

Basic Form of a Syn-Struc (Syntactic Structure)

The simplest syn-strucs are those for simple nouns, like *dog*. Simple nouns are nouns that do not take arguments (an example of an argument-taking noun is *destruction*, since we can say *the destruction of the city by the invaders*).

The syn-struct for a simple noun looks as follows:

```
(syn-struct
  ((root $var0) (cat n)))
```

(*cat n*) means that the category being described is a noun or noun phrase (the distinction is not important at this point).

(*root \$var0*) means that the head – or main – noun is the entity being described in this lexicon entry. Other variables (*\$var1*, *\$var2*, etc.) are used to refer to the arguments of argument-taking words.

These two descriptors can be written in any order:

```
(syn-struct
  ((root $var0) (cat n)))
```

or

```
(syn-struct
  ((cat n) (root $var0)))
```

To further illustrate the formal conventions used to write syn-strucs, we will use the following example, which describes a typical ditransitive verb, like *give* in *He gave Mary the book*.

The syn-struct opens with an indication that this is the syn-struct

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n)))
   (directobject ((root $var2) (cat n)))
  ))
```

The rest of the content of the syn-struct is contained in another set of parentheses. One way of helping to keep parentheses matched – apart from working in Lisp or Emacs in Lisp mode, which have automated parentheses matching (which is a must!) – is to write the syn-struct in the same physical orientation all the time. For example, it helps to write the emboldened parentheses below always like this:

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n)))
   (directobject ((root $var2) (cat n)))
  ))
```

or like this

```
(syn-struct
  (subject ((root $var1) (cat n)))
  (root $var0) (cat v)
  (indirectobject ((root $var3) (cat n)))
  (directobject ((root $var2) (cat n)))
)
)
```

That leaves the body of the syn-struct looking like this—an orderly listing of syntactic elements:

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n)))
   (directobject ((root $var2) (cat n)))
  ))
```

Each element of the syn-struct (i.e., each line above) describes one syntactic element. Each syntactic element must be described by the following, listed in any order.

1. its part of speech, indicated by the relevant abbreviation and introduced by “cat” (= category)
2. a variable that will link it to an element in the semantic structure (sem-struct). Variables in the syn-struct, by convention, take the following form: dollar sign + *var* + number, e.g., \$var1, \$var2, \$var3. One can use any numbering for variables with the exception that the main element being described in the sense (in our example, the verb) is always \$var0. \$var0 is never linked to an element in the sem-struct because the entire sem-struct describes \$var0. The word *root* indicates that the given property applies to the head word – so, for example, the root of a noun phrase is the head noun (*tree* in *the big green tree*). It is the root that is assigned a variable. In addition, as will be shown later, one can specify that the root has to be one or more specific words.

Returning to parentheses, consider the subject from the example above. One set of parentheses enclose its full description

```
(subject ((root $var1) (cat n)))
```

another set of parentheses enclose its set of descriptors

```
(subject ((root $var1) (cat n)))
```

and yet more parentheses enclose each individual descriptor.

```
(subject ((root $var1) (cat n)))
```

The word being described in any lexicon sense is special in that it is not introduced by any category name in the syn-struct. Continuing with our verbal example, the verb, as the “core” of the syn-struct, is described simply as: (root \$var0) (cat v).

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n)))
   (directobject ((root $var2) (cat n)))
  ))
```

Contrast this with the subject, indirect object and direct object, which are introduced by their respective labels before the listing of their category and variable number.

Lining up the corresponding parts of the syn-struct elements will further clarify this point.

```
(syn-struct
  (subject      ((root $var1) (cat n)))
                (root $var0) (cat v)
  (indirectobject ((root $var3) (cat n)))
  (directobject  ((root $var2) (cat n)))
  ))
```

This alignment also helps to emphasize that the verb **selects** the other categories as its arguments and/or adjuncts. The above is an entry that describes the verb – it’s all about the verb.

(There are, however, a few situations in which the \$var0 element will be introduced by a category label; more on that later.)

