Computational Linguistics (Spring 2010, Exercise 1b)

Goals

1. Learn how to use Lisp structures to create abstract data types;
2. encode, manipulate, and output a simple context-free grammar;
3. study the behavior of a naïve top-down parsing algorithm.

1 Bring up the Editor and the LKB Lisp Environment

(a) As always, when logged into one of the IFI Linux machines, launch the development environment that we will use throughout the semester, by typing (at the shell prompt):

```
lkb &
```

This will lauch emacs, our editor of choice, with the Lisp and LKB running as sub-processes to the editor. The benefits of this set-up will become apparent sooner or later. In emacs, find the buffer named '`*common-lisp*`'. Here you can interact with Allegro Common Lisp, the Lisp system behind the LKB.

(b) Our course setup should have magically copied a new file '`parse.lsp`' into your home directory; open the file in emacs to take a peak at its contents. In the course of this exercise, we will modify and extend the source code in this file. To test your code after *each* modification, you need to make the Lisp environment aware of your changes. Experiment with a combination of (a) just pasting into the '`*common-lisp*`' buffer; (b) using the *Compile and load file* command from the *ACL* menu; or (c) selectively evaluating s-expressions straight from your file via the *C-M-x* (Control + Meta + 'x') keyboard shortcut.

2 Pretty Printing the Grammar

(a) Lisp structures print in a format that can be hard to appreciate by the human eye. Write a function `print-rule()` to take two arguments, a grammar rule and an optional *stream* argument. Note that the syntax for optional function parameters in Lisp is as follows:

```
(defun print-rule (rule &optional (stream t))
  ...)
```

The above means that the function can be called with either one or two arguments. When the *stream* argument is not present in a call to `print-rule()`, then it will have `t` (the 'standard output' stream typically connected to our terminal) as its default value.

(b) Read through the (almost excessive) comments in '`parse.lsp`' and work out how the *rule* structure works. The output of `print-rule()` should look somewhat like this, for example:

```
S --> NP VP
```

(c) Now that we can print individual rules, write a function `print-grammar()` to operate on the global variable `*grammar*` and nicely print out all productions, one per line. Additionally, `print-grammar()` should mark all rules that have the start symbol of the grammar as their left-hand side (LHS) category with a bit of additional decoration, e.g.

```
S --> NP VP [start]
NP --> KIM
NP --> SANDY
NP --> SNOW
VP --> V
VP --> V NP
V --> LAUGHED
V --> ADORED
```

3 Some Theory: Mirror English

Consider the language defined by the following grammar (assuming the conventional start symbol ‘S’):

S → VP NP	NP → *kim*
VP → PP VP	NP → *oslo*
NP → PP NP	NP → *snow*
VP → NP V	V → *adores*
PP → P NP	P → *in*

(a) For each of the following items, identify the number of readings (distinct analyses) that the grammar of Mirror English assigns:

(i) *in oslo snow adores kim*

(ii) *kim adores snow in oslo*

(iii) *snow adores in oslo kim*

(b) Draw the constituent tree for each of the readings.

(c) Where possible, provide one example each of a sentence of Mirror English with exactly (i) three and (ii) four distinct readings.

4 Top-Down Parsing

(a) We have helpfully provided a near-complete implementation of a very simple parser for context-free grammars, following the presentation of top-down parsing in Section 13.1.1 of Jurafsky & Martin (2008). The only missing piece to invoking our parser is the helper function `rules-deriving()`; it seems that the definition of this function somehow got lost in our file ‘`parse.lsp`’. Assuming our default value for `*grammar*`, a call like `(rules-deriving 'vp)` should return a list of two rule structures, viz. those two with the category VP as their left-hand side symbol, in other words: all rules deriving VPs. Replace the ‘`...`’ in our version of the file with your own code. As always, test your new code before you move on.

(b) Even though we will have more to say about the parsing problem, based on your reading of the first few sections in Chapter 13 of Jurafsky & Martin (2008), see what happens when you invoke:

```
? (parse '(kim laughed))
```

Remember that, by default, Lisp will abbreviate large s-expressions when printing the results of evaluation. To see the full return value, wrap the `parse()` call inside a call to `pprint()`. Why do you think `parse()` returns a list of results? Study our comments in ‘`parse.lsp`’ discussing the edge and parse structures. Can you see how each edge instance corresponds to a tree? Draw the full tree(s) corresponding to the result(s) of the above example call to `parse()`.

(c) Write a recursive function that transforms the tree corresponding to an edge into a Lisp list. When using lists to represent trees, we want the `first()` to correspond to the mother node, while the `rest()` corresponds to the list of daughter nodes. For example:

```
? (edge-to-tree (parse-edge (first (parse '(kim laughed)))))
→ (S (NP (KIM)) (VP (V (LAUGHED))))
```

5 Recursion in the Grammar

(a) Try to get a better understanding of how the top-down parser works. One useful technique might be to augment the definitions of `parse()` and `extend-parse()` with a `format()` statement right at the top of the function body. Make each function print the arguments with which it was called, for example:

```
parse(): input: (KIM LAUGHED); goal: S
extend-parse(): goal: S; analyzed: NIL; unanalyzed (NP VP); input: (KIM LAUGHED)
parse(): input: (KIM LAUGHED); goal: NP
extend-parse(): goal: NP; analyzed: NIL; unanalyzed (KIM); input: (KIM LAUGHED)
parse(): input: (KIM LAUGHED); goal: KIM
...
```

Note that for such debugging output, it may be convenient to not print the actual `analyzed` values given to each call of `extend-parse()`, but instead to print the more compact result of invoking `edge-to-tree()` on each element of the `analyzed` list. Make sure you understand what happens at each level of recursion, and how the arguments to later invocations of `parse()` and `extend-parse()` result from earlier calls. In your own words, what are the pieces of information in the structures returned by the `parse()` function, and why are they needed?

(b) Our starting grammar has no way of analyzing inputs like, for example, *Kim laughed with Sandy.* Assuming that the category for *with* is P (for preposition), add the rules necessary to (a) include *with* in our grammar, (b) combine a preposition followed by an NP to form a PP (prepositional phrase), and (c) allow PPs to attach to both NPs and VPs, where in either case the LHS category is unchanged, i.e. again NP or VP, respectively. Before you explore the world of parsing with recursive grammar rules in its full beauty, test your PP rules in isolation. Does a call like `(parse '(with) 'P)` yield the expected result(s)? What should be the return value for this call?

(c) Try parsing an input including a PP. What happens? Given the helpful logging output from part (a) above, can you work out what causes the behavior that you see? Draw the tree(s) that you think would have been the correct result(s), and explain in a few sentences why our top-down parser fails so miserably in solving this particular problem.

6 Submitting Your Results

Please submit your results in email to Stephan (‘`oe@ifi.uio.no`’) and Arne (‘`arnskj@ifi.uio.no`’) before midnight on Monday, March 8. Ideally, please provide a single file (an extended version of our ‘`parse.lsp`’) including your code and answers to the questions above. Note that it is good practice to generously document your code with comments (using the ‘`;`’ character, making the Lisp system ignore everything that follows the semicolon).