Lkit(v2.x) User Guide

Rule Structure

The structure of grammar rules and lexicon entries are modifiable to suit the preferences of the user. The mechanism for doing this is discussed in the section dealing with the programmable interface. For the purposes of all examples in this section the structure of grammar rules and lexicon entries is as shown in figures A4.1and A4.2, this form is the *raw* form as described elsewhere.

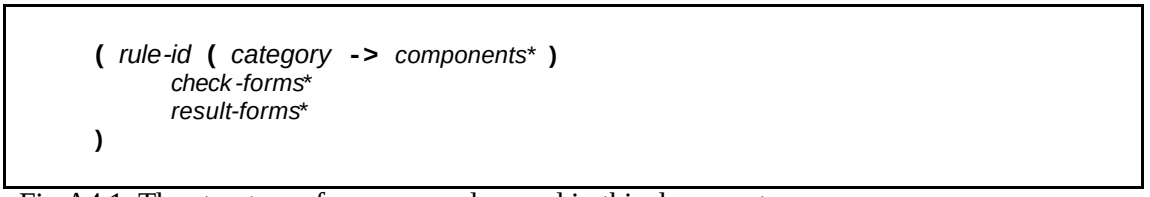


Fig A4.1. The structure of grammar rules used in this document.

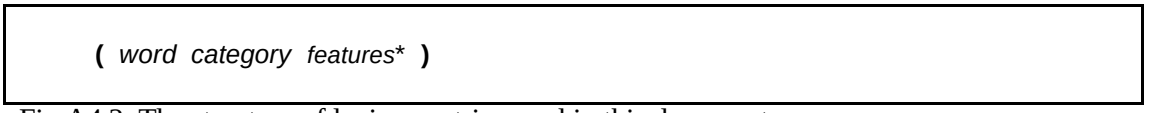


Fig A4.2. The structure of lexicon entries used in this document.

Grammars and lexicons are constructed by users in a textual form then submitted for compilation using `build-grammar` and `build-lexicon` respectively. In the development phase the parser is called with its goal (the syntactic category it will try to build) and a sentence. The sentence is expressed as a list. For example the expression

```
(parse 'noun-phrase '(the large black dog))
```

initiates the parser to attempt to construct a noun phrase from the words "a large black dog". The parser always uses the most recently compiled grammar & lexicon.

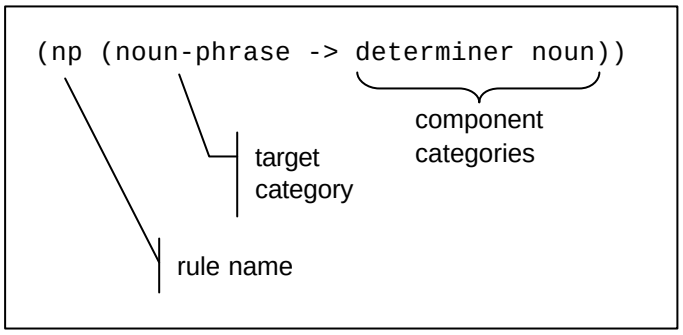
In the section that follows some lexical entries and parts of grammar rules are expressed as structures using the lisp backquote macro with comma and "@". This syntax is part of the Common Lisp standard and is not explained here (a brief introduction is provided in Appendix 5).

Expressing Syntax

The simplest specification of a lexicon is as a list of words with their respective syntactic categories. Words with two or more syntactic definitions should have multiple entries in the lexicon.

```
(build-lexicon
  `((a      determiner )
    (cat    noun       )
    (chased verb       )
    (dog    noun       )
    (the    determiner )
    ))
```

A grammar is expressed as a list of rules where each rule has a name, a target syntactic category and one or more component categories. For example:



An example grammar:

```
(build-grammar
  '((s1 (sentence -> noun-phrase verb-phrase))
    (np (noun-phrase -> determiner noun))
    (vp (verb-phrase -> verb noun-phrase))
    ))
```

The different parts of the output from the parser are explained part by part in this section. Using the grammar and lexicon specified above the output will consist of a top line containing the target category, the rule name which produced this category and the words contained in it. The next section of output is headed "Syntax", this is a representation of the phrase structure of the sentence. The final section is headed "Semantics", this is discussed later. Note also that `parse` is a simple function interface with the parser. It always returns T even in the event of a failed parse. Other values will only be returned in the event of system error an example of which could occur if users called their own (faulty) code within a grammar rule.