When describing the category of a noun phrase as a subject, directobject or indirectobject, one can call its category either **n** or **np**, which are understood by the system to be synonymous. However, if one introduces *np* as an immediate constituent – in contexts to be described below – its category must be **np**.

```
(np ((root $var1) (cat np)))
```

All elements in the syn-struct are assumed to be obligatory unless noted as optional. Optionality is noted using the descriptor (opt +). For example, if the indirect object in our example were optional we would write the following:

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n) (opt +)))
   (directobject ((root $var2) (cat n)))
  ))
```

Optionality must be indicated at the level of the root, though its ordering with respect to the root and its category is not fixed. Thus the following variants are also fine:

```
(indirectobject ((root $var3) (opt +) (cat n)))

(indirectobject ((opt +) (root $var3) (cat n)))
```

One can specify a lexical category as the root, if applicable. There are many cases where one might want to specify not only a type of lexical category but a specific word or words that must be found in the syntax. One such instance (we'll see many more later) is when writing phrasal lexicon entries, as must be done, for example, for idioms. Take the idiom *draw a comparison*, which means 'compare' (*He drew a comparison between the IRS and the CIA*, which can be rephrased as *He compared the IRS with the CIA*). Its syntactic structure is

[subject] + [some form of *draw*] + [*comparison* (possibly with modifiers)].

Draw + comparison will be entered in the lexicon as a verbal sense of *draw*. In order to specify that the direct object is not any noun phrase but namely one headed by the word *comparison*, we write (*root comparison*) as a descriptor of the direct object.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (directobject ((root $var2) (cat n) (root comparison)))
  ))
```

We still must assign the direct object a variable (for purposes of semantic linking), so (*root \$var2*) and (*root comparison*) are compatible and must be used together.

Note also that any lexical constituent specified as the root must be recognized by the preprocessor, which for all intents and purposes means that it must have its own lexical entry. So you cannot build an ad hoc phrase on the fly and specify it as a root in a syn-struct. E.g., if you want to specify *brick_wall* as the root in some other entry, there must be a lexicon entry called *brick_wall* (NOT an entry for *brick* that includes a specification of *wall* or vice versa, but a head phrase *brick_wall*). As long as none of the components of a multi-word entity inflect, you can have quite long head phrases in lexicon senses: e.g., "an_ounce_of_prevention_is_worth_a_pound_of_cure" is a perfectly valid head phrase.

Two Methods for Writing Syn-Strucs

You can express the elements in a syn-struct in one of two ways:

1. referring to **syntactic functions** like subject, direct object (directobject), indirect object (indirectobject), prepositional phrase (pp)
2. referring to **immediate constituents** like np, n, adj, prep, conj, etc.

There are advantages and disadvantages to each method, and although any syn-struct can be described using immediate constituents, there are restrictions on using syntactic functions.

Writing Syn-Strucs Using Syntactic Functions

It is preferable, when possible, to write syn-structs using syntactic functions because there are certain properties (primarily word order in English) of subjects, directobjects, etc., that our analyzer has rules for. To exploit these rules, we need to use syntactic function labels. However, not all things that need to be expressed in lexical senses can be expressed using the limited inventory of syntactic functions; therefore, we often have to refer to immediate constituents, which are not covered by any such rules.

The inventory of syntactic functions is as follows:

subject

I gave him the book. [entry: *give*]

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n)))
   (directobject ((root $var2) (cat n)))
  ))
```

NB: Progressive (i.e., *-ing*) verb forms and infinitivals (e.g., *to go*) are often used as nominals: *Laughing is good for the health. To laugh is to make people around you happy.* The analyzer knows this and no special mention need be made in lexicon entries.

directobject

I gave him **the book**. [entry: *give*]

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
  ))
```

```
(indirectobject ((root $var3) (cat n)))
(directobject ((root $var2) (cat n)))
))
```

indirectobject

I gave **him** the book AND I gave the book **to him**. [entry: give]

Both of these variants are automatically covered when you indicate *indirectobject*: there is no need to make a separate sense with a PP headed by *to*.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var3) (cat n)))
   (directobject ((root $var2) (cat n)))
  ))
```

comp

I know **(that) you have to go soon**. [entry: know]

Comp is a sentential complement optionally introduced by complementizers like *that*, *how*, *whether*, etc. There is no need to explicitly indicate whether a complementizer is used and if so, which. (This will come later.) The *cat* for *comp* is *v* (i.e., a *comp* represents a verb-headed constituent).

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (comp ((root $var2) (cat v)))
  ))
```

xcomp

I want **to go**. [entry: want]

