

Crib Sheet

Version of 6/3/13

Quick tour of the directory structure.....	1
Load options, “scripts”, and “workspaces”	3
‘Switch’ settings.....	4
A Debugging session — Function abbreviations.....	4
Crib sheet for Krisp categories	9
Crib sheet for phrase structure rule forms.....	10

Quick tour of the directory structure

The only real way to gain an understanding of what Sparser can do and how to use it is to try it out and explore the code. If the Lisp and editor you are using does not support the “meta-dot” capability to bring up a function definition by typing that key combination (meta-key + period) then find another one. The code in the system was under continuous development for seven years and then intermittently for another 15. I don’t pretend to remember everything it can do (ddm).

At the very top of the distribution we have

```
sparser
  load-nlp.lisp
  Mumble
  README
  Sparser
  util
```

As described in README, loading `load-nlp.lisp` will load all of the code for Sparser and its companion linguistic realization component Mumble. The code in `util` is a generic set of useful Lisp code that is incorporated into both Sparser and Mumble. This is the only code designed to be loaded by an ASDF system. The rest is loaded by a home-brewed loading system because of the accidents of history.

Ignoring the code layout under Mumble, the layout under Sparser is this:

```
sparser
  code
    clean empty tree
    f
    s
      analyzers
      drivers
      grammar
      init
      interface
      objects
      tools
```

The directory `s` stands for “source”, `f` stands for “fasl” (the usual Lisp format for compiled files), and the `clean empty tree` is what it’s name suggests: a copy of the

full directory structure under `s` (or `f`) without any files. Under Allegro or any other Lisp that doesn't automatically store its compiled files in an invisible directory Sparser's compiler directive will arrange for them to be stored in the `f` directory tree.

The `tools` directory holds utility code, some of it specific to Sparser, some that could migrate to the general `util` directory.

The code in the `analyzers` and `drivers` directories controls what Sparser does. The broad form of an algorithm tends to be in `analyzers` and runtime control in `drivers`. In particular, the tail-recursive state machine that governs Sparser's moment to moment action is in `drivers/chart/psp/`.

The code in the `objects` directory defines all of Sparser's types, the forms for defining them, and all of the mechanics of creating, recycling, and coordinating what they can do. It's model subdirectory has the code behind what was once the "new" conceptual model representation as described in the Krisp paper.¹

With minor exceptions that will be moved, all of the grammar (and semantic model, and database facilities) is under the directory `grammar`.

```
grammar
  model
    core
    dossiers
    sl
  rules
  tests
```

The `tests` directory is a repository of miscellaneous test routines. The `citations` directory there is the closest we presently have to a regression testing facility.

The directories inside the `rules` directory hold all of the syntactic grammar and the other grammar-relative rules such as for traversing paired punctuation, e.g. "`<...>`" (`traversal`) or introducing segment boundaries (`brackets`). The most active parts are `words`, which has directories that provide minimal definitions for close to all of the English function words; `tree-families`, which holds the stock of exploded TAG tree families that are used to schematically define the semantic grammars; and `syntax`. The `words` directory also had the code for doing word frequency tabulation for documents (`frequency.lisp`), and within its `one-offs` subdirectory there are files derived from NYU's Comlex, and others lifted from the ERG grammar.

The `model` directory of `grammar` (as in "conceptual model") divides into treatments for particular sublanguages (`sl`), and for phenomena that are common to most if not all sublanguages (`core`). The stock of instances for both of these are in specific files in the `dossiers` directory.

The `init` directory is for initialization. Any substantial extension to Sparser starts here and will involve changes to several files.

```
init
  everything.lisp
  images
  Lisp
  scripts
```

¹ Need a uniform reference location for it.

```
versions
  v4.0
    config
      grammars
      image.lisp
      launch.lisp
      load.lisp
    loaders
      salutation.lisp
      updating.lisp
    workspace
workspaces
```

Sparser was originally a commercial system. Different customers and licensing arrangements were coordinated by maintaining different “versions” – each with its own configuration of grammars, file loaders, and so on. That is less important now, but provides an option if we were to do some major factoring of Sparser development.

Load options, “scripts”, and “workspaces”

The file `init/everything.lisp` controls the entire process of loading Sparser’s code into a Lisp. It establishes values for the defaults and accommodates to different file system conventions.² It loads the core and the grammar, and does all the final initializations.

