards pointers in the `lmods` and `rmods` fields. Also there are the pointers in the `frame` field, which can point to a remote filler, as in examples like

Who did you say the book was written by?

where the parse tree is



Note here that the second argument (with `wordno` equal to 1) of the `by1` node points back to the `who1` node, showing that *who* is the `objprep` of *by*. In turn the agent `by1` node (with `wordno` equal to 9) is shown as the second argument of the `write1` node, which is for the logical subject of `write1`. This means essentially that *who* is the logical subject of *write*.

13 Punctuation and tags in SG analyses

The `sgph` data structure described in the preceding section by default does not show punctuation symbols or tags that appear in the input sentence. Many punctuation symbols, such as commas that set off adverbial modifiers of clauses, or font-change tags, are not crucial to the grammatical structure of the sentence. The *X***SG** parser *does* notice them and use them in constraining parses; but they are often optional to the grammaticality of the sentence, and their presence in the parse output is not essential to some applications. In fact, the omission of them in the parse structure simplifies most applications of *X***SG**, because this reduces the number of cases one has to look at in postprocessing of the parse structures. However, they are important for some applications, and they are indeed accessible in the *X***SG** API. We describe this in the current section. More generally, we describe how to see the results of *X***SG** tokenization.

The kinds of tags that can be omitted in the parse trees include font-change tags, HTML anchor tags, the chunking tags described in Section 11, tags that denote punctuation symbols, and others. Such tags usually occur in pairs (begin and end tag) within a segment, and often (but not always) bracket grammatical phrases. The tag pairs can sometimes cross segments, however. We call these *bracketing* tags in general, also sometimes *highlighting* tags.

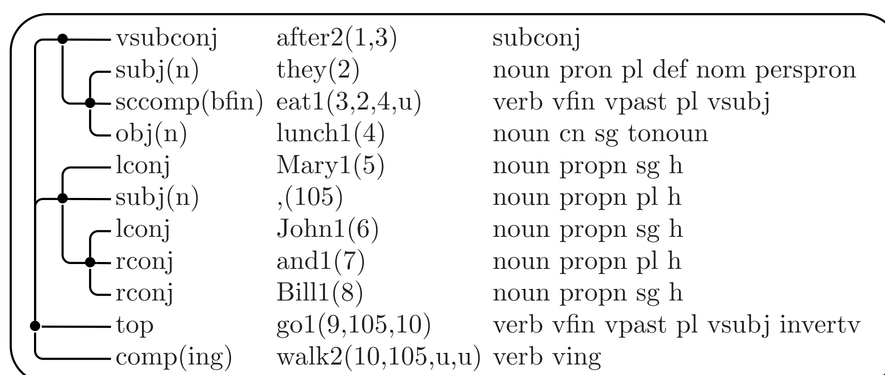
Of course some tags are used to denote grammatical elements that act as words or whole phrases in the segment, especially proper nouns. These *are* shown in the parse tree.

In the remainder of this section, we will usually just use the term "punctuation" to include both (normal) punctuation and bracketing tags.

There is an exception to the omission of punctuation in the *X***SG** parse trees. When a punctuation symbol plays the role of a coordinator, then it is shown in the parse tree. Consider for example the sentence:

After they ate lunch, Mary, John, and Bill went walking.

The parse tree is:



Note that the parse tree shows only one comma – the one after *Mary* that acts as a coordinator. (This is the case for the `sgph` data structure also.) The first comma (after *lunch*) is merely a phrase separator, and could even be omitted without loss of grammaticality in the sentence. The third comma is of course connected with coordination, but *X***SG** considers the real coordinator there to be *and*, with the comma used in an ancillary way. (Note that this comma is optional also.) The period at the end of the sentence is also not reflected in the parse tree.

When a punctuation symbol, like the comma after *Mary* in our example, is used as a coordinator, we say that it is *promoted* to a sentence tree node. In such a case, the promoted symbol is assigned a node number which is 100 plus the node number of the preceding word. In our example, *Mary* is the 5th word, so the comma after it gets node number 105 in the parse tree. In counting *words* of the sentence, one does not count the punctuation tokens. The constant 100 is actually a #defined constant in *X***SG**, called `sentlenmax`, which sets a maximum on the number of words in a segment that *X***SG** will try to parse. *X***SG** can be compiled with a higher value for `sentlenmax`.

So how does one see the punctuation symbols that are *not* promoted? The intuitive answer is that they are stored "in the interstices" between the words of the segment. The *interstices* are indexed by integers that run from 0 to the word number for the last word 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 punctuation occurring right after word i. Punctuation symbols that are promoted are also shown in these token lists in the interstices, in their proper place.

The next step in explaining these notions is to make precise what a *token* is. The *X***SG** tokenizer produces, for each segment, a list of objects of type `token`. These include both the *words* of the segment and the punctuation between the words. One can see a display of the tokens for a segment by turning on the flag `toktrace`. An example of `toktrace` output for a sentence is shown in Figure 9. (We have lined up the output in a tabular form that is a bit easier to read.)

```
Using <b>Actions</b>, you can open the data base.
```

type	id	word	lcword	spword	tagname	sp
wordtok	gtok	"Using"	"using"	"Using"	"using"	{3, 0, 3, 5}
lhili	gtok	""	""	" "	"b"	{3, 6, 3, 9}
wordtok	gtok	"Actions"	"actions"	"Actions"	"actions"	{3, 9, 3, 16}
rhili	gtok	""	""	""	"/b"	{3, 16, 3, 20}
septok	comma	","	","	","	","	{3, 20, 3, 21}
wordtok	gtok	"you"	"you"	" you"	"you"	{3, 22, 3, 25}
wordtok	gtok	"can"	"can"	" can"	"can"	{3, 26, 3, 29}
wordtok	gtok	"open"	"open"	" open"	"open"	{3, 30, 3, 34}
wordtok	gtok	"the"	"the"	" the"	"the"	{3, 35, 3, 38}
wordtok	gtok	"data"	"data"	" data"	"data"	{3, 39, 3, 43}
wordtok	gtok	"base"	"base"	" base"	"base"	{3, 44, 3, 48}
termtok	dot	"."	"."	"."	"."	{3, 48, 3, 49}

Figure 9: Display of token analysis for a segment

In Figure 9, seven fields of each token are displayed; they are labeled at the beginning of the output. There is an eighth field, called `annotation`, which is not shown in `toktrace` output, and we explain that below. Let us describe the seven fields.