Xcomp is an infinitival complement that includes *to* (i.e., there is no need to indicate that 'to' needs to be used – that is understood when one uses *xcomp*). Like *comp*, *xcomp* has *(cat v)*.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (xcomp ((root $var2) (cat v)))
  ))
```

She's employing bad methods **to get her way**. [entry: employ]

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (directobject ((root $var2) (cat n)))
   (xcomp ((root $var3) (cat v)))
  ))
```

pp

He transferred money (**into the bank account**). [entry: transfer]

pp can indicate a prepositional phrase that is either an argument or an adjunct. In order to show that it is an adjunct (i.e., optional), one must add the descriptor (opt +). (In fact, (opt +) can show the optionality of any syntactic category.) When a specific preposition is, or prepositions are, necessary for a given sense, they are written as the root. So, for the example of *transfer*, the *pp* must be headed by the preposition *into*, which is explicitly indicated by (*root into*). Most commonly, the lexical root of a *pp* is (or lexical roots are) indicated in the *syn-struct*.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (directobject ((root $var2) (cat n)))
   (pp
    ((root into) (root $var3) (cat prep) (opt +)
     (obj
      ((root $var4) (cat n))))))
  ))
```

mods

Mods is used for adjectives and adverbs used as modifiers. If you label them as *mods*, then the order of constituents is based on general rules of English, stored in the analyzer. If you label adjectives and adverbs as *adj* or *adv*, then you are fixing their order as the order listed in the lexicon sense (see the next section about word order restrictions when using immediate constituents).

green book (entry: green)

```
(syn-struct
  ((mods ((root $var0) (cat adj)))
   (root $var1) (cat n)
  ))
```

walk **quietly** (entry: quietly)

```
(quietly-adv
 (syn-struct
  ((root $var1) (cat v)
   (mods ((root $var0) (cat adv))))
 ))
```

obj

I put the book on **the table** (entry: put)

Obj is used to indicate the object of a preposition in descriptions of prepositional phrases.

```
(syn-struct
 (subject ((root $var1) (cat n)))
 (root $var0) (cat v)
 (directobject ((root $var2) (cat n)))
 (pp
  ((root on) (root $var3) (cat prep)
   (obj
    ((root $var4) (cat n))))))
 ))
```

The examples above use the syntactic-functions method of writing syn-structs.

The benefits of writing syn-structs using syntactic functions:

- The label **indirectobject** covers both the NP (*I gave **John** the book*) and PP (*I gave the book **to John***) realizations of indirect objects.
- When using syntactic function labels, the elements of a syn-struct can be written in **any order** and the analyzer understands their actual (often multiple) potential orderings based on general rules of English syntax, stored in our analyzer. For example, the order of objects selected by English *give* depends on whether the indirect object is realized as a pp (*he gave **a book** to his sister*), or as an NP (*he gave her **a book***). Using the label *indirectobject*, written in any order in the syn-struct, covers both options.

The drawback of writing syn-structs using syntactic functions:

- Only the syntactic functions in the above inventory can be used. If you need to indicate any other categories – like a post-verbal particle, punctuation, a possessive, an oblique NP, etc. – you need to write the syn-struct using **ordered immediate constituents**.

NB: As soon as you include in a syn-struct any label apart from the ones listed above, the entire syn-struct will be converted by the analyzer into a phrase structure rule: e.g., subject, directobject and indirectobject will be converted to NPs and their ordering will be understood as fixed in whatever order is indicated in the syn-struct, with no reference

to global rules of English syntax. In other words, including any immediate constituents converts the whole syn-struct into the form of a phrase structure rule. To state it another way, **once a “fixed-order” element is included in a syn-struct, the order of all elements is understood to be fixed.**

Writing Syn-Structs Using Immediate Constituents

Among the types of entries that must be written using immediate constituents are verbs with post-verbal particles (*She started **out** on the project already*), conjunctions, any entries that include punctuation marks, instances where possessives or noun-noun compounds are explicitly indicated, and more.

Syn-structs presented in terms of immediate constituents are **ordered** lists of elements using the following labels.

prep

She started **out** on the project (entry: start)

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (prep ((root $var3) (cat prep) (root out)))
   (prep ((root $var4) (cat prep) (root on)))
   (np ((root $var2) (cat np))))
)
```

Prep is most often used for post-verbal particles. However, once you are already writing a syn-struct using immediate constituents, you may (or may not) choose to write pps as prep + np. In the example below, *out* is a verbal particle that must be written using *prep*, whereas *on* is the head of a prepositional phrase that can either be written as prep followed by np, or as pp in the way shown above.

