Goals

1. Implement an agenda-driven, active, bottom-up chart parser;
2. read out complete trees from the chart: unpack local ambiguity;
3. augment the grammar to increase ambiguity but reduce overgeneration.

Obtain the Starting Package and Bring up the LKB

(a) As always, bring up bash, obtain a starting package by typing `cvs checkout week4` (in bash), launch emacs, and start the LKB.

(b) Open the files `active.lsp` and `sets.lsp` (from the `week4` directory). Again, we will be fleshing out skeletal (dummy) functions provided in these files, typically by replacing occurrences of ‘???’ in the function bodies with an actual implementation.

1 Implementing the Chart Parser (40 Points)

For this exercise, we have revised the parse chart implementation to make it somewhat more space-efficient. Fortunately, you do not need to know about our revisions, since the chart functionality is encapsulated in an abstract data type. Our chart provides the following functions to you:

- `(chart-initialize n)` — create an empty chart with room for \( n \) input words;
- `(chart-cell from to)` — retrieve the set of edges starting at string index `from` and ending at `to`;
- `(passive-edges-from from)` — retrieve the set of passives edges starting at string index `from`;
- `(active-edges-to to)` — retrieve the set of active edges ending at string index `to`;
- `(chart-adjoin edge)` — add `edge` to the chart (at the indices specified in `edge`); `chart-adjoin()` does not perform a ‘duplicate’ test, i.e. assumes that its caller checks for equivalent chart edges itself.

Our starting package also includes a definition of the `edge` structure as discussed in class (see slide # 54). An `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 (\( \beta_{i+1} \ldots \beta_n \)) to be instantiated; and
- `alternates` — list of equivalent edges, i.e. edges whose `daughters` also derive `category` from `from` to `to`.

We have arranged for `edge` structures to print in the (reasonably readable) format shown on slide # 54; note that although the `alternates` value, for simplicity, is a list of equivalent edges, the pretty printer actually prints the identifiers of `daughters` within each alternate, each sequence (if any) bracketed within curly braces.

Finally, our starting package also includes a simple-minded abstract data type implementing the 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.
(a) Go through the function definitions in ‘active.lsp’ and fill in the missing parts in the functions \texttt{parse()}, \texttt{process-passive-edge()}, \texttt{process-active-edge()}, \texttt{fundamental-rule()}, and \texttt{pack-edge()}. 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.

\textit{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.

\begin{verbatim}
? (agenda-initialize) \rightarrow \texttt{nil}
? (chart-initialize 3) \rightarrow \#((\texttt{nil nil} (\texttt{nil nil}) (\texttt{nil nil}) (\texttt{nil nil}))
? (fundamental-rule
  (make-edge :category 'S :from 0 :to 1 :daughters (list (make-edge)) :unanalyzed '\texttt{\texttt{(VP)})
  (make-edge :category 'VP :from 1 :to 3 :daughters (list (make-edge) (make-edge)) :unanalyzed \texttt{nil)})
\rightarrow (\texttt{#\texttt{[5: \{0-3\} S \rightarrow 0 4 \}])
\end{verbatim}

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 \texttt{trace()} individual functions (or even add \texttt{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.

If you find the effects of local ambiguity packing confusing while debugging your parser, you can temporarily disable ambiguity packing: substitute the call to \texttt{pack-edge()} in the body of \texttt{parse()} with a test that will never be true (\texttt{nil}, for example) to obtain a version of the chart parser without local ambiguity packing (i.e. allowing duplicate entries in chart cells, constructing a set of complete trees rather than a parse forest, and going back to a worst-case exponential complexity).

2 Unpacking Local Ambiguity (20 + 20 Points)

In an earlier exercise you defined a function \texttt{cross-product()} which took as arguments two sets (represented as lists), and returned a list containing all pairs of elements from the two sets; e.g.

\begin{verbatim}
? (cross-product '(1 2 3) '(A B))
\rightarrow ((1 A) (1 B) (2 A) (2 B) (3 A) (3 B))
\end{verbatim}

(a) For the active chart parsing algorithm, it will be convenient to generalize the definition of \texttt{cross-product()} so that it takes as its single argument a list of lists (conceptually, a sequence of sets), and returns a set of all the tuples (again represented as lists), each of which contains exactly one element from each of the input sets; e.g.

\begin{verbatim}
? (cross-product '((1 2 3) (A B) (X Y)))
\rightarrow ((1 A X) (1 A Y) (1 B X) (1 B Y)
(2 A X) (2 A Y) (2 B X) (2 B Y)
(3 A X) (3 A Y) (3 B X) (3 B Y))
\end{verbatim}

Implement this function (in the file ‘sets.lsp’), which we will use while parsing to build candidate combinations of right-hand sides of rules, for rules with an arbitrary number of elements in the right-hand side. So be sure to test your function on lists of sets of varied sizes.

(b) Now look at the definition of \texttt{unpack-edge()} in ‘active.lsp’. Given an edge, \texttt{unpack-edge()} returns a list of one or more trees corresponding to that edge once all ‘packed’ ambiguities have been unfolded, e.g.

\begin{verbatim}
? (unpack-edge (first (parse '(kim adores snow in oslo))))
\rightarrow ([S [NP kim] [VP [VP [V adores] [NP snow]] [PP [P in] [NP oslo]]]]]
[S [NP kim] [VP [V adores] [NP [NP snow] [PP [P in] [NP oslo]]]]])
\end{verbatim}

The implementation of \texttt{unpack-edge()} is a little tricky, since it has to look at both the \texttt{daughters} and \texttt{alternates} components of an \texttt{edge} structure. The general idea is straightforward, though: for edges without daughters, \texttt{unpack-edge()} returns a singleton list containing just the category of that edge (i.e. an ‘atomic’
tree or leaf node). For all other edges, `unpack-edge()` recurses over all daughters, cross multiplies all variations in each daughter position with all the other daughters, and builds as many trees as there are combinations; in addition to unfolding the current edge itself, `unpack-edge()` also needs to recurse over all equivalent edges (for which the current edge acts as a host) and combine the results of unpacking those with the list of trees corresponding to the host edge itself.

As for testing, again go with inputs of increasing complexity (and make sure to enable ambiguity packing, in case you had temporarily eliminated the call to `pack-edge()` for debugging earlier). You may choose to relax the selection of what is returned from the `parse()` function, so as to be able to parse just noun or verb phrases; if you were looking for an edge with a non-empty `alternates` value, consider parsing just `adores snow in oslo`—or use `chart-cell()` with appropriate indices after parsing a full sentence.

3 Extending the Grammar (5 + 5 + 10 + 0 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 `vp2` 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 your 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.

(d) Optional If you are familiar with the LKB grammar development environment, you are welcome to implement those changes in the grammar and test your results with both the built-in LKB parser and your own implementation (which both use the same grammar). In which respect(s) does your parser appear superior to the one in the LKB?

4 Changing the Agenda Policy (0 Points)

Study the implementation of the agenda abstract data type in ‘agenda.lsp’. Produce an alternative implementation that implements a first-in-first-out (FIFO) policy instead of the original stack-based model. Trace the functions `agenda-push()` and `agenda-pop()` with both agenda policies and compare the effects. Summarize your findings in a few sentences.

Submitting Your Results

Since our programs are of some interesting complexity at this point, please make sure to submit the contents of the entire ‘week4’ directory.

Submit your results in email to Dan and Stephan by noon on Tuesday, May 6.