The `type` field of a token (of C datatype `sint`) shows the general category of the token. The most common type is `wordtok`, for normal words. The type `lhili` (for "left highlight") is for begin bracketing tags, and `rhili` is for the corresponding end tags. The type `septok` is for punctuation (like commas and dashes) that can serve as a separator between a modifier and its modifiee, but such a token may also be promoted to a coordinator. The type `termtok` is for segment terminators. We will not try to list all the token types in this report, but the API function

```
st toktypename(token tok)
```

provides the string form of the `type` field of token `tok`.

The `id` field (of C datatype `sint`) is used to name very special tokens that figure heavily in the *X***SG** grammar. In Figure 9 one sees three examples, `comma` and `dot`, with obvious meanings, and `gtok` (standing for "general token"). There are quite a few possible `id`s in the system, mainly corresponding to specific punctuation symbols, and we will not list them here. The API function

```
st tokidname(ptr handle, token tok)
```

provides the string form of the `id` field of token `tok`. The first argument is the SG handle, as described in Section 14.

All the rest of the fields are of datatype `st` (string), except for `sp`.

```
typedef struct { sint slineno; sint scolno;
                 sint elineno; sint ecolno; }
   * span;

typedef struct { sint type; sint id;
                 st word; st lcword; st spword; st tagname;
                 st annotation; span sp; }
   * token;

typedef struct Tokenlist {token hd; struct Tokenlist * tl;}
   * tokenlist;
```

Figure 10: Definition of the `token` data structure

The `word` field is the underlying string for the token, just as it appears in the text, including capitalization.

The `lcword` field is like the `word` field, but is made lower case.

The `spword` field (for "space plus word") consists the `word` string, prefixed by any whitespace characters that occur between the preceding token and the current token. One can reconstruct the whole segment string by concatenating all the `spword` fields of the tokenlist created by the tokenizer.

For tags, the `tagname` field is the name of the tag, i.e. the part of the tag string that starts after the initial `<` and goes up to a blank or otherwise up to the closing `>`. It is also normalized to lower case. For non-tags, the `tagname` is the same as the `lcword`.

Finally, the `sp` field of a token gives its *span*. This represents the start and end positions of the token in the input document (not counting preceding whitespace). The `sp` field is of type `span`, defined in Figure 10. It is a pointer to a quadruple of integers giving, respectively, the start line and column numbers and end line and column numbers of the token. See the discussion on page 51 in Section 11 for more detail on how these positions are calculated.

The *X***SG** data structure for a `token` is defined in Figure 10. In this figure, we define not only `token` but also the auxiliary data types `span` and `tokenlist`.

From the preceding discussion, it should be clear what the meaning is of all these datatype ingredients, except for the `annotation` field of a `token`. This relates to chunking, as described in Section 11. If chunking is done through a chunk hook function (see page 51), then the `annotation` field of the chunk token is set to the annotation returned in the last argument of the chunk hook function.

Now let us return to the way the *X***SG** API allows one to see punctuation and word tokens.

First, the API function

```
token * sgWords(ptr id)
```

where `id` is the SG handle (as described in Section 14), returns an array of the regular word tokens for the most recently parsed segment. For convenience, let us call this array `Words` in the following. The API function

```
sint sgWordsLen(ptr id)
```

returns the index of the last word of the segment. Let us denote this by n in the remainder of this section. Thus the the members of `Words` are indexed by 1 through n inclusive.

Next, the API function

```
tokenlist * sgPunct(ptr id)
```

returns an array – call it `Puncs` – such that for each $i, 0 \leq i \leq n$, `Puncs[`i`]` is the list of punctuation tokens in interstice i.

For the example in Figure 9, the arrays `Words` and `Puncs` would look like this:

i	0	1	2	3	4	5	6	7	8
Words[i]		Using	Actions	you	can	open	the	data	base
Puncs[i]	()	()	(,)	()	()	()	()	()	(.)

In the `Words` row we are displaying the word tokens by their `word` fields. In the `Puncs` row we are displaying the punctuation tokenlists in the form $(w_1\ w_2 \cdots)$, where w_j is the `word` field of the jth list member.

It should be noted that the tokens in the `Words` array do not show multiword agglomeration. For example, the **ESG** lexicon has a multiword entry for *data base*. In the **ESG** parse of our example sentence, `data base` will show as a single node. Its `wordno` field will be 8, and no node in the tree has `wordno` 7. In general, the parse `wordno` of a multiword will be the `Words` index of the token that is the head word of the multiword.

14 Compiling your own *X*SG application

If you have the object files or a DLL or library file for the *X***SG** modules and you want to compile your own application that uses *X***SG**, then the following information is relevant.

You will get a header file `xsg.h` with the SG datatypes given in Section 12, and the prototypes for the API functions described in the current section. And you will get a DLL `xsg.dll`, `xsg.lib` or library file `xsg.so`.

In the following, let us use the following abbreviations (these definitions are in `xsg.h`).

```
#define nil NULL
typedef FILE * file;
```

Two API functions that you will need to call in your own program are:

```
sint sgInit(int argc, unsigned char **argv, ptr *id)
void sgClose(ptr id)
```

The function `sgInit` should be called to initialize *X***SG**. Its first two arguments are like the two arguments of `main` in a C program used to hold command-line arguments of the corresponding executable. Thus `argv` should be an array of strings, and `argc` is the length of that array. These arguments can be used to control initialization of *X***SG** and to cause *X***SG** to perform certain auxiliary operations. The initialization command-line arguments for the executable `xsg` described at the end of Section 3 can all be given to `sgInit` in its `argc` and `argv` arguments. We also discuss other aspects of this in Section 15. It is permissible to call `sgInit` with `argc` equal to 0 and `argv` equal to `nil` if no auxiliary operations are desired. The third argument `*id` will be set by `sgInit` to a pointer called the SG *handle*, which is used in other API functions. If `*id` is set to `nil`, this means that initialization was not successful – perhaps because of insufficient memory or because *X***SG** could not access its lexical files. In this case, *X***SG** will write an error message to `stderr`, and `sgInit` will return 0. The function `sgInit` may also return 0 if its first two arguments initiate certain auxiliary operations, as discussed in Section 15. Otherwise, 1 is returned.