When you read through `~/sparser/load-nlp.lisp`, you will see that you can designate a script file that can override these defaults for a particular project. There are examples in `init/scripts/`. These can ‘preset’ parameters, load additional grammar models and establish switch settings. They always end with their own call to `load everything.lisp`.

There are also scripts that are intended to be run after Sparser is loaded, particularly `compile-everything.lisp`. If you are running in a Lisp like Franz’s Allegro that does not automatically compile forms as they are loaded, you need to follow the instructions in that file to use Sparser’s loader to walk all the files and compile them. If you don’t, the tail-recursive controller at the heart of Sparser will blow out the stack on any text longer than about 40 words.

The files in `init/workspaces/` are places for putting what you’re currently working on. They reflect projects from various periods and tend to contain examples, function names to use with meta-dot to maintain some context, sets of special switch settings, and such. They are loaded after all of the rest of Sparser has been loaded. Anything from that should not be executed needs to be commented out. Most recently, we (ddm, sf) were working from the Grok workspace, which is not gradually being emptied out. New newest workspace is Strider.

² The default assumption is that the files are in some sort of unix-style file system, and it is tuned to OS X. It has also been loaded under Windows, and under the original logical pathname style of the Apple Macintosh prior to OS X.

‘Switch’ settings

Sparser’s operation is highly parameterized. What it does depends on the values of a large set of global symbols and programmable functions, i.e. functions where we swap the definition on the fly or as part of the load-time initialization. Sensible combinations of values for these parameters are bundled into functions. All of these functions are in the file `drivers/ints/switchs2.lisp`, which is well worth reading to get a sense of what alternatives are possible.

<code>switch-settings</code>	function
Displays the current values of the more important switches. Convenient for meta-dot access to the file that defines all the settings	
<code>top-edges-setting/ddm</code>	switch-setting function
This is the default-setting. It is designed for doing full analyses with completely defined semantic grammars and a few heuristics.	
<code>word-frequency-setting</code>	switch-setting function
Removes any analysis operations and just passes each terminal to the function <code>record-word-frequency</code> . See <code>grammar/rules/words/frequency.lisp</code> for the different ways word-frequency counting can be tuned.	
<code>tuned-grok</code>	switch-setting function
A variation on the original setting for the Grok project that works better and reflects what does and doesn’t work in that project presently.	

A Debugging session — Function abbreviations

Most of the time what you are doing is debugging some extension to the grammar. This comes down to running short text strings and looking at the results. What rules applied, what words are unknown. To cut down on the amount of typing required we have short form functions for common operations. The full set is defined in `/init/versions/v4/workspace/abbreviations.lisp`.

<code>d (object)</code>	function
Calls the built-in function <code>describe</code> on the object and returns the object, making it easy to reference it in your next call using the built-in variable <code>*</code> .	
<code>pp (string)</code>	function
Short for <code>analyze-text-from-string</code> .	
<code>tts (&optional(stream *standard-output*))</code>	function
Runs over the entire chart printing the labels of the treetops	
<code>p (string)</code>	function

Calls `pp` then calls `tts`. This is the most convenient way to do small to medium sized tests.

Here is an example from when I was evaluating the grammar and model for adjectives or adverbs that we could conceptualize as ‘approximators’.

```
sparser> (p " it is almost Wednesday ")
^Ait is almost Wednesday^B

                                source-start
e0      pronoun                 1 "it" 2
e1 e2      "is" :: is, be
e5      time                    3 "almost wednesday" 5
                                end-of-source

:done-printing
```

The numbers on the left (e0...e5) are the edges in the chart. The next column is the label on the edge. The column on the right is the words that are spanned by the edge (Note that chart positions are between the words; see §3.1.) For edges over words, this third column is their labels, where for e2 over “is” it rewrites both as itself and as the category ‘be’. The leading control-A and trailing control-B characters are automatically inserted terminators.

Edge 5 has the label I want, but I wasn’t sure that it was being derived via a general rule from the approximator code (`/grammar/model/core/adjuncts/approx/`) or by a special rule from the time grammar (a bad idea because there are so many other words like “almost” that pattern the same way). First I looked at the edge.

