7. Hooks for User Code

Not all of the things that a user may wish to do in the course of an analysis can be provided by the customary facilities of a parsing system. One designer's needs may be different from another's, and no system architect can anticipate all the options that could be desired, nor should they try to if the core system is to be efficient.

Sparser's accommodation to this problem is to provide a set of "hooks"—standard moments during its core algorithms when user-written code is looked for and executed. In this section we describe two hooks: one when an edge or a word is completed, and a second triggered by designated word or edge labels during word-level actions.

Formally a hook is a Sparser-internal function. The placement of the calls to the function dictate when the hook will be applied. Every hook takes a specified set of arguments, one of which will act as a key by which to lookup a user function. The user-defined function, if one has been defined for this key, is then called with the same arguments as passed to the hook function. The user function may be required to return a specific class of value as a means of communicating with Sparser's algorithms and updating its state information.

The operations at a hook are managed by a set of routines. These routines allow users to associate functions that they have written with particular labels. The functions are then executed (`funcall`'d) whenever an object with one of those labels is passed to the hook. The functions must take a set of arguments that are specific to the hook involved, and often must return a specific kind of value.

Hooks operate on the symbol that names a function, not on the function's lambda body at the moment is was associated with its key. Consequently you can change the effect of a hook for given label by changing the definition of the function; there is no need to redefine the hook–label association. Associations can also be deleted.

Obviously, there is no constraint on what the code does once one of your functions is called at a hook. There is usually enough information about the data structures passed to the hook or accessible from it to perform any imaginable calculation on or traversal of the chart in its state at that moment, to introduce new edges, or to update your own data structures.

The only substantive constraint is implicit in the requirements on the value the user's code must return. If the wrong kind of argument is returned, or if its value is inappropriate for what actually happened, then there may be a Lisp error or there may be irregularities in the rest of the analysis, with portions of the input skipped or processed redundantly.

The natural way to integrate the result of operations at a hook is to create one or more edges over the span of text the user's code has analyzed. These edges will then be incorporated into Sparser's edge-based processing just like any other edge.

7.1 The “Completion” hook

When the pattern of adjacent edges on the righthand side of some rule is found and a new edge is formed, we say, following Earley, that the rule has been “completed”. Extending this notion somewhat, we can also speak of a word being “completed” at the moment when it is scanned.

There is a hook for user code at these points. For edges it is placed just after all of the fields of a newly created edge have been populated and before its label is checked for the possibility of instantiating further edges (§6.5). For words it is placed just after the check for polywords and user-defined analysis routines and before any edges over the word are introduced (see §5.5). The occurrence of a polyword pre-empts the completion hook for all of the words within the polyword. If the completion is to be applied to any of the words covered by the routine’s analysis, a user-defined routine must call the hook explicitly (see below). The hook is always applied to edges.

At the completion hook, you may place code that will operate over instances of an object corresponding to a particular label (a word or edge object). This code will also have access to the position at which the object occurred: for an edge this is encoded in the object; for a word it is supplied explicitly by the position objects before and after the word that are passed to the code as arguments.

The intended purpose of the completion hook is to make it possible to incrementally tally what words and edges have occurred and to do any user-specific operations that should occur to every edge (or word) as an individual: maintaining a discourse history, displaying the parse, matching paired punctuation, etc.

For example, if one was developing a system to categorize the topic of texts based on the occurrences of specific words, one could link designated words to routines that would keep a running tally of how many instances of a word of a given category had occurred: The word “parsing” might be linked to a function that incremented a counter for the topic ‘computational linguistics’; if the word occurred four times in the text under analysis then that function would be called four times.

Several logically distinct actions can all occur at completion by associating each action with a different ‘tag’. Some, all, or none of the different action-types may be defined for a particular label. This is (almost) computationally equivalent to having just one large function subsume all of the different actions one might wish to occur, but is easier to manage and encourages smaller units of more structured code. The only significant difference from lumping several logically distinct actions into a single function is that in the present version of Sparser, the order in which different action-types are performed at completion is not defined, whereas in a all-in-one routine the user can enforce a given ordering.

Actions at the completion hook are individuated by the combination of a label and an arbitrary tag (a symbol). One can define any number of actions for a given label if they all have different (non-EQ) tags. A second definition for a given label–tag pair will supersede (replace) the earlier definition.

```
define-completion-action (label, tag, function)              function
```

The ‘label’ must be a category, a word, or a polyword—the actual object, not a symbol or string naming it. The ‘tag’ must be a symbol. The ‘function’ must be a symbol that has been defined as a function.