We mention here one kind of parameter pair that `sgInit` can take in its first two arguments. A pair of parameters of the form

```
-lexpath Path
```

will allow the *X***SG** lexical files to be in another directory than the current directory. The `Path` string should show either the complete directory path (with an initial slash) for the lexical files, or else the subpath (without an initial slash) from the current directory. (The first option would be the normal one.) The `Path` string should not contain a final slash.

You should call `sgClose(id)`, where `id` is the SG handle set by `sgInit`, when you are finished using *X***SG**. This frees storage.

In the following, we will use `id` to refer to the SG handle set by `sgInit`.

There are two basic ways of using *X***SG**:

1. Your program is in charge of file handling and text segmentation, and you pass segment strings to *X***SG**, getting parses (`sgph` data structures) back.

2. *X***SG** is in charge of processing input texts and files, and calls a "hook" after each parse that invokes a function of yours that does what you want with the parse.

Let us look at these two methods.

(1) For each segment string `s`, you should call the *X***SG** API function:

```
sgph sgParsit(ptr id, st s)
```

This returns the best *X***SG** parse as an `sgph` tree.[4] Note: `sgParsit(id, s)` should not be called when the number of tokens in `s` is greater than 100.

You should call the API function

```
void sgClear(ptr id)
```

sometime after each parse but before calling `sgParsit` the next time. This clears storage space for the next parse. The storage area used in the returned parse from `sgParsit` will get overwritten after you call `sgClear`, so you should make sure to do what you need to do with the returned parse (like writing results to a file, or storing information in some other storage area of your own).

Under method (1) there is an alternative API function:

```
sgphlist sgParses(ptr id, st s)
```

This returns the list of *all* the parses for segment `s`, ordered according to parse score (best first).

(2) In the second method of interfacing *X***SG** to your own program, *X***SG** is in charge of file handling, segmentation, tag handling, etc. You need to define a "hook" function, with a prototype like this:

```
void my_hook(ptr id, sgph ph)
```

After each parse, *X***SG** will call `my_hook` with `id` as the SG handle and `ph` as the best parse for the sentence. But in order to communicate to *X***SG** that this is your hook function, you should make the call:

```
sgSethook(id, my_hook);
```

in your program at some point after your call to `sgInit`.

There are three ways that your program can invoke the operation of *X***SG**.

(a) A call to

```
void sgRun(ptr id)
```

will put *X***SG** into interactive mode, with the "`Input sentence`" prompt, described in Section 3. As indicated in that section, you can leave this loop by typing "`stop.`".

(b) You can call:

[4] Best according to parse score in the field `eval` of `sgph` phrase structures.

```
sint sgSt(ptr id, st text, st *analysis, sint *alen)
```

This takes input text as a string `text`, which may consist of any number of segments. The results (parse displays or other output) are written into the string `*analysis`. The length of the output string in bytes is put into `*alen`.

(c) A call to

```
sint sgFile(ptr id, st InFile, st Outfile)
```

will process the input file `InFile`, writing results to output file `OutFile`. Actually, the input argument string `InFile` for `sgFile` can be a *pattern* of files (using `*` as wildcard), for example `d:\texts*.html`. In this case, all the files matching the pattern will be processed, and all the output goes to the single file `OutFile`.

The API function

```
file sgOut(ptr id)
```

returns a pointer to the *(current) SG output file*. If `sgFile` is operating, then the SG output file is the output file specified for that function. If `sgRun` is operating, then the SG output file is `stdout` by default, but will be `sg.out` if the flag `xout` is on and *X***SG** is writing results. You can use `sgOut` to write your own data to the SG output file. You can also open the SG output file for a file name `fln` by calling:

```
sint sgOpOut(ptr id, st fln)
```

You can close the current SG output file by calling:

```
sint sgClOut(ptr id)
```

This will not only close the file, but will also set the current SG output to `stderr`.

Three output functions provided in the API are these:

```
void sgPrst(ptr id, st x)
void sgPnl(ptr id, st x)
void sgPri(ptr id, sint x)
```

The first writes string `x` and the second is similar but adds a newline. The third write the integer `x`. "Writing" is either to the current output file in the case of the function `sgFile` or the output string `*analysis` in the case of `sgSt`.

See Section 13 for a description of the following API functions, which allow access to the results of *X***SG** tokenization and the treatment of punctuation and tags.

```
token * sgWords(ptr id)
sint sgWordsLen(ptr id)
tokenlist * sgPunct(ptr id)
```

A useful API function for setting flags is:

```
sint sgSetflag(ptr id, st flg, sint val)
```

This sets a flag with name `flg` to the integer value `val` and returns 1, if `flg` is indeed a valid flag name. Otherwise it returns 0.

To set the SG editor to an editor with name `Ed`, call:

```
void sgSetEd(ptr id, st Ed)
```

To display a phrase `ph`, you can call:

```
void sgShowph(ptr id, sgph ph)
```

Our two basic methods of interfacing to *X***SG** are illustrated in short main programs in Figure 11 and Figure 12.

15 Using your own lexicons

For the format of SG lexicons, see [19]. Suppose you have such a lexicon and you want to use it with *X***SG**, as an addendum to the main lexicons supplied with *X***SG**. Let us be specific, and suppose that the source language is English and the name of the addendum lexicon is `ena.lx`. In general, lexicons should have file type (extension) `lx`.

The first step (after creating `ena.lx`) is to *compile* `ena.lx`. The purpose of compiling a lexicon is to produce binary files that can be loaded and accessed by *X***SG** more efficiently.