Note that all interaction with Lkit is indicated by a vertical bar in the left margin and input is prefixed by a > symbol.

```
> (parse 'sentence '(the dog chased a cat))

_____________
complete-edge 0 5 s1 sentence (the dog chased a cat) nil
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence
   (noun-phrase (determiner the) (noun dog))
   (verb-phrase
      (verb chased)
      (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence)
t
```

Expressing Semantics

Following a word's syntactic category in its lexical entry a user may include any symbol or list of symbols. The primary use for this is to express features of the word such as its semantics or its syntactic dependencies.

```
(build-lexicon
  `((a      determiner any     )
    (cat    noun       feline  )
    (chased verb       hunts   )
    (dog    noun       canine  )
    (the    determiner specific)
    ))
```

Grammar rules may access the additional information contained in the lexicon and use it to build structures. These structures are specified inside grammar rules by grammar builders. They can be of arbitrary complexity. This facility allows rules to construct semantic representations derived from their component parts (the approach works consistently whether their components are terminals, relating to lexical entries, or non-terminals relating to structures formed by other rules).

For example the rule:

```
(vp (verb-phrase -> verb noun-phrase)
        (action . verb)
        (object . noun-phrase)
        )
```

states that the result of a completed verb-phrase is a structure containing two slots called *action* and *object*. The value of the *action* slot is the semantics of the verb. The verb is a terminal symbol so this value is taken from the verbs lexical entry. The value of the *object* slot is the semantics of the noun-phrase which will have been formed from application of a noun-phrase rule. The default semantic structure of Lkit v2 is a slot filler notation and version 2 of Lkit provides tools for manufacturing this notation.

The syntax `(action . verb)` can be read as *create a new slot with the name 'action' and a value which results from retrieving the semantics of the verb form.*

This syntax can be expanded by listing subcategories (or more accurately sub-slots) as well as categories as in: `(action . verb-phrase.action)` which reads *retrieve the semantics of the action slot of the verb-phrase as the value of this new slot.* Slots can be nested indefinitely in this way. Note that the two "." (full-stop/period) characters have a different function. The first, which is surrounded by space characters (and may be familiar to Lisp users) is used as a slot constructor. The second (which must not be pre- or post-fixed with a space character) indicates a sub-category, its use in Lkit2 is analogous to its use in C++ or Java where the "dot" operator is used to access member/instance variables of structures and classes.

The grammar which follows illustrates the use of these forms.

```
(build-grammar
  '((s1 (sentence -> noun-phrase verb-phrase)
        (actor  . noun-phrase)
        (action . verb-phrase.action)
        (object . verb-phrase.object)
        )
    (np (noun-phrase -> determiner noun)
        (determiner . noun)
        )
    (vp (verb-phrase -> verb noun-phrase)
        (action . verb)
        (object . noun-phrase)
        )
    ))
```

The result of a parse now includes a semantic representation formed from the application of the semantic components of the grammar rules.

```
> (parse 'sentence '(the dog chased a cat))
_____________
complete-edge 0 5 s1 sentence (the dog chased a cat) nil
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner the) (noun dog))
          (verb-phrase (verb chased)
          (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (actor (specific canine))
          (action hunts)
          (object (any feline)))
t
```

Checks, Fails & Glitches

There are many situations in language use where the semantic nature of one word in a sentence influences the interpretation of another word. As an example of this consider the meaning of the word "report" in the following sentences:

1. He heard her report on the conflict.
2. He read her report on the conflict.

In practice there is also a need to reject sentences for being ill-formed when the rules for a word's legitimate use are violated as in:

*Colourless green ideas sleep furiously.

A more common situation with simple grammars for English is the need to check for agreement between the use of adjectives and the nouns they qualify and for numeric agreement between subject and verb. The latter case is illustrated by the following:

1. Cats eat mice.
2. *Cat eat mice.

The structure of grammar rules allow linguists to specify checks and act on them in one of three ways:

1. perform some operation detailed in the Lisp code written as part of the grammar rule
2. signal that the edge[1] has failed and that the parser should not investigate it further
3. mark the edge as ill-formed but allow the parser to continue with it

The following example examines this by enforcing checks for numeric agreement between the subject and verb. To achieve this it is necessary for the lexical entries of words to contain numeric information as well as their syntactic and semantic details.

