Computational Linguistics (Spring 2008)
— Exercise 4, Part B (Obligatory) —

Goals

1. Add additional functionality to the chart abstract data type;
2. complete your implementation of the generative chart parser.

Update the Starting Package and Bring up the LKB

(a) As always, on the IFI Linux environment, launch the LKB grammar development and Lisp environment. Several familiar windows appear: the 'LKB Top' window and the Common-Lisp console within emacs.

(b) To implement the generalized chart parser, you will need to obtain an additional file; download the following from the course web site:

`http://www.delph-in.net/courses/08/cl/active.lsp`

Place this file inside your existing '`exercise4/`' directory, replacing the older version of '`active.lsp`' that is there already.

1 A Few More Chart Operations (1 + 2 + 2 = 5 Points)

We need to add a few more functions to our *chart* abstract data type (implemented in the file '`chart.lsp`').

(a) As we move to recording structures of type *edge* in each chart cell (compared to just category symbols, as we used to in the CKY implementation), our approach to avoiding duplicates will change (more details on that in a few weeks). Provide a function `chart-insert()` to add an edge to the chart; the function requires only one argument, of type *edge*, as the edge structure contains components *from* and *to*, indicating the corresponding sub-string indices. In contrast to `chart-adjoin()`, `chart-insert()` does not need to check for uniqueness of edges in a cell, i.e. it can unconditionally simply add its *edge* argument to the cell indicated by the *from* and *to* values inside the edge.

(b) Take a peek at the definition of the *edge* structure towards the bottom of '`chart.lsp`'. Our implementation of chart edges is the same as discussed in class (see the slide copies). Each *edge* has the following components:

- *id* — unique identifier that is automatically assigned by `make-edge()`;
- *from* and *to* — starting and ending string indices, respectively, for this edge;
- *category* — root (aka LHS) category (taken from the set C of the grammar) for this edge;
- *daughters* — list of daughter edges, i.e. RHS elements instantiated successfully already;
- *unanalyzed* — list of remaining RHS elements (β_{i+1} ... β_n) to be instantiated; and
- *alternates* — list of equivalent edges, i.e. edges whose *daughters* derive *category* from *from* to *to*.

Furthermore, in our usual helpful way, we already provide a predicate `passive-edge-p()` to distinguish active (or incomplete) and passive (or complete) edges. Looking further at the *edge* structure —and quite similar to what we did for the original top-down parser—the 'dot' position in the RHS of a rule is recorded through two separate lists: *daughters* is the (possibly empty) sequence of RHS categories found already, and *unanalyzed* is the remaining sequence of RHS categories yet to be found. For an edge to be passive, thus, means that its *unanalyzed* component is empty.

Now, flesh out the implementation of the functions `passive-edges-from()` and `active-edges-to()` in '`chart.lsp`'. The first of these, for example, is to return a list of edges from the chart, comprising (passive) chart entries that start at the given *from* string index. Note that each individual chart cell corresponds to a pair of *from* and *to* values; thus, `passive-edges-from()` actually needs to iterate through the chart row designated by *from*, accumulating all passive edges from all cells in that row. As our chart has a

variable size, depending on the length of the current input provided to the parser, you will need a way of querying the global `*chart*` for its current size: the function `array-dimensions()` can be called on an array and returns a list of integers (two in the case of our two-dimensional chart). These numbers reflect the current dimensions of the array, i.e. the first argument given to `make-array()` when the array was originally created. In the same general spirit, `active-edges-to()` accumulates the sets of active edges for a given *to* string index, i.e. it scans through one complete column of the chart.

(c) Finally, note that our starting package also includes a simple-minded abstract data type implementing an agenda. Much like our own To-Do lists, our agenda implementation is a stack: the element that was added to the agenda last is the first to be returned (unlike our personal agendas, though, nothing ever gets lost). We will interact with the agenda by virtue of the following functions:

- `(agenda-initialize)` — reset the agenda to an empty stack;
- `(agenda-push` *edge*`)` — add *edge* to the top of the agenda;
- `(agenda-pop)` — retrieve the topmost edge from the agenda and take it off the stack.

Feel free to appreciate our agenda implementation in '`agenda.lsp`' (but please ignore the `format()` statements, which are disabled by default, but could be activated for debugging purposes). Or feel free to accept our notion of the agenda as an abstract data type, with the interface functions sketched above as everything one needs to know about it.