`ie (number-of-edge)` function

Short for ‘inspect edge’. It gets the edge by calling `edge#` and then calls `d` on it

```
sparser> (ie 5)
#<edge5 3 time 5> is a structure of type edge. It has these slots:
category      #<ref-category time>
form          #<ref-category np>
referent      #<psi relative-time 58>
starts-at     #<edges starting at 3>
ends-at       #<edges ending at 5>
rule          #<PSR406 time -> modifier weekday>
left-daughter #<edge3 3 modifier 4>
right-daughter #<edge4 4 weekday 5>
used-in       nil
position-in-resource-array 5
constituents  nil
spanned-words nil
#<edge5 3 time 5>
```

This chart edge is a Lisp ‘struct’. This shows the values of its fields. Its semantic label is `time`; its syntactic form label is `np`; and its referent is an instance of the category `relative-time`, which is what makes me suspicious that it’s a special rule in the time grammar. To confirm this I look at the rule.

`ir (number-of-rule)` function

Looks up the rule using `psr#`, calls `d` on it.

```
sparser> (ir 406)
#<PSR406 time -> modifier weekday> is a structure of type cfr. It
has these slots:
symbol      rule::PSR406
```

```

category      #<ref-category time>
rhs           (#<ref-category modifier> #<ref-category weekday>)
completion    nil
form          #<ref-category np>
relation      nil
referent
  (:instantiate-individual-with-binding #<ref-category relative-time>
   (#<variable relativizer> . left-referent)
   (#<variable reference-time> . right-referent))
schema        #<schr np -> modifier np-head >
plist
  (:relation :definite-modifier :grammar-module
   #<grammar-module *time*> :file-location
   "/Users/ddm/Sparser/Sparser/code/s/init/../../../../code/s/grammar/model/c
ore/time/relative-moments.lisp")
  #<PSR406 time -> modifier weekday>

```

This indicates that this is an ordinary binary context-free phrase structure rule, as opposed to the form rule that I was hoping for, or a context-sensitive rule or the result of running something more exotic. The most useful thing about this description is often in the `plist` field (for ‘property list’), that indicates what file was being loaded when this rule was defined—a file in the time grammar.

Looking at that file reveals a very nice conceptualization (concept + realization) for circa 1994. But in 2011 we have grander notions. Here is the definition.

```

(define-category relative-time
  :instantiates time
  :specializes time
  :binds ((relativizer (:or relative-time-adverb
                           approximator sequencer))
          (reference-time (:or time time-unit month weekday)))
  :index (:sequential-keys reference-time relativizer)
  :realization (:tree-family modifier-creates-definite-individual
                  :mapping ((np . time)
                           (modifier . (approximator
                                         sequencer
                                         modifier))
                           (np-head . (time
                                         time-unit
                                         month
                                         weekday ))
                           (result-type . relative-time)
                           (individuator . relativizer)
                           (base-category . reference-time))))

```

For the full description of what’s going on here see §TBD, but in short, the category is a simple frame that defines two variables named `relativizer` and `reference-time` and specifies what types (categories) their values are restricted to. That list of alternative categories matches up with the categories in the realization mapping. It links into the taxonomy (`:specializes`) as a subcategory of `time`. Instances of this class are stored in the discourse history (`:instantiates`) also as `time`.

Loading that category definition created a lot of rules.

```
ic (name-of-category) function
```

Looks up the category using `referential-category-named` and describes it.

```

sparser> (ic 'relative-time)
#2=#<ref-category relative-time> is a structure of type