She used a brush to paint (**with**). (entry: use)

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (directobject ((root $var2) (cat n)))
   (xcomp ((root $var3) (cat v)))
   (prep ((root with) (root $var4) (cat prep) (opt +)))
  ))
```

np

She started out on **the project**. (entry: start)

np is, of course, a noun phrase. When you express a noun phrase as np, its category must be np (not n, as was possible when we expressed noun phrases using subject, directobject, etc.).

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (prep ((root $var3) (cat prep) (root out)))
   (prep ((root $var4) (cat prep) (root on)))
   (np ((root $var2) (cat np)))
  ))
```

NOTE: Once you use any immediate constituents in an entry, word order is fixed. So you might wonder, why not just use all immediate constituents and completely forget about syntactic function labels? Well, you can if you want to. But there actually is some benefit in, e.g., labeling a subject as *subject* rather than just *np*, so we suggest that you do that to the extent possible. The current lexicon is inconsistent in this matter.

n

Among the instances in which *n*, as opposed to *np*, is used are

1. in noun-noun compounds, to indicate the nominal components. For example, if we wanted to list the noun-noun compound *salt deposit* under the head word *deposit* (we could also give it its own head entry), we write:

```
(syn-struct
  ((n ((root $var1) (cat n) (root salt)))
   (root $var0) (cat n)
  ))
```

A synonymous writing convention for n-n compounds is:

```
(syn-struct
  ((n ((root $var1) (cat n) (root salt)))
   (n ((root $var0) (cat n))
  ))
```

2. in adjectival entries, to indicate the noun that an adjective modifies, as in *tempermental person* (entry: tempermental)

```
(syn-struct
  ((mods ((root $var0) (cat adj)))
   (root $var1) (cat n)
  ))
```

or a *student* able to finish his work on time (entry: able)

```
(syn-struct
  ((n ((root $var1) (cat n)))
   (adj ((root $var0) (cat adj)))
   (xcomp ((root $var2) (cat v))))
))
```

adj

adjective or adjective phrase

This scheme proved (to be) **ridiculous** (entry: prove)

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (inf ((root to) (root $var4) (cat inf)))
   (v ((root $var2) (cat v) (root be) (form infinitive) (opt +)))
   (adj ((root $var3) (cat adj))
   ))
```

Note:

Be is a special case when it comes to being explicitly referred to in a lexicon entry. In some cases, *be* means only and precisely *be* (or any of its inflectional forms), as above. In other cases, however, *be* can actually be replaced by *seem*, *seem to be*, *appear*, *appear to be* (and perhaps some others). In this case, one writes (*root *be**) in the syn-struct, to indicate expanded *be*. For example, the following syn-struct is from a sense of the adjective *adept* that covers the structures like *He is <seems, seems to be, appears, etc.> adept at skiing*.

```
(syn-struct
  ((np ((root $var1) (cat np)))
   (v ((root $var2) (cat v) (root *be*)))
   (adj ((root $var0) (cat adj)))
   (prep ((root $var3) (cat prep) (root at)))
   (np ((root $var4) (cat np)))
   ))
```

adv

adverb or adverb phrase

They made **away/off** with the loot (entry: make)

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (adv ((root (or away off)) (root $var2) (cat adv))
   (pp
    ((root with) (root $var3) (cat prep)
     (obj
      ((root $var4) (cat n))))))
   ))
```

))

Note 1. The notation for expressing variants is, as above, (or x y).

Note 2. We consider the category of the word *not* to be adv simply because we have to give it some category name, *Not* is actually a singleton grammatically.

v

verb

This refers only to the verb itself (in any of its inflected forms), not to any of its arguments. Our system has no vp level. When one wants to refer to a verb plus its internal arguments, one can use the syntactic function labels xcomp or comp (described above) or the immediate constituent inf-cl (described below).