2 Generalized Chart Parsing (15 Points)

(a) Go through the function definitions in '`active.lsp`' and fill in the missing parts in the functions `parse()`, `process-passive-edge()`, `process-active-edge()`, and `fundamental-rule()`. In all cases, we have marked the places where we expect you to fill in missing code with the '`???`' marker. Before making your changes, make sure you read (and understand, preferably) all of the comments in the file, as these substitute for more detailed instructions as part of this exercise.

Note Because most of these functions depend on the current state of the chart and agenda (while parsing), it can be difficult to test individual functionality in isolation. In some cases it may be possible to construct test cases for individual functions (often called unit tests), e.g.

```
? (agenda-initialize) → nil
? (chart-initialize 2) → #2a((nil nil nil) (nil nil nil) (nil nil nil))
? (fundamental-rule
   (make-edge :category 'S :from 0 :to 1
              :daughters (list (make-edge)) :unanalyzed '(VP))
   (make-edge :category 'VP :from 1 :to 3
              :daughters (list (make-edge) (make-edge)) :unanalyzed nil))
→ #S(edge :id 5 :from 1 :to 3 ...)
```

In general, however, you should aim to put everything together, perform unit testing where possible, and then test the parser thoroughly on a set of examples of increasing complexity. Remember that it can be useful to `trace()` individual functions (or even add `format()` statements inside of a function definition) to make more of the underlying call sequences visible. For each test, determine what the expected result should be *before* performing the test.

Note For ease of development, we have currently disabled the packing (or factoring) of local ambiguity, i.e. the function `pack-edge()` always returns `nil`. This means that the chart can actually contain 'duplicate' equivalent edges, i.e. ones that span the same sub-string and have the same *category*, while differing in their internal structure. We leave this final refinement of our generalized chart parser for a later exercise.

3 Extending the Grammar (3 + 3 + 4 = 10 Points)

After several weeks of restricting ourselves to adoration of snow, we have enriched our tiny vocabulary at least a little, bringing back our old acquaintance *snores* and adding another verb *shovels*, as well as the noun *lifts* and the preposition *on*. You'll see that some snoring and shoveling is already possible with your current grammar, but one of the analyses we want for *kim shovels snow on lifts* is still missing, viz.

```
(S (NP kim) (VP (V shovels) (NP snow) (PP (P on) (NP lifts))))
```

This structure corresponds to the interpretation 'Kim is using a shovel to move snow onto some (ski) lifts' (presumably so it can be taken away somewhere). Here Kim is not necessarily on a ski lift while shoveling, nor does the snow necessarily start out on the lift. On this reading, the verb takes one NP argument and one PP argument in a flat VP structure with all three as immediate daughters. To build such a VP the parser will need another rule to be added to the grammar.

(a) Enable this rule by opening the file '`rules.tdl`' and deleting the semicolon character preceding the *vp3* rule, then save the file, reload the grammar, and ensure that you now get three parses for the sentence *Kim shovels snow on lifts*. If you don't, review the implementation of your parser, and make the necessary corrections. Then explain in a few sentences how this grammar rule is different from the ones we have used up to now.

(b) The addition of this grammar rule reminds us that because of flaws in the grammar, your parser wrongly accepts a number of strings that are not sentences of English, such as **kim adores*, and the parser assigns too many analyses to some well-formed sentences, e.g. *kim adores snow in Oslo*. Using our newly enlarged vocabulary, construct a list of several ungrammatical strings that our parser wrongly accepts as sentences, and a list of good sentences for which the parser assigns too many analyses.

(c) In a few sentences, provide a diagnosis of the flaws in the grammar, and sketch the specific changes that you would make to the grammar in order to correct these flaws while still providing all of the desired analyses.

4 Submitting Your Results

To provide your results to us, please pack up the entire contents of your '`exercise4/`' directory when you are done—for example as one '`.tgz`' or '`.zip` file. Email the archive file to both André and Stephan before the final deadline, 9:00 in the morning on Monday, March 10. For the IFI Linux environment, we will provide a command-line tool for you to automate the process of submitting results to us. Watch the updates on the course web site for further announcements on how to use our Linux submission tool.