Short-cuts, events, and individuals

Sparser is a rule-driven, chart-based, bottom-up, model-driven natural language parsing and analysis system with a substantial build-in grammar. An analysis is driven either by FSAs or phrase-structure rewrite rules. FSAs are ad-hoc and build out of raw code for very specialized situations; the only FSAs one is every likely to deal with (see the results of) are used for number expressions and proper names. There is rarely a need to write another FSA.

Raw phrase structure rules

Phrase structure rewrite rules will do most everything you will ever need to accomplish. Virtually all rules are binary: two adjacent constituents (righthand-side) are composed to form a new constituent that spans them both (lefthand-side). Following the conventions for chart-based parser, constituents in Sparser are referred to as edges. Every edge as three (primary) components: its semantic category, its syntactic category, and its referent. Semantic categories are arbitrary label from a domain-specific semantic vocabulary that describes the text it covers in terms of what sort of thing it is: a company, a date, a TPFID. Syntactic, or 'form' categories are taken from the standard set of labels: NP, VP, proper-noun.[1] The referent of an edge is the data structure used to represent its meaning, what it is. Sparser has a native representation language known as Krisp that is basically a frame system that has been tuned for a close fit to how information is distributed in a natural language.

Rules (and the code for FSAs) are defined by written expressions stored in files and interpreted as Sparser is launched. You can write rules one at a time. Here is an example.

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

This Lisp 'special form' is literally a macro. When it is evaluated it creates the data structures for the rule and knits it into the parsing machinery for immediate effect. The macro's names stands for "define context free rule." The lefthand side (lhs) is the label (semantic category) 'in-date', the righthand side (rhs) is the ordered pair of terms consisting of the word spelled "in" followed by the label 'date'. The :form keyword argument indicates that an edge formed from the application of this rule is syntactically a prepositional phrase. The expression after the :referent keyword indicates that the referent for this new edge should be the same as the referent of its right daughter (the date); in this instance we are throwing away the preposition.

[1] The complete set of form categories defined for Sparser are given on file …/grammar/rules/syntax/categories.lisp

Model-driven rules

While one can write all the rules by hand, the whole point of Sparser as a model-driven system is the intimate link between the rules that are driving the parser (the semantic grammar) and the stuff that makes up the semantic model that the 'consumer' of the texts' analyses is using. By using Sparser's native representational language, we can tie rules to the corresponding elements in the model in a natural way. Here, for example, is one of the model definitions from the Who's News domain. It is responsible for the analysis of texts like "*George L. Ball resigned yesterday as chairman and chief executive officer of Prudential-Bache Securities Inc*."

This form is also a macro. It defines a 'referential-category' with two slots (:binds) that fits into the taxonomy as a subcategory of leave-position but using the category job-event as its label on edges. The :index information is there to ensure that two instances of the same person retiring, for instance, are mapped to a single object representing that event.

```
(define-category  retire-from-position
  :instantiates  job-event
  :specializes  leave-position
  :binds ((person . person)
          (position . position))
  :index (:special-case :find find/je
                        :index index/je
                        :reclaim reclaim/je )
  :realization
     (:tree-family intransitive
      :mapping ((s . job-event)
                (vg . :self)
                (np/subject . person)
                (agent . person))
      :main-verb "retire"
      :additional-rules
         ((:pp-adjunct (s (s as-title)
                          :head left-referent
                          :binds (position  right-referent))))))
```

The realization of this category is based on a predefined, parameterized 'family' of rules, and the mapping and main-verb information provide the values for the parameters. Here is the definition.

```
(define-exploded-tree-family  intransitive
  :description "A verb that requires only a subject to have a
complete sentence. It may be followed by prepositional phrases
that carry crucial information, but that information may also
just be conveyed by context."
  :binding-parameters ( agent )
  :labels ( s vp np/subject )
  :cases
     ((:subject (s  (np/subject vp)
                 :head right-edge
                 :binds (agent left-edge)))))
```

When the mapping is applied (part of the work of executing the definition of retire-from-position), we will get a total of six semantic grammar rules. Four for the different

conventional morphological forms for the main-verb "retire" that each rewrite it using the label 'job-event', and one rule that if written by hand would look like this. We get a second binary rule because of the 'additional-rules' specified with the realization.

```
(def-cfr job-event (person job-event)
     :form clause
     :referent (:head right-edge
                :binds (agent left-edge)))
```

This combination of defining a category (class) for the reasoner along with description of its English realization as a set of specialized rules is the standard way of writing grammars for Sparser. You figure out the concepts first, and assuming they are close to the 'surface' forms of the language you spell out the realization at the same time.[2]

Short-cuts, events, and individuals

All this takes a long time however, if only because there is so much to type. So at the cost of the specificity that a semantic grammar normally has, I've introduced the first few of what will eventually be a large set of abbreviated forms that get all this down to a single terms plus the target word. These forms in effect do the typing for us, at the cost of simplifying the mapping to just two categories: individual and event.

Nearly all of the vocabulary for the checkpoint operations demo is done this way. The words that are defined are general and should eventually be moved to a better location within Sparser's file system, but for the moment this makes for a simpler work flow. So for example, if you type

```
(np-head "trunk")
```

That will invoke this function.

```
(defun np-head (string-for-noun) ;; "trunk", "car", ...
  (unless (stringp string-for-noun)
    (error "Argument must be a string providing the base noun"))
  (let* ((name (intern (string-upcase string-for-noun)
                       (find-package :sparser)))
     (form
     `(define-category ,name
        :instantiates :self
        :specializes individual
        :realization
          (:tree-family NP-common-noun
           :mapping ((np . individual)
                     (np-head . :self))
           :common-noun ,string-for-noun))))
    (eval form)))
```

Which does the typing for us within that back-quoted sexp. It creates a new concept with the same name as the word, and it links it to a tree family that already knows about

[2] If the concepts don't directly map to head words like verbs and a set of arguments, then the clean way to recover them from a parse is to invert the planning options of a language generation system. Otherwise the burden goes to the reasoner because Sparser's grammar is strictly oriented to surface forms.

combinations with "*the*" and other determiners. (Though a spot-check just now shows some lacunas that should be remedied.)

I think of a form like np-head as a 'short-cut'. It is right at the surface level and it is lacking in the sources of precision that semantic grammars can provide, but the trade-off of speed for accuracy is a worthy experiment. A short-cut is based on a syntactic paradigm: what syntactic class is the word(s) and what is the canonical syntactic construction that is supports (which is a long-winded way of saying what arguments does it take.

There are presently short cuts for ignorable-np-modifier, intransitive verbs (so), transitive verbs (svo), and intransitive verbs that take prepositions (sv-prep). There are several other obvious candidates, but that already goes a fair way.