A simple slot-filler notation is used for all *features* of a lexical entry, eg:

```
(cat noun (number . singular) (semantics . feline))
```

Lkit provides primitives to manipulate these slot-filler notations which are in the form of generalised association lists - association lists whose association values can themselves be association lists and whose association names need not be symbolic. While Lkit recommends this structure and provides primitives for its manipulation users are free to use any representation they choose. Users may use any of their own lisp functions or macros from within grammar rules so are free to develop representations as they choose.

In the following examples the slot name semantics is abbreviated to *sems* and values singular and plural are abbreviated *sing* and *plur* respectively.

```
(build-lexicon
  `((a      determiner (sems . any)    )
    (all    determiner (sems . every)  )
    (cat    noun       (sems . feline)   (number . sing))
    (cats   noun       (sems . feline)   (number . plur))
    (chase  verb       (sems . hunts)    (number . plur))
    (chases verb       (sems . hunts)    (number . sing))
    (dog    noun       (sems . canine)   (number . sing))
    (dogs   noun       (sems . canine)   (number . plur))
    (the    determiner (sems . specific))
    ))
```

[1] An edge is some formed or partially-formed phrase with or without valid semantics, as such it is the fundamental unit of parsing.

The grammar below builds on the previous example grammar but introduces a new slot at the sentence level which indicates the state of numeric agreement.

```
(build-grammar
  ; comments start with a semi-colon
  ; this grammar ignores numeric agreement with determiners
  '((s1 (sentence -> noun-phrase verb-phrase)
      (actor  . noun-phrase.sems)
      (action . verb-phrase.action)
      (object . verb-phrase.object)
      ; check number of noun-phrase & verb-phrase agree
      (if (noun-phrase.number = verb-phrase.number)
          numeric-agreement-ok
          numeric-agreement-bad
          )
      )
    (np (noun-phrase -> determiner noun)
      (number . noun.number)
      (sems   . (determiner.sems noun.sems))
      )
    (vp (verb-phrase -> verb noun-phrase)
      (action . verb.sems)
      (object . noun-phrase.sems)
      (number . verb.number)
      )))
```

The first example using this grammar is of a numerically valid sentence.

```
> (parse 'sentence '(the dog chases a cat))
_____________
complete-edge 0 5 s1 sentence (the dog chases a cat) nil
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner the) (noun dog))
          (verb-phrase (verb chases)
                       (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (actor specific canine)
          (action . hunts)
          (object any feline)
          numeric-agreement-ok)
t
```

This next example is using a sentence which violates numeric agreement.

```
> (parse 'sentence '(the dogs chases a cat))
_____________
complete-edge 0 5 s1 sentence (the dogs chases a cat) nil
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner the) (noun dogs))
          (verb-phrase (verb chases)
                       (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (actor specific canine)
          (action . hunts)
          (object any feline)
          numeric-agreement-bad)
t
```

Notes:
1. the use of the **if** form for this kind of activity is not recommend, **fail** and/or **glitch** forms (see below) are considered more suitable;
2. this **if** form is not the same as the **if** form in Lisp, for more sophisticated conditional forms users should break into Lisp.

Fails

`fail if` and **`fail if not`** are primitives used to halt the processing of a rule. In the following example **`fail if`** is used to prevent processing of numerically ill-formed sentences.

```
(build-grammar
 '((s1 (sentence -> noun-phrase verb-phrase)
       (fail if noun-phrase.number /= verb-phrase.number)
       (actor  . noun-phrase.sems)
       (action . verb-phrase.action)
       (object . verb-phrase.object)
       )
   (np (noun-phrase -> determiner noun)
       (number . noun.number)
       (sems   . (determiner.sems noun.sems))
       )
   (vp (verb-phrase -> verb noun-phrase)
       (action . verb.sems)
       (object . noun-phrase.sems)
       (number . verb.number)
       )
   ))
```