When describing a verb, its syntactic form is important. The possible forms are listed below:

1. verb in any inflected form

E.g., in a lexicon sense for **tell** that covers the structure

[subject] + [can] + [tell] + [complement]

as in *I can tell that he's happy*, we need to indicate that the auxiliary *can* is a required part of this structure. Since *can* can be in any inflected form (*could*, *could have...*), we write it as follows, with no restrictions. (We do, however, restrict the form of *tell* which follows it, as described below.)

```
(syn-struct
  ((np ((root $var1) (cat np)))
   (v ((root $var2) (cat v) (root can)))
   (root $var0) (cat v) (form infinitive)
   (comp ((root $var3) (cat v)))
  ))
```

2. progressive (i.e., -ing) form of the verb

E.g., in a lexicon sense for **help** that covers the structure

[subject] + [can] + [not] + [help] + [-ing] + [complement]

as in *He could not help **thinking** that it was a bad idea*, the verb that follows *help* (here: *thinking*) must be in the progressive form, so we add the descriptor (*form progressive*).

```
(syn-struct
```

```

((subject ((root $var1) (cat n)))
 (v ((root $var2) (cat v) (root can)))
 (verb-neg ((root $var5) (cat verb-neg)))
 (root $var0) (cat v) (form infinitive)
 (v ((root $var3) (cat v) (form progressive)))
 (comp ((root $var4) (cat v) (opt +)))
 ))

```

3. the bare infinitive (without ‘to’) form of the verb

E.g., in a lexicon sense for **help** that covers the structure

[subject] + [can] + [not] + [help] + [-ing] + [complement]

as in *He could not **help** thinking (to himself) that it was a bad idea*, the verb *help* must be in the bare form, so we add the descriptor (form infinitive).

```

(syn-struct
 ((subject ((root $var1) (cat n)))
 (v ((root $var2) (cat v) (root can)))
 (verb-neg ((root $var5) (cat verb-neg)))
 (root $var0) (cat v) (form infinitive)
 (v ((root $var3) (cat v) (form progressive)))
 (pp
 ((root to) (root $var6) (cat prep) (opt +)
 (obj ((root $var7) (cat n)))))
 (comp ((root $var4) (cat v) (opt +)))
 ))

```

4. the past participle of the verb

E.g. in the lexicon entry for *design* we have a sense to cover the construction

[subject] + [*be*] + [designed] + [infinitive]

as in *This is (seems to be, etc.) designed to expediate work*. The syn-struct describes the verb form as a past-participle:

```

(syn-struct
 ((subject ((root $var1) (cat n)))
 (v ((root $var2) (cat v) (root *be*)))
 (root $var0) (cat v) (form past-participle)
 (xcomp ((root $var3) (cat v)))
 ))

```

inf-cl

Infinitival clause, that is, the infinitival form of the verb, excluding ‘to’, plus any of the verb’s internal arguments.

For example, in a lexicon sense for *make* that covers the structure

[subject] + [make] + [indirect object] + [infinitival complement]

as in *He made her go*, ‘go’ is an inf-cl. To reiterate, when you use inf-cl, all internal arguments of the verb are understood as potentially present: e.g., *He made her go to the store* would also be covered by this structure.

```
(syn-struct
  ((np ((root $var1) (cat np)))
   (root $var0) (cat v)
   (np ((root $var2) (cat np)))
   (inf-cl ((root $var3) (cat v)))
  ))
```

inf

the infinitival word ‘to’ by itself

E.g., in a lexicon sense for *help* that covers the structure

[subject] + [help] + [us] + ([to]) + [infinitival complement]

as in *They helped us (to) write the paper*, the optionality of *to* can conveniently be expressed by expressing *to* and the infinitival complement separately, and making the *to* optional.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (indirectobject ((root $var2) (cat n)))
   (inf ((root to) (root $var4) (cat inf) (opt +)))
   (inf-cl ((root $var3) (cat inf-cl)))
  ))
```

(If we were to express *to write the paper* using *xcomp*, we could not indicate the optionality of *to*.)

verb-neg

verb-neg describes the word *not* when it negates a verb.

For example, in the lexicon sense for *help* that covers the structure *He could not help thinking (to himself) that it was a bad idea*, we write:

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (v ((root $var2) (cat v) (root can)))
   (verb-neg ((root $var5) (cat verb-neg)))
   (root $var0) (cat v) (form infinitive)
   (v ((root $var3) (cat v) (form progressive)))
   (pp
```

```

((root to) (root $var6) (cat prep) (opt +)
 (obj ((root $var7) (cat n))))
(comp ((root $var4) (cat v) (opt +)))
))

```

cl

finite clause – i.e., a verb with all its arguments

E.g., the sense of *ago* that covers structures like *Three years ago* (*,*) *I finished school* will be

```

(SYN-STRUC
 (n ((root $var1) (cat n)))
 (adj ((root $var0) (cat adj)))
 (punct ((root *comma*) (root $var3) (cat punct) (opt +)))
 (cl ((root $var2) (cat cl))))
))