If we have compiled **ESG** with the main program in either Figure 11 or Figure 12, and the name of the executable is `esg`, then we can compile `ena.lx` by the command:

```
esg -compiles ena.lx
```

This will produce binary files:

```
ena.lxw, ena.lxv, ena.lxi
```

If we do not have the convenient relationship between the `main` program's command-line arguments and `sgInit` that exists in the programs in Figure 11 and Figure 12, and instead we want to compile `ena.lx` more directly in a call to `sgInit`, then we would set up the integer variable `argc` and the string array variable `argv`, and set

```
#include <xsg.h>

int main(int argc, st argv[]) {
   char seg[1000];
   sgph ph;
   ptr id;
   sint stopped = 0;

   if (sgInit(argc, argv, &id) != 1)
      return -1;
   while (!stopped) {
      printf("Input sentence (\"stop.\" to stop):\n");
      gets(seg);
      if (strcmp(seg, "stop.") == 0)
         stopped = 1;
      else {
         sgOpOut(id, "sg.out");
         ph = sgParsit(id, seg);
         fprintf(sgOut(id), "Parse of segment: \n%s\n", seg);
         sgShowph(id, ph);
         sgClOut(id);
         system("edit sg.out");
         sgClear(id);
      }
   }
   sgClose(id);
   return 0;
}
```

Figure 11: Interface to X`SG` based on `sgParsit`

```
#include <xsg.h>

void hook(ptr id, sgph ph) {
   fprintf(sgOut(id), "Head word of phrase = %s\n", ph->cite);
   fprintf(sgOut(id), "Showing phrase again:\n");
   sgShowph(id, ph);
}

int main(int argc, st argv[]) {
   ptr id;
   if (sgInit(argc, argv, &id) != 1)
      return -1;
   sgSetflag(id, "xout", 1);
   sgSethook(id, hook);
   sgSetEd(id, "edit");
   sgRun(id);
   sgClose(id);
   return 0;
}
```

Figure 12: Interface to X`SG` based on `sgSethook`

```
argc = 3;
argv[0] = "";
argv[1] = "-compiles";
argv[2] = "ena.lx";
```

and call `sgInit(argc, argv, &id)`.

When `sgInit` is called with the `-compiles` keyword in this way, `sgInit` returns 0. One can see that for the programs in Figure 11 and in Figure 12, the program will not go on with the rest of its possible activities after a compile via `-compiles`. Also in this case, `sgInit` will not try to find and load the main lexicons of *X***SG**.

The compiling function `-compiles` just described does not encrypt the binary lexical files. There is another version, `-compilesenc`, which is used the same way, but applies a certain amount of encryption to the binary files.

The next step is to cause *X***SG** to *use* `ena.lx` during parsing as an addendum to **ESG**ś main lexicon `en.lx`. Again, assuming that we have `esg` compiled from the program in Figure 11 or in Figure 12, we would invoke it like so:

```
esg -mlex "ena.lx en.lx"
```

Or if we wanted to do it more directly through a call to `sgInit`, we would set its arguments as:

```
argc = 3;
argv[0] = "";
argv[1] = "-mlex";
argv[2] = "ena.lx en.lx";
```

Although we refer here to the *source* lexicon file names (like "`ena.lx`"), the system actually reads the binary compiled forms of the lexicon.

The use of `-mlex` for specifying the lexicons in this way will cause the sequence of lexicons

```
ena.lx, en.lx
```

to be used in *preference order*. This means that when a word w is looked up during lexical analysis, *X***SG** will look in order in those two lexicons (`ena.lx` first), and will use only the first successful look-up of w (ignoring later lexicons in the list for that particular w). In this sense, the addendum lexicon *overrides* the other lexicon (with this order of arguments for `-mlex`). Of course we can change what overrides what by changing the order of arguments.

There is another way to use an addendum lexicon. Instead of letting its entries override entries for the same word in the other lexicons, we can set it up so that results of look-up in the addendum lexicon are merged or "unioned" with the results (for the same word) in the other lexicons. To cause this to happen, we can call `esg` like so:

```
esg -alex ena.lx
```

Or we could make a corresponding direct call to `sgInit`.

For an example of the use of `-alex` method, suppose we add a noun entry for the word `write` to `ena.lx`, like so:

```
write < n
```

Suppose `write` had only verb senses in the main **ESG** lexicons. Then with `ena.lx` used as an `alex`, the word `write` will get the noun sense from `ena.lx` *and* its verb senses from the main lexicon.

But one must be careful with `alex` lexicons. If there is too much overlap in entries with the main lexicons, then this could overload the parser with redundant analyses of words.

It is possible to specify both `alex` and `mlex` in the same call to `sgInit`. Example for the command-line case:

```
esg -mlex "ena.lx en.lx" -alex enc.lx
```

There is another compiling function for lexicons, `-compilex`. This is simpler than `-compiles`, in that it pays no attention to multiwords in head words for entries. Actually, `-compilex` does no encryption, but there is a variant of it, `-compilexenc`, which does do encryption. If you have a source form of the ontology lexicon `ont.lx` and you need to compile that, then you must use `-compilexenc`.

This completes our discussion of user lexicons. The initialization function `sgInit` can also invoke special utilities besides the lexical utilities discussed in this section, if suitable keywords are given to it. But these are not as important as the lexical utilities, and we will not discuss them in the current form of this report.

16 Lexical augmentation via WordNet

The lexicon `en.lx` distributed with **ESG** (in the binary files `en.lxw` and `en.lxv`) has no data derived from WordNet [21, 5]. **ESG** works well with just that base lexicon. However, it is possible to augment `en.lx` with information from WordNet, to improve performance. In this section, we describe how to do this.

WordNet is not distributed with ESG, but a user's installation of WordNet can be used, along with a utility `EnWNaug` packaged with the **ESG** executable `esg` to do the lexical augmentation of `en.lx` automatically. Automation of the process has the advantage that new versions of WordNet released from Princeton can be used – as long as the WordNet data files retain the same format as current ones, as in versions 2.0 and 3.0.