The first example below uses a valid sentence, the second uses an ill-formed sentence. Note that PARSE returns T even when no valid parses are found. PARSE only returns an error code (or nil) if some run-time error is cased by a grammar rule.

```
> (parse 'sentence '(the dog chases a cat))
_____________
complete-edge 0 5 s1 sentence (the dog chases a cat) nil
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner the) (noun dog))
          (verb-phrase (verb chases)
          (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (actor specific canine)
          (action . hunts)
          (object any feline))
t

> (parse 'sentence '(the dogs chases a cat))
t
```

Glitches

`glitch if` and **`glitch if not`** have a similar use to **`fail if`** and **`fail if not`** but instead of halting the processing of a rule or sub-phrase they cause it to be marked which allows parsing to proceed. This is useful when dealing with situations that are grammatically flawed but not so serious that semantics are unrecoverable. In many cases there are multiple (potentially valid) parses, choosing between them can be aided by glitch markers. The following grammar marks numerically ill-formed sentences with the symbol *`numeric-agreement`*.

```
(build-grammar
 '((s1 (sentence -> noun-phrase verb-phrase)
       (glitch numeric-agreement
           if not noun-phrase.number = verb-phrase.number)
       (actor  . noun-phrase.sems)
       (action . verb-phrase.action)
       (object . verb-phrase.object)
       )
   (np (noun-phrase -> determiner noun)
       (number . noun.number)
       (sems   . (determiner.sems noun.sems))
       )
   (vp (verb-phrase -> verb noun-phrase)
       (action . verb.sems)
       (object . noun-phrase.sems)
       (number . verb.number)
       )
   ))
```

```
> (parse 'sentence '(the dogs chases a cat))
_____________
complete-edge 0 5 s1 sentence (the dogs chases a cat) nil
     Glitches: (numeric-agreement)
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner the) (noun dogs))
          (verb-phrase (verb chases)
                       (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (actor specific canine)
          (action . hunts)
          (object any feline))
t
```

Multiple glitches may be generated and typically will occur at different places in the phrase structure this is illustrated in the next example. The lexicon and grammar for this example extend the idea of numeric agreement by considering agreement between a determiner (words like *the*, *a*, *all*, etc) and its noun. They also allow words to have multiple categories (*the* for example can be both singular and plural). To make this possible the lexicon entries for number tags are all lists. Lisp programmers may find the syntax slightly surprising: why use the "." constructor followed by a bracketed list? That syntax is used here to be in keeping with previous examples, standard Lisp syntax works ok.

```
(build-lexicon
  `((a      determiner (sems . any)      (number . (sing)))
    (all    determiner (sems . every)    (number . (plur)))
    (cat    noun       (sems . feline)   (number . (sing)))
    (cats   noun       (sems . feline)   (number . (plur)))
    (chase  verb       (sems . hunts)    (number . (plur)))
    (chases verb       (sems . hunts)    (number . (sing)))
    (dog    noun       (sems . canine)   (number . (sing)))
    (dogs   noun       (sems . canine)   (number . (plur)))
    (the    determiner (sems . specific) (number . (sing plur)))
    ))
```

The new grammar checks for an intersection of numeric tags (instead of equality) this is done using the **$*** intersection operator, an empty (null) intersection set is considered taken as a false condition for glitches and fails (see docs\utils\sets for additional information on set operators: set-union, set-difference, etc).

```
(build-grammar
 '((s1 (sentence -> noun-phrase verb-phrase)
       (glitch (subj-verb number)
               if not noun-phrase.number $* verb-phrase.number)
       (actor  . noun-phrase.sems)
       (action . verb-phrase.action)
       (object . verb-phrase.object)
       )
   (np (noun-phrase -> determiner noun)
       (glitch (det-noun number)
               if not determiner.number $* noun.number)
       (number . noun.number)
       (sems   . (determiner.sems noun.sems))
       )
   (vp (verb-phrase -> verb noun-phrase)
       (action . verb.sems)
       (object . noun-phrase.sems)
       (number . verb.number)
       )
   ))
```

```
>  (parse 'sentence '(a dogs chases a cat))
_____________
complete-edge 0 5 s1 sentence (a dogs chases a cat) nil
     Glitches: ((subj-verb number) (det-noun number))
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner a) (noun dogs))
                       (verb-phrase (verb chases)
                       (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (actor any canine) (action . hunts) (object any feline))
t
```

Other Primitives for Grammar Rule Specification

Participation Conditions

Rules in Lkit can specify their components to have optional or multiple participation conditions and Lkit provides a specific syntax for stating this. This syntax is in the form of single character prefixes as shown in the table below.