```

punct

Punct is used as an immediate constituent for a punctuation mark. If you want to refer to any punctuation mark (without discrimination), the description will look as follows:

```
(punct ((root $var3) (cat punct)))
```

If you want to specify a particular punctuation mark, you can indicate it as the root, with its name surrounded by asterisks. The inventory of legal punctuation marks (so far) is:

comma *semi-colon* *dash* *colon* *open-paren* *close-paren*

We do not yet deal with sentential punctuation, though we will soon. An example of punctuation, using the same sense of *ago* as above (*Three years ago*(*,*) *I finished school*), is as follows:

```

(SYN-STRUC
 (n ((root $var1) (cat n)))
 (adj ((root $var0) (cat adj)))
 (punct ((root *comma*) (root $var3) (cat punct) (opt +)))
 (cl ((root $var2) (cat cl))))
))

```

conj

conjunction

E.g., the lexicon sense of *regard* that covers structures like *He regarded it as a waste of time* will have the following syn-struct

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (directobject ((root $var2) (cat n)))
   (conj ((root $var3) (cat conj) (root as)))
   (np ((root $var4) (cat np)))
  ))
```

The syntactic structure of the final np in this structure is not among our basic inventory of syntactic functions – that is why we simply call it *np*. We don't want to expand the inventory of syntactic function labels for such cases, since it does not buy us anything for processing.

content-marker

This is a label for the word *that* as a conjunction. It is used rarely, only in cases like the following, where *that* needs to be expressed outside of the subordinate clause in which it typically occurs (when *that* is used within the subordinate clause, it is part of the constituent *comp*).

For example, in the following sense of *require*, we want to express that *require* does not take a typical complement (*comp*), **He required that I went*, but instead that it takes an atypical series of elements:

[require] + [that] + [subject] + [bare form of verb]

We write out each of these elements separately.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (content-marker ((root $var2) (cat content-marker) (root that)))
   (np ((root $var3) (cat np)))
   (inf-cl ((root $var4) (cat v)))
  ))
```

num

num indicates a number, as in the following sense of *serve* that covers examples like *He served 10 years in/at the local prison*.

```
(syn-struct
  ((subject ((root $var1) (cat n)))
   (root $var0) (cat v)
   (num ((root $var2) (cat num)))
   (n ((root $var3) (cat n)))
   (pp
    ((root (or in at)) (root $var4) (cat prep) (opt +))
  ))
```

```
(obj ((root $var5) (cat n))))
))
```

The Processing of Syntactic Function Categories vs. Immediate Constituents

For orientation, it might help to understand how the program processes syntactic-function categories versus immediate constituents in syn-strucs.

If the syn-struc contains only syntactic-function elements, a function is called to get the default orderings, which are:

subject verb indirectobject directobject pp comp xcomp

If there is an indirectobject, the following is also returned

subject verb directobject pp("to" + IO) pp comp xcomp

There are also multiple rules that permit all pps to occur in any order.

Once the ordering possibilities are established, the syntactic-function categories (like subject) are converted into their respective immediate constituents (like np). The inventory of conversions is:

Syntactic Function Category	Immediate Constituent
subject	np
directobject	np
indirectobject	np and pp (with 'to' as its head)
xcomp	inf + inf-cl
comp	(content-marker) + cl
pp	prep + np

If the syn-struc contains any immediate constituents, then all categories are immediately converted to their immediate constituent equivalents, with the listed order of elements being considered fixed.

Inventory of Features for Each Part of Speech

Different parts of speech in different languages can have different features: e.g., an English noun like *child* be in the possessive form (*child's*), in the plural (*children*), or in the possessive plural (*childrens'*). Features like these can be used as descriptors in the sem-struc, as applicable. **They need only be indicated when their values are in some**

way restricted: e.g., if a noun can be in the singular or the plural, no indication of number is indicated.

The inventory of features for each part of speech in English, along with examples, is presented below. The category names and values are presented in the form they will be used in the syn-struct: e.g., *plural* is written as *pl* and is thus presented. Brackets are used to describe abbreviations, as necessary.