There are two aspects to the augmentation by `EnWNaug`: (1) Lexical entries derived from WordNet's noun vocabulary (both multiwords and single words) are added to the lexicon when they aren't in `en.lx`. (2) Semantic types, based on WordNet senses, are marked on the combined, larger lexicon.

Here's what `EnWNaug` does for (2). There is a small ontology $\mathcal{O}$ used by **ESG**, some of whose types get marked on entries. Currently $\mathcal{O}$ has about 130 semantic types, which I have chosen either because they are useful for **ESG** parsing, or because they are rather high-level types that might be useful to see marked in parse trees used in applications of **ESG**. Most of the types in $\mathcal{O}$ correspond to WN (WordNet) word senses (synsets); let $\mathcal{O}'$ be the set of those that do. The members of $\mathcal{O} \sim \mathcal{O}'$ can still be marked on parse tree nodes, because of hierarchical information in **ESG**.

The semantic types listed at the end of Section 7 are included in $\mathcal{O}$. They are the main ones that are useful for parsing. Some of those types are coded directly in `en.lx` and have long been there, independently of WN, although the WN augmentation sometimes results in *more* words getting marked with such types.

The basic process of type marking by `EnWNaug` on lexical entries is as follows. For each word W in the WN vocabulary, and each open-class POS (part of speech) P of W, we identify a set $S(W,P)$ of WN senses of W with POS P, as follows. Let $S'(W,P)$ be the set of *all* WN senses of W with POS P. If $S'(W,P) = \emptyset$, then we let $S(W,P) = \emptyset$. Otherwise we proceed as follows.

Basically we want to take $S(W,P)$ to be the set of all members of $S'(W,P)$ that occur with high enough (relative) *frequency*. To measure frequency, we use for any WN sense s, the corpus occurrence count $C(s)$ of s provided by WN. We let

$$T = \textstyle\sum_{s \in S'(W,P)} C(s).$$

And if $T > 0$, we take the frequency of a sense $s \in S'(W,P)$ to be

$$F(s,W,P) = C(s)/T,$$

and we let

$$S(W,P) = \{s \in S'(W,P) : F(s,W,P) \geq 0.2\}.$$

And if $T = 0$, we take $S(W,P)$ to consist of just the member of $S'(W,P)$ first listed by WN. The threshold 0.2 for sense frequency is a guess at what is useful, and perhaps should be adjusted.

Now let us describe how the set of WN senses $S(W,P)$ is used to mark types from $\mathcal{O}'$ on the augmented lexical entry for W. For each POS P of W and each $s \in S(W,P)$, we go up the WN hypernymy tree (it can branch upwards) from s, and for each WN sense s' encountered, if s' corresponds to a type $t \in \mathcal{O}'$, then

we mark t on all the sense frames of W with POS P. There is a short table in **ESG** showing for each $t \in \mathcal{O}'$ what WN senses correspond to t.

Note that there is no attempt at WSD. Types appropriate to more than one sense of W (for the given POS) can get marked on W. However, the use of the frequency threshold prevents rarer senses being used. For instance, for the noun "man", all of the senses besides the most common one ("adult male human") have frequency numbers below the threshold. So "man" gets only the types appropriate to that most common sense. On the other hand, **ESG** parsing is *largely* syntactic (though using semi-semantic information from detailed slot frame specifications). So if "man" is used in a rare sense, the parse normally works anyway.

Now let us describe how to run `EnWNaug`. The first step is to convert the WN data files to a form that **ESG** can use. For efficiency purposes, **ESG** has its own API to WN, which I wrote. The only portion of the Princeton WN distribution needed is the set of data files, which would typically be in a directory like

```
\Program Files\WordNet\2.0\dict
```

on Windows. The **ESG** WN API needs nine data files, and expects the names to be:

```
noun.dat, noun.idx, verb.dat, verb.idx
adj.dat, adj.idx, adv.dat, adv.idx,
sense.idx
```

If your distribution of WN uses those file names, then you can use them directly where they are. But if they have another naming scheme, like

```
data.noun, index.noun, ...
```

then they should be copied to files named as above, and it is better to put these in a separate, new directory.

The next step is to execute the command

```
esg -wnmake Dir
```

from the **ESG** directory, where `Dir` is the complete path of the directory (without a final slash) where the nine WN data files are. This should create nine data files:

```
wnsg.gls, wnsg.sid, wnsg.sns, wnsg.spc,
wnsg.swd, wnsg.syn, wnsg.top, wnsg.trc, wnsg.wds
```

in the **ESG** directory. These need to be there for the **ESG** WN API to work.

The next steps are for the actual lexical augmentation. The **ESG** distribution provides the lexical files

```
en.lxw, en.lxv,
enbase.lxw, enbase.lxv
```

and these should be in the **ESG** directory. Copy (not rename) the first two files to something like

```
enSV.lxw, enSV.lxv
```

for safe keeping. It is important to save these if you wish to redo the augmentation process (e.g. with a new WN distribution), or just to run **ESG** without a WN-augmented lexicon.

Then execute:

```
esg -enwnaug
```

This should take about 15 seconds to run, and it should create binary lexical files:

```
enaug.lxw, enaug.lxv, enaug.lxi
```

This is the new, augmented form of `en.lx`. Finally, make `enaug.lx` the new form of `en.lx` by doing a copy-replace:

```
enaug.lxw => en.lxw
enaug.lxv => en.lxv
```

This completes the process of augmentation.

If you want to redo the augmentation process, then the saved base version of `en.lx`,

```
enSV.lxw, enSV.lxv,
```

should be copied (with replacement) to

```
en.lxw, en.lxv
```

and the process can be repeated.

For combining WordNet augmentation (just the semantic type part) with chunk lexicons (see Section 11.4), the sequence of separation commands near the end of that section could be changed to:

```
esg -enwntyaug enbasety.lx
esg -lexdiffa abc.lx enbasety.lx abcM.lx abcE.lx
esg -compilex abcE.lx
esg -lexaug abcE.lx enbasety.lx enaug
esg -compilechunklex abcM.lx
```

17 Logical Forms and Logical Dependency Trees

ESG contains a module for building logical forms (LFs) on the basis of the parse trees, and these can be useful in inference. A sentence (or any NL segment) is assigned a list of logical predications – the logical form of the segment – which can represent the semantics of the segment. The logical form for a segment is actually embedded in a dependency tree structure, a **Logical Dependency Tree** (**LDT**), where the head of each node is a list of predications associated with the head word for the node.