Prefix	Meaning	Example
none	mandatory, single	NounPhrase
?	optional, single	?Determiner
*	optional, multiple	*Adjective
+	mandatory, multiple	+PrepositionPhrase

An example of a rule (with no semantics) which takes advantage of these facilities is a rule which states that a NounPhrase is formed by an optional Determiner, optionally followed by any number of Adjectives and then a Noun. This rule is specified as follows (the symbol "`np1`" is the rule name):

```
(np1 (Noun-Phrase -> ?Determiner *Adjective Noun))
```

Without using participation conditions this structure would have to be described by multiple rules, for example:

```
(np1 (Noun-Phrase -> Noun-Group))
(np2 (Noun-Phrase -> Determiner Noun-Group))
(ng1 (Noun-Group -> Noun))
(ng2 (Noun-Group -> Adjective Noun))
```

Facilities are provided for processing the semantics of components with optional or multiple participation conditions. The example below uses **`if`** as a conditional and the **`*`** iterator for optional and multiple items. The highlighted line **`(if determiner ...)`** checks for the existence of a determiner then builds the appropriate slot form. The line **`(qualifiers . *.adjective)`** dumps all adjective details into a slot named "qualifiers". Note that `IT=>` provides the current binding of any iterator. Additionally `ALL=>` (not shown) can be used in place of an iterator so an expression like...

```
(ALL=> noun-phrase adjective)
```

delivers all adjectives of `noun-phrase`.

This is demonstrated with a simplified lexicon as follows...

```
(build-lexicon
  `((a      determiner any     )
    (cat    noun       feline  )
    (chase  verb       hunts   )
    (dog    noun       canine  )
    (the    determiner specific)
    (black  adjective  (color black))
    (large  adjective  (size 7/10))
    (small  adjective  (size 3/10))
    ))
```

This example uses only one grammar rule, semantics are highlighted in the output.

```
(build-grammar
  '((np (noun-phrase -> ?determiner *adjective noun)
       (if determiner
           (quantification . determiner)
           (quantification undefined))
       (qualifiers . *.adjective)
       (object . noun)
         ))
 ))
```

```
> (parse 'noun-phrase '(small black dog))

_____________
complete-edge 0 3 np noun-phrase (small black dog) nil
np   noun-phrase -> (?determiner *adjective noun)
Syntax
(noun-phrase (adjective small) (adjective black) (noun dog))
Semantics
(noun-phrase
 (quantification undefined)
 (qualifiers ((size . 3/10)) ((color . black)))
 (object canine))
t
```

NB: qualifier semantics are nested in the example above (extra brackets around the *size* slot for example), this does not occur when using more realistic grammars because adjectives (like nouns etc) will have named slots in their lexical entries (see later examples).

```
> (parse 'noun-phrase '(the large dog))

_____________
complete-edge 0 3 np noun-phrase (the large dog) nil
np   noun-phrase -> (?determiner *adjective noun)
Syntax
(noun-phrase (determiner the) (adjective large) (noun dog))
Semantics
(noun-phrase
 (quantification specific)
 (qualifiers ((size . 7/10)))
 (object canine))
t
```

A4.5.2. !words

The prefix '!' is used to introduce a literal, a word which need not have an entry in the lexicon. Words without lexical entries are given default lexical definitions (these like all other system defaults are user modifiable). For example the grammar rule:

```
(eg1 (ConjNP -> NP !and NP))
```