```

```

referential-category. It has these slots:
plist
  (:grammar-module #<grammar-module *time*> :file-location
   "/Users/ddm/Sparser/Sparser/code/s/init/../../../../code/s/grammar/model/core/time/relative-moments.lisp")
  symbol          category::relative-time
  rule-set        nil
  slots
    (#3=#<variable relativizer> #4=#<variable reference-time>)
  binds          nil
  realization
    (:schema
     (:no-head-word #<etf modifier-creates-definite-individual>
      ((np . #1=#<ref-category time>)
       (modifier #<ref-category approximator> #<ref-category sequencer>
        #<ref-category modifier>)
       (np-head #1# #<ref-category time-unit> #<ref-category month>
        #<ref-category weekday>)
       (result-type . #2#) (individuator . #3#) (base-category . #4#))
      nil)
     :rules
     (#<PSR422 weekday -> hyphen weekday>
      #<PSR421 weekday -> hyphen month>
      #<PSR420 weekday -> hyphen time-unit>
      #<PSR419 weekday -> hyphen time>
      #<PSR418 month -> hyphen weekday>
      #<PSR417 month -> hyphen month>
      #<PSR416 month -> hyphen time-unit>
      #<PSR415 month -> hyphen time>
      #<PSR414 time-unit -> hyphen weekday>
      #<PSR413 time-unit -> hyphen month>
      #<PSR412 time-unit -> hyphen time-unit>
      #<PSR411 time-unit -> hyphen time>
      #<PSR410 time -> hyphen weekday>
      #<PSR409 time -> hyphen month>
      #<PSR408 time -> hyphen time-unit>
      #<PSR407 time -> hyphen time>
      #<PSR406 time -> modifier weekday>
      #<PSR405 time -> modifier month>
      #<PSR404 time -> modifier time-unit>
      #<PSR403 time -> modifier time>
      #<PSR402 time -> sequencer weekday>
      #<PSR401 time -> sequencer month>
      #<PSR400 time -> sequencer time-unit>
      #<PSR399 time -> sequencer time>
      #<PSR398 time -> approximator weekday>
      #<PSR397 time -> approximator month>
      #<PSR396 time -> approximator time-unit>
      #<PSR395 time -> approximator time>))
  lattice-position #<top-lp-of relative-time 149>
  operations      #<operations for relative-time>
  mix-ins         nil
  instances       nil
  rnodes         nil
  #<ref-category relative-time>

```

The definition from 1994 is probably good for ‘sequencers’ like *next* or *after*. But the notion of a relative time doesn’t really apply to a phrase like *almost Wednesday*, which means something like “it’s close to Wednesday”.

The correct fix here is to remove approximators from the value-restrictions and mapping of the category definition, but if you want to experiment right away, or aren't sure, then you can surgically delete the rule/s that you don't want.

`delete/cfr# (number-of-rule)` function

Looks up the rule (of any sort) that has that number and removes it from the rule catalog. Returns the (now ineffective) rule in case you want to reinstate it.

```
sparser> (delete/cfr# 406)
#<PSR406 time -> modifier weekday>

sparser> (p "it is almost Wednesday")
it is almost Wednesday

                                source-start
e0      pronoun                 1 "it" 2
e1 e2    "is" :: is, be
e3      modifier                 3 "almost" 4
e4      weekday                  4 "wednesday" 5
                                end-of-source

:done-printing
```

Other useful short functions.

`ip (number-of-position)` function

Runs describe (d) on the position with that number.

```
sparser> (ip 4)
#<position4 4 "wednesday"> is a structure of type position. It has
these slots:
array-index      4
character-index   14
display-char-index nil
token-index      4
ends-here        #<edges ending at 4>
starts-here      #<edges starting at 4>
terminal         #<word "wednesday">
preceding-whitespace #<word one-space>
capitalization    :initial-letter-capitalized
assessed?        :word-completed
#<position4 4 "wednesday">
```

The position object records the word that follows it (p4 lies between *almost* and *Wednesday* in this example), the index in the character source at which the word starts, and the whitespace (if any) that separated the words on each side of the position. It records any special facts about the word (e.g. that is capitalized). The `starts-here` and `ends-here` fields provide the machinery for anchoring edges to positions.

`iw (string-for-word)` function

Describes the fields of the word

```
sparser> (iw "almost")
#1=#<word "almost"> is a structure of type word. It has these slots:
plist
(:grammar-module #<grammar-module *approximators*> :file-location
"/Users/ddm/Sparger/Sparger/code/s/init/../../../../code/s/grammar/model/d
ossiers/approximations.lisp")
symbol      word::almost
```



```

rule-set          #<rule-set for #1#>
pname            "almost"
morphology        nil
capitalization    :lower-case
capitalization-variants nil
#<word "almost">