This function either takes one argument—an edge—if the label is either a category or a polyword; or else take three arguments—a word, the position in front of it, and the position following it—if the label is a word. No check for the correct argument pattern is made.

The function will be funcall’ed every time an object with the indicated label is seen at the designated positions of the completion hook within the analysis algorithms (see discussion above).

Any number of tag–function pairs may be defined for a given label. They are stored in the ‘completion-actions’ field of the label’s rule-set; see §4.6.

`delete-completion-action (label, tag)` — `function`

The function associated with the label – tag pair is removed from the data-structure that defines completion actions, and will no longer be called when objects with that label are seen at the completion hook.

The completion hook itself is manifest in the Sparser function “complete”. This function is called automatically at specific points in Sparser’s analysis algorithms as described earlier. The description below is provided so that user’s have the option of calling it explicitly in their own word-level analysis code; as it will not be otherwise called on the words the code may scan if the code signals a successful analysis. (Note that it is incumbent on the user to carefully coordinate her usage of complete. It should not be applied unless the analysis succeeds, since if the code signals a failure (see §7.2.2), Sparser will do its own application. If the user’s code did completions on a set of words that it scanned even though it later signaled a failure, Complete would then be redundantly applied as second by Sparser.)

`complete` — `Sparser-internal function`

(< word or edge > &optional position position)

Called by Sparser at specific points in its algorithms to supply the completion hook. Its return value is ignored. When the first argument is an edge there will be no other arguments. When it is a word the two optional positions will be supplied: The first position is the one logically before the word—the one that has the word in its ‘terminal’ field. The second position is the one logically just after the word—it is the value returned by chart-position-after when applied to the first position.

7.2 Special-purpose, word-level parsing

There are some kinds of constituents that have an internal organization that is not well described by phrase structure rules. A constituent formed by phrase structure rules will have a branching structure internally, reflecting a cascade of successive rule applications in a multi-layered tree structure. Branching structure does not make sense for all kinds of phrases however, and to handle such phrases in a natural way Sparser provides a hook for special-purpose, word-level analysis routines that you can write.

Forming numbers from sequences of digits is a clear example. The number “42,735” has three literal constituents, and we work out its meaning by appreciating that the comma is acting as a separator and indicating, e.g., that the digit sequence “42” is to be interpreted as a number of thousands. However if we make the number longer, say “42,735,000”, that same sequence “42” now indicates a number of millions.

Position-specific interpretations are very difficult to formulate using phrase structure rules, as are so-called “Kleene star” constructions in general, since their definition places no limit on the number of constituents a phrase of that type can have, yet each constituent lies at essentially the same level when the meaning of the phrase is considered, and its interpretation is dependent on its position relative to an indefinite number of neighbors.

Digit-based numbers are the most obvious case of this phenomena, since there is no limit to the number of digit-triples there can be. Proper names are an other case, with the same properties of unlimited length and position-specific interpretation. Generally speaking, phrase structure rules are an inappropriate basis for analysis whenever the span of a phrase cannot be determined without appreciating what its constituents mean and this meaning is dependent on the analysis of their neighbors. Sequences of number words that reflect several different types of object, as can arise in the air-traffic control domain for altitude and speed, are an example of this.[1]

At present, Sparser provides no general mechanism for writing word-level, type-specific parsing routines such as these kinds of constructions call for. They must be written directly in Lisp, and must make their own calls to scan words, introduce edges, etc. as described below.

7.2.1 Initiating a word-level routine

Word-level routines are initiated by a hook that is one of the word level actions as described in §6.5. The hook takes as its key either a word or a category that labels a pre-terminal edge. The hook makes its check on a word just after the word has been checked for the initiation of any polywords (assuming no such polyword succeeded); and on the categories of edges just after all of the word’s pre-terminal edges have been introduced.

The check itself consists of examining the `fsa` field of the word or category’s `rule-set` field (see §4.6). The fsa field will hold either a list of one or more symbols naming user functions that the hook is to execute, or else it will be `nil` if nothing is to be triggered. This field has to be explicitly set by the user. The present version of the system has no routine for doing this automatically. (Similarly, there is no delete routine for these user-defined actions: one has to remove the no longer desired function from the field explicitly.)