states that a category of type ConjNP is formed by two NP categories separated by the word "and". The following grammar illustrates this (it uses the style of lexicon developed previously with separate associations for *number* and *semantics*).

```
(build-grammar
  '((s2 (sentence -> sentence !and sentence)
        (conjunction . *.sentence)
        )
    (s1 (sentence -> noun-phrase verb-phrase)
        (actor  . noun-phrase)
        (action . verb-phrase.action)
        (object . verb-phrase.object)
        )
    (np (noun-phrase -> ?determiner *adjective noun)
        (if determiner
            (quantification . determiner.sems)
            (quantification undefined))
        (qualifiers . *.adjective.sems )
        (object . noun.sems)
        )
    (vp (verb-phrase -> verb noun-phrase)
        (action . verb.sems)
        (object . noun-phrase)
        (number . verb.number)
        )))
```

```
> (parse 'sentence '(small dogs chase the small cats
                         and large dogs chase the large cats))

_____________
complete-edge 0 13 s2 sentence (small dogs chase the small cats and
                                large dogs chase the large cats) nil
s2   sentence -> (sentence !and sentence)
Syntax
(sentence
 (sentence
   (noun-phrase (adjective small) (noun dogs))
   (verb-phrase
      (verb chase)
      (noun-phrase (determiner the) (adjective small) (noun cats))))
 (undefined and)
 (sentence
   (noun-phrase (adjective large) (noun dogs))
   (verb-phrase
      (verb chase)
      (noun-phrase (determiner the) (adjective large) (noun cats)))))
Semantics
(sentence conjunction
 ((actor  (quantification undefined)
          (qualifiers (size . 3/10))
          (object . canine))
  (action . hunts)
  (object (quantification . specific)
          (qualifiers (size . 3/10))
          (object . feline)))
 ((actor  (quantification undefined)
          (qualifiers (size . 7/10))
          (object . canine))
  (action . hunts)
  (object (quantification . specific)
          (qualifiers (size . 7/10))
          (object . feline))))
t
```

A4.5.3. The Lisp Tag

Note: (ref: sl-021210)
[SL, 11 Dec 2002] the Lisp tag is currently under re-write with the Lkit-v.2 rule interface & is buggy with foo & applyfoo. This will be resolved soon.

The semantic transformations associated with grammar rules can contain any legal Lisp code if they are preceded by a **lisp** tag, similarly definitions in the lexicon can contain any Lisp structure. This allows linguists to specify lambda expressions as an elegant way to build nested constructs. The following example uses an Lkit2 facility to apply a pseudo lisp lambda, for Lkit users who are not also Lisp programmers this example may appear complex. It is provided here to illustrate an alternative grammar structure which makes use of the **lisp** tag.

```
(build-lexicon
  '((a      determiner (fn .(foo (x) (any x))))
    (cat    noun       feline  )
    (chased verb       (fn .(foo (y)
                                 (foo (x) (hunts x y)))))
    (dog    noun       canine  )
    (the    determiner (fn .(foo (x) (specific x))))
    ))

(build-grammar
  '((s1 (sentence -> noun-phrase verb-phrase)
      (lisp (applyfoo verb-phrase.fn noun-phrase)) )
    (np (noun-phrase -> determiner noun)
      (lisp (applyfoo determiner.fn noun)) )
    (vp (verb-phrase -> verb noun-phrase)
      (fn . (lisp (applyfoo verb.fn noun-phrase))) )
    ))
```

```
> (parse 'sentence '(the dog chased a cat))
_____________
complete-edge 0 5 s1 sentence (the dog chased a cat) nil
s1   sentence -> (noun-phrase verb-phrase)
Syntax
(sentence (noun-phrase (determiner the) (noun dog))
          (verb-phrase (verb chased)
                       (noun-phrase (determiner a) (noun cat))))
Semantics
(sentence (hunts (specific canine) (any feline)))
t
```

A4.5.4. Schema Variables

The '$' prefix introduces a static variable. Static variables can match with any syntactic category. for example:

```
(eg1 (daft -> !hello $X !goodbye))
```

This rule builds a "daft" category from any string of three words starting with "hello" and ending "goodbye". In this rule `(=> $X)` can be used to reference the semantics of the variable.

Variables may occur multiply in any rule schema but each variable may associate with only a single syntactic category in a given rule:

```
(eg2 (Conj -> $X !and $X))
```

States that a Conj is formed from any two sub-phrases *of the same category* separated by "and". Variables may also be used on the left hand side of a rule as in:

```
(eg3 ($X -> $X !and $X))
```

Rule eg3 states that any phrase type can be formed from two phrases of the same type connected by "and".