ESG+LDT allows you to look at the whole LDT, or just see the list of all the predications. Accessing the whole LDT can be useful for systematic traversal of the LF of the segment. There are many user-settable options (explained below) for controlling exactly what is displayed.

ESG LFs are being used by Jerry Hobbs's group at ISI – see [22]. For the work at ISI, the **ESG**-based LFs for that work are more or less in the form developed by Hobbs – see [7, 8]. But what we describe here is an alternative (and optional) version of **ESG**-based logical forms – embedded in LDTs. These LFs are more "generic" – they do not have some of the special characteristics of Hobbs LFs – and are improved in various ways over what **ESG** was doing before.

The LDTs constructed by **ESG** can actually be produced in two different data structures – `term`s or `sgph`s. A `term` is an internal data structure used by *X***SG** in several areas. It is much the same as the "S-expressions" used in Lisp, having the parenthesized external syntax of Cambridge Polish notation. (The Penn Treebank uses this notation also.) An `sgph` is the normal data structure for *X***SG** parse trees, and was described in Section 12 above. **ESG** produces an `sgph`-based LDT by first building the `term` form, and then converting that to `sgph` form. We begin the discussion with the `term` form and then describe the `sgph` form below in Subsection 17.4.

17.1 Nature of ESG LFs and LDTs

Here is a sample of the LDT analysis of a sentence, "Three important papers on logic were published by Smith", shown with one of the display options:

```
      three(e1,e3,u)
      important(e2,x3,u,u)
    paper(e3,x3,u,e4,u) pl(e3)
      on(e4,x3,x5)
        logic(e5,x5,u)
  be_pass(e6,x6,x3,e7) vpast(e6)
    publish(e7,x7,x9,x3)
      Smith(e9,x9)
```

In general, the LDT has a formal tree data structure (described below), but you can see the tree structure in this display, indicated by indentation and newlines. The LF (without the tree structure) is a flat list of predications, which should be viewed as and'ed. The variables should be viewed as existentially quantified, unless a quantificational predicate overrides that. Most predicates have as first argument a generalized event argument (the *entity* argument). The first predication for a node corresponds closely to the **ESG** word sense predication for the node, but there are various other kinds of predications, explained below.

A Logical Dependency Tree is a term of the form

```
(Lmod1 ... Lmodm (hd HeadPred*) Rmod1 ... Rmodn)
```

where the `Lmods` and `Rmods` are recursively LDTs. The head component `(hd HeadPred*)` has the information about the head of the tree. Each `HeadPred` is a logical predication that says something about the head word for the node or the whole associated phrase for the node. The first `HeadPred` is always the word sense predication for the node; we will describe the other kinds of `HeadPreds` below. The arguments of a `HeadPred` are usually atomic.

Here is an example of an LDT display for the above example that shows the whole LDT term structure.

```
(
  (
    (
    (hd three(e1,e3,u))
    )
    (
    (hd important(e2,x3,u,u))
    )
  (hd paper(e3,x3,u,e4,u) pl(e3))
    (
    (hd on(e4,x3,x5))
      (
      (hd logic(e5,x5,u))
      )
    )
  )
(hd be_pass(e6,x6,x3,e7) vpast(e6))
  (
  (hd publish(e7,x7,x9,x3))
    (
    (hd Smith(e9,x9))
    )
  )
)
```

We describe below how to control the form of the output.

There are similarities of LDTs with the **Predicate Argument Structure** (**PAS**) of J. W. Murdock described e.g. in [12]. (PAS representations are built from regular **ESG** parses.) In both LDT and PAS, inessential nodes like the "by"-prepositional nodes for passive subjects are removed and their `objpreps` are shown directly as logical subject arguments (as with `Smith` (`x9`) as the logical subject of the verb `publish` in our example). But a difference is that the PAS argument structure combines, into a single "argument" list, the complement slot frame members and the adjunct modifiers of a head word, whereas LDT preserves the usual *X***SG** distinction between deep (logical) structure and surface structure. This is appropriate, because, logically, adjunct modifiers often have the head word as an argument rather than the other way around.

The **ESG** LF/LDT structure includes a treatment of generalized quantifiers that shows scoping/precedence. The quantifiers include determiners and pronouns like "all", "some", and "many", as well as focusing adverbs like "always", "sometimes" and "often". Even though scoping choices are shown, there is *no reshaping* of the LDT tree. Quantifier words (word senses) are treated basically like any other words. They appear in their normal surface positions – no reordering in cases of unusual quantifier precedence/scoping – and they are viewed as logical predicates. But the difference from regular word sense predicates is that the main arguments of a quantifier predicate are *sets* of node entities, and these sets determine the scoping and higher-order nature of the quantifier.

Here's an LDT output for a sentence in Medline:

```
All mycobacteria showed some degree of homology.

     all(e1,[e2],[e3])
   mycobacterium(e2,x2) pl(e2)
 show(e3,x3,x2,x5,u,u) vpast(e3)
     some(e4,[e5],[e3,-e2])
   degree(e5,x5,u,e6)
     of(e6,x5,x7)
       homology(e7,x7)
```

The two quantifiers here, with their arguments, are

```
all(e1,[e2],[e3])
some(e4,[e5],[e3,-e2])
```

The 2nd and 3rd arguments of a quantifier show, respectively, its *Base* (what it quantifies over) and its *Focus* (very roughly what is asserted of things in the Base). The Base and Focus are exhibited as lists of node entities (shown in square brackets here), and it is understood that everything under a specified node is also included. But you can specify exceptions, as in the `-e2` in the Focus of `some`. The particular arguments of `all` and `some` here show that `all` scopes higher than `some`.