Some features are not used much, if at all, in the lexicon but are used when the analyzer builds a text-meaning representation (TMR) from real input text. Some are listed in this section, and some in the next section, for the sake of completeness. For example, there is usually no reason why a noun phrase that is acting as a verbal argument should be restricted in number: e.g., the subject of *give* in its ditransitive usage can be *I, you, he, my mother, her brothers*, etc. While no restriction on the subject will be listed in the lexical entry for *give*, the TMR produced from the processing of the sentence *My mother gave me a book* will include the fact that *my mother* is (cat n) (person third) (number singular).

Noun Features

<u>feature</u>	<u>values</u>	
number	sing pl	
person	first second third	
type	pn pro dem-pro interrog-pro poss-pro	[proper noun] [pronoun] [demonstrative pronoun] [interrogative pronoun] [possessive pronoun]
possessive	+	[means that an np is in the possessive form]

NOTE: pronouns are (cat n) (type pro)

greens – pl only – meaning ‘leafy vegetables’ [entry: green]

```
(syn-struct
  ((root $var0) (cat n) (number pl)
  ))
```

it is (was, seems to be, etc.) a foregone conclusion that... [entry: foregone-conclusion]

```
(syn-struct
  ((subject ((root $var1) (root it) (cat np) (type pro)))
   (v ((root $var2) (root *be*) (cat v)))
   (np ((root $var0) (cat n)))
   (comp ((root $var3) (cat v)))
  ))
```

A few more words are needed about the use of (number pl). The analyzer's interpretation of (number pl) depends on whether it is used to describe \$var0 or some other variable in the entry.

1) If (number pl) describes \$var0, it means that \$var0 (i.e., the head word of the entry) is, itself, plural and requires plural agreement. So the entry for 'pants' ('slacks' etc.) will be:

```
(pants
 (pants-n1
  (cat n)
  (synonyms "trousers" "slacks")
  (anno
   (def "")
   (ex "")
   (comments ""))
 )
 (syn-struct
  ((root $var0) (cat n) (number pl))
 )
 (sem-struct
  (trousers))))
```

This means that the preprocessor will search for specifically the plural form of these words and, when found, will mark them as (number pl) without any further morphological processing necessary. We want to have senses like this a) for words that don't have a singular counterpart at all (e.g., *pant); and, b) for words that don't have a singular counterpart in the given meaning and therefore are better listed as plural in the given sense. An example of the latter is 'atmospheric conditions', in which 'condition' must be in the plural in the phrase.

```
(atmospheric_conditions
 (atmospheric_conditions-n1
  (cat n)
  (anno
   (def "")
   (ex "")
   (comments "")))
```

```
(syn-struct
  ((root $var0) (cat n) (number pl)))
(sem-struct
  (weather))))
```

2) If (number pl) describes some variable other than \$var0, it means that the given word must be in the plural form to be acceptable in the entry. It can get in the plural form in one of two ways: a) because there is a lexical sense in which it is listed in its plural form (like pants above); b) because it is productively made plural during processing. Here are some examples:

The quantifier 'a few' requires a plural complement: in analysis, the analyzer must check to be sure there is a plural complement, and in generation, it must generate a plural complement.

```
(a_few-quantifier1
  (cat quantifier)
  (anno
    (def ""))
    (ex "a few people like him.")
    (comments "the complement is a 'n' since it can't have a determiner or article.")
  )
  (syn-struct
    ((root $var0) (cat quantifier)
      (n ((root $var1) (cat n) (number pl))))
    ))
  (sem-struct
    (set
      (cardinality (<> 2 5))
      (indeterminate yes)
      (member-type (value ^$var1)))
    ))
```

Similarly, the word 'interest' must be in the plural in order for this phrasal to be used (be it in analysis or generation).

```
(affect-v2
  (cat v)
  (anno
    (def "in phrasal 'affect (company's) interests'")
    (ex ""))
    (comments ""))
  )
  (syn-struct
    ((subject ((root $var1) (cat n)))
      (root $var0))
```

```

(cat v)
(directobject ((root $var2) (cat n) (root interest) (number pl)))
))
(sem-struct
(change-event
(agent (value ^$var1))
(theme asset))
(^$var2 (null-sem +))
))

```

Verb Features

<u>feature</u>	<u>values</u>
form	progressive past-participle participle infinitive (i.e., bare form, without 'to')
tense	present past future
number	sing pl
person	first second third

Adjective and Adverb Features

<u>feature</u>	<u>values</u>
degree	comparative superlative

The positive degree is the default and need not be indicated.