Note: (ref: lk2-021210a)
the Lkit-v.2 rule interface does not currently support multiple accessing of schema variables in the formation of semantic slots, ie: forms like **`*.$x.tense`** are not processed. Schema variables denoting syntactic categories (in a rule's component list) may be prefixed with **`*`** (eg: `strange-phrase -> (noun *$x ?$y)`) though grammar authors should use this feature with care as it can proliferate phrase forms which tend to be of little use.

A4.5.5. Using Numeric Indexing

When two or more of the same syntactic categories occur in a single rule (and their names are therefore ambiguous) Lkit allows numeric indexes to reference them though a better approach is to use aliases (see next subsection).

So using the grammar rule (defined using the Lkit-v.1 rule interface)...

```
(s2 (sentence -> sentence !and sentence)
  `((conjunction
     ,(=> 1)
     ,(=> 3)
  ))
```

`(=> 1)` refers to the first sentence component and `(=> 3)` the second.

Note: (ref: lk2-021210b)
the Lkit-v.2 rule interface does not support numeric indexing. Further there is no intention to support this feature in future releases. Numeric indexing reduces the readability of grammar rules and makes them more fragile during modification.

A4.5.6. Aliases

A better method for disambiguating category names (which is the only sensible method when categories have participation conditions) is to use aliases. Aliases are schema variables listed in brackets after a schema component as below.

```
(example-rule (X -> A($firstA) B A($secondA) ))
```

In this case **`$firstA`** refers to the first component of the resulting category X.

Note: (ref: lk2-021210c)
the note above (lk2-021210a) refers to schema variables introduced as aliases as well as those used as categories. The Lkit-v.2 rule interface does not currently support multiple accessing of them when building semantic slots.

A4.6. An Example

The following example illustrates some of the points discussed above. The lexicon, in particular, is too small & restricted to be of any practical use. Notice that the grammar is also very small yet can handle a reasonable variety of sentence forms. The lexicon and grammar use the concept of *case*. Some words have cases and others associate with certain cases. In the example below these cases are labelled **dest** for destination and **locn** for location.

```
(build-lexicon
 `((a      det   (sems . any)      (numb . (sing)))
   (all    det   (sems . every)    (numb . (plur)))
   (black  adj   (sems . (color . black)))
   (cat    noun  (sems . feline)   (numb . (sing)))
   (cats   noun  (sems . feline)   (numb . (plur)))
   (chase  verb  (sems . hunts)    (numb . (plur)) (cases . (dest)))
   (chases verb  (sems . hunts)    (numb . (sing)) (cases . (dest)))
   (dog    noun  (sems . canine)   (numb . (sing)))
   (dogs   noun  (sems . canine)   (numb . (plur)))
   (in     prep  (sems . *in*)     (case . locn))
   (into   prep  (sems . *in*)     (case . dest))
   (large  adj   (sems . (size  . 7/10)))
   (park   noun  (sems . place)    (numb . sing))
   (small  adj   (sems . (size  . 3/10)))
   (the    det   (sems . specific) (numb . (sing plur)))
   ))
```

To filter grammar rules on the basis of case information two variables are used called ***verb-only-cases***, which is a list of the cases which may only be used to qualify a verb, and ***verb-legal-cases***, which is a list of cases that are not excluded from qualifying verbs. Note: the use of an ***** character around the variable name is a lisp convention for global variables.

```
(defvar *verb-only-cases*)
(defvar *verb-legal-cases*)
(setf *verb-only-cases* '(dest))
(setf *verb-legal-cases* '(dest locn))
```