[1] Historically, other potential candidates for this kind of treatment have included dollar amounts, dates, bibliographic citations, SGML tags, etc; anything where the organization of the phrase is largely dependent on the identity of the words in it rather than their parts of speech, and the meaning of the whole phrase is an idiosyncratic function of the organization rather than simply compositional. In these particular cases, however, one can can write quite reasonable phrase structure accounts using Sparser because it allows the grammarian to use a semantic grammar rather than just a syntactic one, e.g. `date -> month number, sgml-tag -> “<” “text” “>”`.

For example, if you want to set up a routine for parsing sequences of digits as numbers, you would write something like the following, where the function that we want to call to do the parsing is named “digit-fsa”. Note that Sparser’s tokenizer and word-level are responsible for appreciating arbitrary sequences of digits as such and introducing pre-terminal edges over them (or not—this action is controlled by the setting of various flags, see §6.6.4.). If digit sequences are being handled, they will introduce edges with the label “digit-sequence”, as assumed in this example.

```
(let ((rule-set (cat-rule-set (category-named 'digit-sequence))))
  (setf (rs-fsa rule-set)
  (push 'digit-fsa rule-set)))
```

The function you use must take specific arguments. If the routine is being initiated by a word, then the function must take two arguments: the word that triggered it and the position at which the word occurred (the position logically just in front of the word). If the routine is initiated by an edge with a particular category, then the function must again take two arguments: the edge (which will necessarily span only one word), and the position at which that edge starts. Our example, digit-fsa, would thus start off like this:

```
(defun digit-fsa (edge starting-position)
   … )
```

When a label has been assigned more than one word-level analysis routine, the protocol is different depending on whether it is a word or an edge that does the triggering. In the case of words, once one of the routines has succeeded the hook returns and none of the others are tried. In the case of edges, all of the routines are executed whether they succeed or not, and the value returned by the hook is the furthest position returned by any of them.

7.2.2 Return values

Since the purpose of routines at this level is to make an analysis of a section of the text—an analysis which is presumably intended to pre-empt any other analyses, certainly any other analyses at the word-level—each word-level routine must communicate back to the hook whether it has ‘succeeded’, i.e. has found an instance of the kind of phrase or word pattern that it was designed to look for; or whether it has ‘failed’, no such phrase existed at the point where it was called and some alternative analysis should be tried.

Following the standard Lisp convention of overloading predicates so that they can pass back useful information when they succeed, Sparser uses the following convention: When a word-level routine fails, it is to return the value `nil`. When one succeeds, it is to return the position object logically just after the farthest word that it has incorporated into its analysis.

After a routine has succeeded, Sparser will continue with its standard word-level actions at the position the routine returned, i.e. with the word that immediately follows that position.

7.2.3 Coordinating the scan

Typically, a word-level analysis routine is triggered by a word or category of words that might be the first word of some phrase-type of interest. (A category of words is indicated by having all such words rewritten as single-term edges with the same category, e.g. “digit-sequence”.) For example the word “$” is an appropriate trigger when one is looking for money phrases.

Invariably the purpose of a word-level analysis routine is to scan additional words (and possibly to look backwards to already analyzed portions of the text) and to examine them or the edges that they introduce to determine whether one of the looked-for patterns is present. Carrying out this additional scanning is the responsibility of the analysis routine, and it must be done in such a way that it will be correctly coordinated with the standard scanning process. (Note that the recording of any intermediate results is also the responsibility of routine, though a convenient way to do this is to introduce intermediate edges.)

The potential problems in coordination concern words that are scanned by the analysis routine but not incorporated into the phrase (edge/s) that it forms when it succeeds. There will almost always be at least one such word, since the routine will typically have to look at at least one word beyond the end of the phrase it accepts in order to determine that the phrase has ended.

A number of operations would normally be applied to each word as it is scanned in the normal course of events, some of which may be unnecessary or even non-sensical when the scan is made by a word-level analysis routine (for instance triggering word-level analysis routines). The point of coordination is to ensure for the scanned but unincorporated words that the operations not taken by the analysis routine are done by the normal procedure and that those that the routine did are not inadvertently repeated. (Note that if the sought-for pattern is not present then every word that the routine scans will be unincorporated, and the normal procedure will have to resume practically where it left off.)