```

Crib sheet for Krisp categories

Like most everything else, you define a category by writing an expression in a text editor in a file that is loaded as part of Sparser. Here is a category definition from the ‘amounts’ module of Sparser’s core grammar.

```

(define-category amount
  :specializes nil
  :instantiates self
  :binds ((measurement . measurement)
          (stuff)
          ;; adjuncts that can be expected
          (alternative-amount)
          (time-period))
  :index (:temporary :sequential-keys stuff measurement))

```

The name of the macro that creates the category object (on the fly at the point where the expression is read during the load) is **define-category**. The symbol following it is the name of the category: `amount`. The ‘nil’ after the keyword **:specializes** indicates that it is a toplevel category (i.e. there is nothing above it in the inheritance hierarchy). The **:instantiates** keyword says what semantic category should be used as the label on any grammar rules based on this definition (though see `:rule-label` below) and how it should be treated in the discourse history.

The core of a category definition is its **:binds** statement. This statement creates a set of local variables. (You can think of them as slots in a frame or members in a class; the actual reference point is the Lambda Calculus.) Here we are defining four variables, including the constraint that the ‘measurement’ variable can only be bound to objects of type ‘measurement’ (e.g. the number “50”). The first two variables are used to individuate different instances of the category, as indicated by the **:index** statement. As noted in the comment the other two are frequent adjuncts to amount phrases.

If you want to define a category that is a subclass from a semantic perspective but patterns like some other class, then you add the parameter **:rule-label**, which as the name suggests will over-rule the default treatment that uses the name of the category as the label on the rule(s). For example, if you have this definition,

```

(define-category boy
  :instantiates person
  :specializes person
  :realization (:common-noun "boy"))

```

then the name of the category and this simple **:realization** statement lead to these rules

```

#<PSR476 boy -> "boy">
#<PSR477 boy -> "boys">

```

However, if you use **:rule-label** then the rules change. (N.b. you can freely redefine a category at any time by changing its definition and re-evaluating. The identity of the category remains the same (is the same `eq` object), but the contents and in particular the generated rules can change. This version of the category definition changes the label on the rules (and deletes the old rules if this is a redefinition).

```
(define-category boy
  :instantiates person
  :specializes person
  :rule-label person
  :realization (:common-noun "boy"))

#<PSR2218 person -> "boy">
#<PSR2219 person -> "boys">
```

This next example shows how to pair a category definition with a set of automatically created semantic grammar rules for phrases based on it. Using the **:realization** keyword, we associate a particulate syntactic schema with a mapping that determines how it should be instantiated. The terms on the left of the mapping expression are from the schema, those on the right are either references to variables in the category (unit and quantity), the keyword **:self** to indicate the category, or other categories in the grammar (e.g. ‘unit-of-measure’).

```
(define-category measurement ;; "10 yards"
  :specializes nil
  :instantiates self
  :binds ((units . unit-of-measure)
          (quantity :or quantity number))
  :realization (:tree-family quantity+kind
    :mapping ((quantity . quantity)
              (base . units)
              (np . :self)
              (np-head . unit-of-measure)
              (modifier . (quantity number))
              (result-type . :self))))
```

Crib sheet for phrase structure rule forms

Absolute minimum

```
(def-cfr based-at ("based" "at"))
```

Daughter referent — transparently incorporating prepositions

```
(def-cfr in-date ("in" date)
  :form pp
  :referent (:daughter right-edge))
```

Binding a variable of an already instantiated individual (Chomsky adjunction)

```
(def-cfr date (number date)
  :form np)
```

```

:referent (:head right-edge
            :bind (day . left-edge)))

```

Instantiating a category given its head word

```

(def-cfr earliest-arrival-date ("EAD")
  :referent (:instantiate-individual earliest-arrival-date))

```

Instantiating a category and binding one of its variables

```

(def-cfr quantity-of (number "x")
  :form determiner
  :referent (:instantiate-individual requested-resource
            :with (quantity left-edge)))

```

Context Sensitive Rules

```

(def-csr kind location
  :left-context based-at
  :form np
  :referent (:function recast-kind-as-a-location right-edge))

```

Form Rules

```

(def-form-rule (quantity-of n-bar)
  :form np
  :referent (:function recast-as-resource right-edge))

```

Also takes a `:new-category` keyword with the name of the category.

“Exporting”

```

(set-generic-treetop-action category::earliest-arrival-date
  'export-bindings/recursively)

```