In the preceding example, scoping precedence is left-to-right (leftmost is highest), but here's an ESG+LDT analysis of a Medline example where the scoping order is reversed:

```
Some dizziness was experienced by all patients.

      some(e1,[e2],[e3,-e7])
    dizziness(e2,x2,u)
  be_pass(e3,x3,x2,e4) vpast(e3)
    experience(e4,x4,x7,x2)
        all(e6,[e7],[e3])
      patient(e7,x7,u) pl(e7)
```

On the other hand, here's an example where "some" and "all" occur in the same order as in the example just given, but the scoping is left-to-right:

```
Some muscle cells lost all functional contacts with the neurones.

      some(e1,[e3],[e4])
      muscle(e2,x2,u) nmod(e11,x2,x3)
    cell(e3,x3) pl(e3)
  lose(e4,x4,x3,e7,u,u,u) vpast(e4)
      all(e5,[e7],[e4,-e3])
      functional(e6,e7)
    contact(e7,x7,u,e8) pl(e7)
      with(e8,e7,x10)
          the(e9,e10)
        neurone(e10,x10) pl(e10)
```

17.2 Accessing ESG+LDT

To turn on LDT/LF production, you should turn on the flag `doLF`. To see the output in the above form, using the executable for **ESG**, also turn on the flag `showLF`, setting its value to an integer that controls the LDT output form. The value 5 (explanation below) for `showLF` will produce a display as in the first example above.

There is another flag, `showLFf` ("show LF as a feature") which causes the LDT to be shown as a feature (string) on the top node of the parse tree, in the form:

```
(LF LDT eLF)
```

This would be a way of passing LF/LDTs to a superordinate module that works with regular **ESG** parse trees.

17.3 Controlling ESG+LDT Output

When `doLF` is on, **ESG** always constructs a full form of the LDT, with several kinds of `HeadPreds`. The flags `showLF` and `showLFf` control how many of these `HeadPreds` are actually displayed, as well as other options about the tree display.

In describing the display control, we'll just use `showLF`, but the same rules apply to `showLFf`.

Think of `showLF` as a binary number. The bit positions control various characteristics of the tree display, as follows. Let `V` be the value of the flag `showLF`.

- The `HeadPreds` can use either regular functional notation, with commas separating the arguments, or Cambridge Polish notation. These two notations are used respectively according as `(V & 1)` is on or off. It is on in our first example above, and off in the second.
- When `(V & 2)` is on, full tree notation is used; else only the `HeadPreds` are shown. It is off in the first example above, and on in the second.
- When `(V & 4)` is on, newlines and whitespace are used to make the tree structure clear; else the LF/LDT is displayed all on one line. It is on in both of the examples above.
- There is a `HeadPred` showing the usual **ESG** features of the node, and this is displayed iff `(V & 8)` is on.
- There is a `HeadPred` for the surface slot and slot option, filled by the node, and its mother node, and this is displayed iff `(V & 16)` is on. It is of the form `slot(e,Slotname,Slotoption,e1)` where `e` is the entity argument for the current node and `e1` is the entity argument for its mother.
- There is a `HeadPred` for the span of the current head word, and this is displayed iff `(V & 32)` is on. It is of the form `span(e,Begin,End)` where `e` is the entity argument for the current node, and the other two arguments show the begin and end offsets for the head word. Perhaps the span should additionally show the offsets for the whole phrase associated with the node.
- The LDT contains a `HeadPred` for the POS of each node, and this is displayed iff `(V & 64)` is on. The POS predication is of the form `pos(e,POS)` (illustrating it with functional notation), where `e` is the entity argument for the node (first argument of its sense predication) and `POS` is its POS in PTB notation.

Here is an example where the value of `showLF` is 69 (`1+4+64`):

```
Ralph invited Alice to have some chocolate.

     Ralph(e1,x1) pos(e1,NNP)
  invite(e2,x2,x1,x3,u,u,e5) vpast(e2) pos(e2,VBD)
     Alice(e3,x3) pos(e3,NNP)
     have(e5,x5,x3,x7) pos(e5,VB)
         some(e6,[e7],[e2]) pos(e6,DT)
       chocolate(e7,x7) pos(e7,NN)
```

Here is an example where the value of `showLF` is 21 (`1+4+16`):

```
What had John said that he tried to find?

     what(e1,x1) slot(e1,obj,n,e2)
   have_perf(e2,x2,x3,e4) vpast(e2) slot(e2,top,nop,e0)
     John(e3,x3) slot(e3,subj,n,e2)
     say(e4,x4,x3,e7,u,u) slot(e4,auxcomp,ena,e2)
         he(e6,x6) slot(e6,subj,n,e7)
       try(e7,x7,x6,e9) vpast(e7) slot(e7,obj,thatc,e4)
         find(e9,x9,x6,x1,u) slot(e9,obj,inf,e7)
```

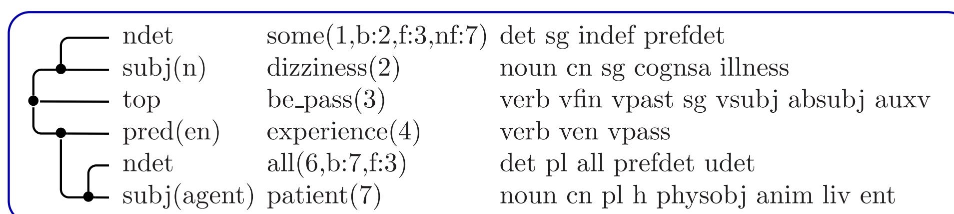
The word sense predication `HeadPred` normally just has atomic arguments, but you can make it show the corresponding complement slots also if you turn on the flag `predargslots`.

17.4 LDTs in sgph form

To turn on production of the `sgph` form of LDTs, turn on the flag `doLDT`. (This is not in addition to the enabling flags discussed above for `term`-form LDTs.) To display these trees, turn on the flag `ldtsyn`.

Here's the `sgph` version for an examplee given above.

Some dizziness was experienced by all patients.