Because of the possibility of departing from the normal default word-level procedures through an analysis routine, the steps in that normal procedure takes are flagged by a set of “status” keywords. Coordination is a matter of ensuring that the status of the scanned positions has the appropriate values, and thereby ensures that all and only the correct remaining operations are taken when the normal procedure is resumed. This will happen (almost) automatically if the standard subroutines are used for scanning the chart, introducing the edges, etc., since the subroutines set the flags as part of their operation.

The function below is provided as a model for coordinating the inner scan loop of a word-level analysis routine.

```
(defun scan-and-add-edges ()
  (let* ((position-before (scan-next-position))
         (word (pos-terminal position-before))
         (position-after (chart-position-after position-before)))
    (let ((edges (install-terminal-edges word
                                         position-before
                                         position-after)))
     (setf (pos-assessed? position-before)   ;; status flag added explicitly
           :pre-terminals-done-from-fsa)
     (values position-before word edges))))
```

Note that when the analysis routine is called, the scanning process will have reached only as far as the word that has triggered the routine—the position following that word will be empty; a call to scan-next-position will be needed to populate it with the next word from the text, and that next word might very well be the word that indicates that the text is finished (§3.1.5.3). Recall also that the chart cannot be scanned beyond the last word. Consequently one must be careful to not make superfluous scans and should check for the end or source explicitly.

7.3 Tree-top generic action hook

An edge that is not dominated by any other edge can be called at "treetop"—the top edge in the tree of daughter edges that it was formed from. Similarly, a chart can be viewed as a sequence, or "forest", of such treetop edges. There are two hooks for 'per-treetop' actions. One for conceptual analysis (§6.?????), and one for any arbitrary 'generic' actions, which is described here.

The treetop generic action hook is like the completion hook (§7.1) in that you associate a function with a label, and the function is called whenever a treetop with that label is reached during the walk over the treetops (§6.8.2). Whether these actions are ever taken is determined by the **`do-general-actions-on-treetops`** switch, and on whether the forest-level is active.

The 'label-object' referred to in the function definitions below is either a word, a polyword, or a category—the actual object, not the symbol or string that names it.

```
set-generic-treetop-action (label-object function)          function
generic-treetop-action (label-object)                access function
delete-generic-treetop-action (label-object)                function
list-generic-treetop-actions ()                             function
```

Here is an example from the conjunction grammar. Note that for this to fire, the word "and" cannot have been incorporated into an edge since that would mean that it was not a treetop and would never be seen in the treetop scan.

```
(set-generic-treetop-action word::|and|
                            'conjoin-adjacent-like-treetops)
```

7.4 Traversal actions

The 'traversal action' hook was developed to handle paired punctuation: parentheses, quotations, XML angle-bracket tags, and so on. They are coupled with a hook for 'interior actions' to dictate what to do with the content between the punctuation pair.

The traversal hook is called at the word-level of the scan (word-traversal-hook). [[before?]] It looks for whether the word has an associated function, and if so calls the function passing it two arguments: the positions on either side of the triggering word.

```
set-traversal-action (word function-name)                   function
```

```
delete/traversal-action (word)                                    function
traversal-action (word)                                           function
```

The grammar already defines a set of traversal actions for parentheses, quotation marks, angle-brackets (< >), curly brackets ({ }), and square brackets ([ ]); see grammar/rules/traversal/parenthesis. They all follow a standard pattern: the opening punctation calls a simple function that sets a global variable to mark its position (e.g. *position-of-pending-open-paren*). The function for the closing punctuation clears that flag after it calls the function do-paired-punctuation-interior to handle any associated actions.

7.4.1 Interior actions

The text region between paired punctuation charcters is parser just like any other portion of text, but since all punctuation characters are defined with brackets that start/stop segment scans within them, the parse will stop at the closing punctuation and call its traversal action, which in turn call the standard ‘interior’function.

The first thing it does is analyze the layout of the de-facto segment between the the characters. This may lead to a treetop-parse if there are several edges. If the final result is a single edge then the paired-punctuation interior-action hook is checked.

```
define-interior-action (label type function)                      function
```

The label corresponds to the category of the edge that spans the interior between the pair of punction characters. The type is a keyword indicating which punctionation this pertains to. The function argument is a symbol naming a function. Note that this is not a macro; the values have to already be evaluated. Here is an example.

```
(define-interior-action  category::capitalized-word  :parentheses
  'elevate-fully-spanning-category/parentheses )
```

The function is called by the interior hook with the following arguments in order. The spanning edge, the position before the open character, the position after the close character, the position just after the open, and the postion just before the close.