Number Features

<u>feature</u>	<u>values</u>
value	[some number]

The number feature ‘value’ is typically used in TMRs, not in lexicon entries. However, if you wanted to make a phrasal lexicon entry for *2 heads are better than one*, the 2 would be described as (num ((root \$var1) (cat num) (value two))).

Features Used (almost) Exclusively in TMRs

Among the features listed above, some are not commonly listed in the syn-structs of lexicon entries (e.g., number). Still other features are virtually only used in TMRs or in the onomasticon (lexicon of proper names). These include the following:

Features of Proper Nouns of Type PERSON

For noun phrases in actual texts that are assigned (pn-type PERSON) by the preprocessor, the following features should be filled in, as possible, from the text:

TITLE
 PREFIX
 FIRST-NAME
 MIDDLE-INITIAL
 MIDDLE-NAME
 LAST-NAME
 SUFFIX

For example, the text input *Mr. John H. Smith, D.D.S.* should return:

PREFIX	Mr.
FIRST-NAME	John
MIDDLE-INITIAL	H.
LAST-NAME	Smith
SUFFIX	D.D.S.

Features of Proper Nouns of Type LOCATION

For noun phrases in actual texts that are assigned (pn-type LOCATION) by the preprocessor, the following features should be filled in, as possible, from the text:

CITY
 STATE
 COUNTRY
 ZIP

For example, the text input *Marbury, Conn.* should return:

CITY	Marbury
STATE	Connecticut

Features of Proper Nouns of Type COMPANY

For noun phrases in actual texts that are assigned (pn-type COMPANY) by the preprocessor, the following features should be filled in, as possible, from the text:

NAME
SUFFIX

For example, the text input *XYZ, Corp.* should return:

NAME	XYZ
SUFFIX	Corp.

Features of DATES

For noun phrases in actual texts that are assigned (pn-type COMPANY) by the preprocessor, the following features should be filled in, as possible, from the text:

YEAR
DAY
MONTH

For example, the text input *December 23, 1923* should return:

YEAR	1923
DAY	23
MONTH	12

Features of TIMEs

For noun phrases in actual texts that are assigned (pn-type COMPANY) by the preprocessor, the following features should be filled in, as possible, from the text:

HOUR (in 24-hour format)
MINUTE
SECONDS

For example, the text input *6:05 p.m.* should return:

HOUR	18
MINUTE	5

Output-syntax and tmr-head

The descriptors **output-syntax** and **tmr-head** are used (outside of any specific 'zone' – written at the same indentation level as 'sem-struc' and 'syn-struc') when it is possible that the analyzer will have difficulty determining what the syntactic or semantic (i.e., tmr) head is. A good rule of thumb is that when you use immediate constituents in a sense, you should include these descriptors.

The syntactic head indicates the overall category of everything in the syn-struct; that is, it gives the construction a syntactic label. E.g., if one composes a syn-struct of [adj] [n] [n] [pp] [xcomp], the analyzer cannot guess what the head is supposed to be so it must be told.

The tmr-head indicates what the tmr – i.e., the semantic representation – should be headed by. This is needed in case the whole thing is modified by something else, in which case the analyzer needs to know where to attach the modification.

If you think that one of these labels is necessary, then include both. If in doubt, include both since it will only add a bit of redundancy if it is not expressly needed (or check for similar English samples that are already done). The following is an example of when and how these labels are used:

```
(albeit-conj1
  (CAT conj)
  (MORPH )
  (ANNO
    (DEF "'albeit' conjoining Adjs")
    (EX "gentle albeit cold breeze")
    (COMMENT ""))

  (syn-struct
    ((adj ((root $var1) (cat adj)))
     (root $var0) (cat conj)
     (adj ((root $var2) (cat adj)))
     (n ((root $var3) (cat n)))
    ))

  (output-syntax n)
  (tmr-head $var3)

  (sem-struct
    (apply-meaning (value ^$var1) (value ^$var3))
    (apply-meaning (value ^$var2) (value ^$var3))
  )
)
```

Miscellaneous

At some point we were writing separate senses for each adjective for its prenominal and predicative uses. However, as of 8/03 we decided that there would be a rule such that if we wrote only the prenominal sense, the predicative sense would be understood as well. There are many adjectival entries in the lexicon now that have both senses, and it's still not incorrect to write both. Since we're mainly analyzing now and not generating, we don't need to specify whether some adjectives have only predicative usage – but eventually that may become necessary.