Note the argument structure for the two quantifiers. In `term` form, the base and focus arguments are each a list of node IDs or their negatives (meant to be excluded). But in `sgph` form, we have to fit the `sgph` data structure, where each word sense argument consists basically of a slot and its filler. So we encode this by using:

- Slot `b` for each node ID in the base
- Slot `nb` for each node ID excluded from the base
- Slot `f` for each node ID in the focus
- Slot `nf` for each node ID excluded from the focus

(Thanks to J. W. Murdock for a suggestion on this.)

The LDT output above only shows slot names for quantifier arguments. But if we turn on the flag `predargslots`, we see slot names in all cases:

```
  .----- ndet          some(1,b:2,f:3,nf:7)         det sg indef prefdet
.-+----- subj(n)       dizziness(2,nobj:u)          noun cn sg cognsa illness
o------- top           be_pass(3,subj:2,pred:4)     verb vfin vpast sg vsubj absubj auxv
`-+----- pred(en)      experience(4,subj:7,obj:2)   verb ven vpass
  | .--- ndet          all(6,b:7,f:3)               det pl all prefdet udet
  `-+--- subj(agent)   patient(7,nobj:u)            noun cn pl h physobj anim liv ent
```

To get the LF associated with an `sgph`-based LDT, one essentially just and's together the word-sense predications of the tree after converting them to logical predications. From the above display form, the numerical arguments (representing phrases) are replaced by corresponding logical variables.

17.5 Related work on logical forms

As mentioned above, the **ESG**-based LFs described in this section are related to those of Hobbs ([7, 8]) – though there are differences. In both cases, the LFs are flat structures, where generalized event/entity arguments are used to effect nesting and scoping. **ESG** can produce LFs also in essentially Hobbs's form, though we don't discuss that here.

The author's chapter of the book [23] describes a logical form language **LFL** with a semantic interpreter based on an earlier version of **ESG**. LFL expressions are nested (not flat), but LFL does use general entity variables that can index arbitrary sub-LFs of LFL expressions. LFL deals with "generalized generalized quantifiers", in a single framework called *focalizers*, which include not only the usual generalized quantifiers but also focusing adverbs and discourse-functional focalizers. This work was done in the early 1980s.

Arendse Bernth used **ESG** as the parser in her discourse understanding system **Euphoria** [2, 1, 3]. Euphoria takes an English discourse and produces a discourse LF for it that has references resolved and various sorts of disambiguation applied. The LF is a flat one, using a version of general entity arguments, and is in that way similar to Hobbs's. Each entity has associated with it an entity-oriented LF for which it is the index: All the things that are said about the entity are listed with it and are accessible efficiently from it. In Euphoria predications, the predicate is a compound expression that combines the word

sense and (in effect) the entity argument, and such predicates are unique across the whole discourse, exhibiting reference resolution in that way. The same entity can be referred to by different parts of speech (like noun vs. verb), and these are normalized.

The *Linguistics and Philosophy* paper [20] by McCord and Bernth, "A Metalogical Theory of Natural Language Semantics", presents a comprehensive logic-based NL semantic representation system. The LFs in this system are not flat – they are nested and use set expressions as predicate arguments – but they do use general entity arguments (as with Hobbs's LFs) for some word sense predicates.

18 ESG parse trees in Penn Treebank form

ESG can produce its parse trees (roughly) in Penn Treebank (PTB) form. They are expressed in the `term` data structure (Lisp-like expression). To activate this output, turn on the flag `ptbtree`. If `ptbtree` has value `1`, then the tree will be output on a single line. If `ptbtree` is `2`, then the tree will be displayed in a nice indented form across several lines. Here is an example for `ptbtree = 2`:

```
Michael likes chocolate mousse cake.

(S
  (NP-SBJ
    (NNP Michael) )
  (VP
    (VBZ likes)
    (NP
      (NP
        (NP
          (NN chocolate) )
        (NN mousse) )
      (NN cake) ) )
  (. .) )
```

If the flag `ptbtreeF` is on, then the PTB tree, as a string without newlines, will be included in the feature list of the regular **ESG** parse tree. This can facilitate communication of the PTB tree in the regular API for **ESG**.

References

[1] Arendse Bernth. Deep analysis of biomedical abstracts. Technical report, IBM T. J. Watson Research Center, 2006. RC 24098.

[2] Arendse Bernth. Implicit predicate arguments and discourse. Technical report, IBM T. J. Watson Research Center, 2006. RC 24046.

[3] Arendse Bernth. Discourse semantics for biomedical information discovery. In *Proceedings of the Seventh International Workshop on Computational Semantics*, pages 301–304, Tilburg University, NL, 2007.

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

[5] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998.

[6] Jeffrey S. Gruber. *Studies in Lexical Relations*. PhD thesis, MIT, 1965.

[7] Jerry R. Hobbs. An improper treatment of quantification in ordinary english. In *Proceedings, 21st Annual Meeting of the Association for Computational Linguistics*, pages 57–63, Cambridge, Massachusetts, 1983.

[8] Jerry R. Hobbs. Ontological promiscuity. In *Proceedings, 23rd Annual Meeting of the Association for Computational Linguistics*, pages 61–69, Chicago, Illinois, 1985.

[9] Ray S. Jackendoff. *Semantics and Cognition*. MIT Press, Cambridge, MA, 1983.

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

[11] 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.

[12] M. C. McCord, J. W. Murdock, and B. K. Boguraev. Deep Parsing in Watson. *IBM Journal of Research and Development*, 56(3/4):3:1–3:15, 2012.

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

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

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

[16] 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.

[17] 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.

[18] Michael C. McCord. A formal system for Slot Grammar. Technical report, IBM T. J. Watson Research Center, 2006. RC 23976.

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

[20] Michael C. McCord and Arendse Bernth. A metalogical theory of natural language semantics. *Linguistics and Philosophy*, 28:73–116, 2005.

[21] George A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38:39–41, 1995.

[22] Ekaterina Ovchinnikova, Niloofar Montazeri, Theodore Alexandrov, Jerry R. Hobbs, Michael C. McCord, and Rutu Mulkar-Mehta. Abductive reasoning with a large knowledge base for discourse processing. In *Proceedings of the 9th International Conference on Computational Semantics*, pages 225–234, 2011.

[23] 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.