The grammar below uses the case information to collect case clauses (which occur as prepositional phrases) either at the noun-phrase level where they qualify noun objects or at the sentence level. Where such phrases could legitimately collect at either level it is preferable to collect them with sentences. To indicate this preference a glitch is signalled whenever they

collect with a noun-phrase. This could be used in later processing (not shown below) to reject nested clauses (in favour of un-nested alternatives) where multiple parses occur.

```
(build-grammar
 '((s2 (S -> S PP)
       (fail if not PP.case $<< *verb-legal-cases*)
       S
       (PP.case . PP)
       )
   (s1 (S -> NP VP)
       (actor  . NP)
       (action . VP.action)
       (object . VP.object)
       (vcases . VP.vcases)
       )
    (np (NP -> ?det *adj noun)
       (if det
           (quant . det.sems)
           (quant . undefined))
       (if adj (qual . *.adj.sems))
       (object . noun.sems)
       )
   (npp (NP -> NP PP)
       (fail if PP.case $<< *verb-only-cases*)
       (glitch nesting if t)      ; glitch always- mark as nested
       NP
       (PP.case . PP)
       )
    (vp (VP -> verb NP)
       (action . verb.sems)
       (object . NP)
       (numb   . verb.numb)
       (vcases . verb.cases)
       )
   (pp (PP -> prep NP)
       (case   . prep.case)
       (object . NP)
       )
   ))
```

The first sentence generates multiple parses, note the glitch in the more nested form.

```
> (parse 'S '(the small cats chase black dogs in the park))

_____________
complete-edge 0 9 s2 S (the small cats chase black dogs in the park)
nil
s2   S -> (S PP)
Syntax
(S
 (S (NP (det the) (adj small) (noun cats))
    (VP (verb chase) (NP (adj black) (noun dogs))))
    (PP (prep in) (NP (det the) (noun park))))
Semantics
(S (actor (quant . specific) (qual (size . 3/10)) (object . feline))
   (action . hunts)
   (object (quant . undefined) (qual (color . black))
           (object . canine))
 (vcases dest)
 (locn (case . locn) (object (quant . specific) (object . place))))
_____________
complete-edge 0 9 s1 S (the small cats chase black dogs in the park)
nil
     Glitches: (nesting)
s1   S -> (NP VP)
Syntax
(S (NP (det the) (adj small) (noun cats))
   (VP (verb chase)
   (NP (NP (adj black) (noun dogs))
       (PP (prep in) (NP (det the) (noun park))))))
Semantics
(S (actor (quant . specific) (qual (size . 3/10)) (object . feline))
   (action . hunts)
   (object (quant . undefined) (qual (color . black))
           (object . canine)
           (locn (case . locn)
                 (object (quant . specific)
                         (object . place))))
 (vcases dest))
t
```

The second example has only one valid parse (the other fails)...

```
> (parse 'S '(the small cats chase black dogs into the park))
_____________
complete-edge 0 9 s2 S (the small cats chase black dogs into the
                           park) nil
s2   S -> (S PP)
Syntax
(S
 (S (NP (det the) (adj small) (noun cats))
    (VP (verb chase) (NP (adj black) (noun dogs))))
 (PP (prep into) (NP (det the) (noun park))))
Semantics
(S (actor (quant . specific) (qual (size . 3/10)) (object . feline))
   (action . hunts)
   (object (quant . undefined) (qual (color . black))
           (object . canine))
 (vcases dest)
 (dest (case . dest) (object (quant . specific) (object . place))))
t
```

A4.7. Tracing Rules

Lkit offers a facility which allows users to trace the activity of their grammar rules. The main use for this is to gain information about the semantic processing performed by rules when users are debugging their grammars. The trace facility operates on two levels, one offering slightly more information than the other. In its more verbose form it provides details of the progress of a rule once it has fired. In its less verbose form it simply traces the activity of a rules semantic processing. The following details provide an overview of the tracing facilities available.

name: trace-rule
description: switch on verbose tracing for a named rule or list of rules.
example: (trace-rule 'vp)
(trace-rule '(vp np))

name: untrace-rule
description: switch off verbose tracing for a named rule or list of rules.
example: (untrace-rule 'vp)
(untrace-rule '(vp np))

name: untrace-all-rules
description: switch off verbose and semantic tracing for any rules which are currently being traced.
example: (untrace-all-rules)

name: trace-semantic
description: switch on semantic tracing for a named rule or list of rules.
example: (trace- semantic 'vp)
(trace- semantic '(vp np))

name: untrace- semantic
description: switch off semantic tracing for a named rule or list of rules.
example: (untrace- semantic 'vp)
(untrace- semantic '(vp np))

name: untrace-all- semantics
description: switch off semantic tracing for any rules which are currently being traced.
example: (untrace-all-rules)