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

Goals

1. Manipulate arrays: implement the parsing chart;
2. implement the Cocke-Kasami-Younger algorithm.

Obtain the Starting Package and Bring up the LKB

(a) As always, on the IFI Linux environment, launch the LKB grammar development and Lisp environment that we will use throughout the semester, by typing (at the shell prompt):

```
lkb &
```

Several new windows appear: the 'LKB Top' window and a Common-Lisp console (running as a subprocess to emacs, our editor of choice).

(b) For the first time this semester, we will start using some of the LKB functionality; while we implement our own versions of the CKY and generalized chart parsers, our grammar is also accessible to the LKB, and we can use its built-in parser (and visualization tools) as a point of reference. Our course setup has magically created a new directory '`exercise4/`', right below your home directory. Throughout this exercise, we will be making changes to the files in '`exercise4/`'. In the LKB Top window, load our baby grammar by selecting *Load – Complete grammar*, then double-clicking on the directory '`exercise4/`' and on the file '`script`' in that directory. Reassuring messages will appear in the Lkb Top window, confirming that the grammar is loaded.

(c) Inspect the grammar briefly by selecting *Parse – Parse input* in the LKB Top window, then typing *Kim adores snow* into the input window that pops up. Click on 'Ok' to start the parser and expect a new window to appear, displaying the parse tree for this sentence. We will have more to say about the LKB parser and additional functionality later in the semester.

(d) In emacs, open the files '`chart.lsp`', '`sets.lsp`', and '`cky.lsp`' (which all are in the '`exercise4/`' directory, of course). They contain skeletal (dummy) functions that we will flesh out throughout this exercise. Often we provide 'skeleton' function definitions for you, where we ask that you replace the '`???`' placeholders in the function body with an actual implementation of the necessary functionality. Please make all of your changes to these files, save everything, and select *Load – Reload grammar* from the LKB Top window; this will re-load the full package, including your modified Lisp code, and will print diagnostic messages to the LKB Top window (and sometimes also the `*common-lisp*` buffer in emacs). After each reload, check carefully for warnings or errors that may have been caused by your changes; always correct load-time errors before moving on. Note however, that our starting package will generate warning messages about unused variables in function definitions, as long as we have not replaced the '`???`' placeholders in the code.

Note that the LKB system is implemented in Lisp and shares the same 'universe' with your own code. Hence, even though the *Load – Reload grammar* command is a convenient way of making sure everything is (re-)loaded, it continues to be true that you can program and debug interactively, for example pasting a single function definition into the `*common-lisp*` buffer, using the *Compile and load file* command from the *ACL* menu in emacs (to re-load a single file), or sending a single s-expression for evaluation to Lisp through the *C-M-x* (Control + Meta + 'x') keyboard shortcut.

Finally, note that the global variable `*grammar*` and the *rule* structure are defined exactly as in the previous exercise. Only, now that we are using the LKB, the value of `*grammar*` is (magically, for the time being) initialized for us by the system.

1 Implementing the Parse Chart (4 + 4 + 7 = 15 Points)

For reasons of tradition, we count inter-word spaces, rather than words, when referring to sub-strings of input to the parser; for example:

$_0$ *Kim* $_1$ *adores* $_2$ *snow* $_3$

Thus, the first word of an input sequence is said to extend from position 0 to position 1. These positions are often referred to as *string indices*, even though we are using a list of symbols as the input to our parsers rather than an actual string.

From now on, all of our parsers will require a parsing chart to record intermediate results. We will realize the chart as a two-dimensional Lisp array but encapsulate the internals of our implementation in an abstract data type. Internally, the chart will be held in a global variable `*chart*` and manipulated using the Lisp array operations; externally (i.e. in our parser implementation(s)), however, all chart access will be exclusively through a suite of functions provided by our abstract data type.

Note: In the current implementation of the parse chart, we deliberately choose to use slightly more space than would be needed (in order to make our implementation simpler). While for the CKY algorithm all that is needed is the upper right (triangle) corner of a matrix, we actually use a complete quadratic array internally; also, column 0 of the chart never gets used in the CKY algorithm, but yet our chart implementation provides cells for it. The encapsulation of the actual chart implementation into an abstract data type ensures that one could move to a more space-efficient (internal) implementation of the chart at a later point without affecting its external functional interface.

(a) Each time the parser is called on an input string, it needs to initialize a chart, i.e. allocate the appropriate number of cells and make sure that each cell is empty. In '`chart.lsp`', implement the function `chart-initialize()` that takes one argument (n, say, the number of words in the input string), and initializes the global `*chart*` to a two-dimensional array with ranks from 0 to n in both dimensions. All cells in `*chart*` will initially contain `nil`. `chart-initialize()` may return the fresh chart, but it is entirely called for its side effect, viz. binding an empty chart, large enough for the current input, to the global variable `*chart*`.

```
? (chart-initialize 2)
→ #2A((nil nil nil) (nil nil nil) (nil nil nil))
```

(b) At various points, our parsers need to retrieve the contents of one cell from the chart, given a row and column index. Write a function `chart-cell()` that takes two arguments, a row and column index, and returns the contents of the cell at the corresponding position, e.g.

```
? (chart-cell 1 2)
→ nil
```

Since, at this point, all of our chart cells contain `nil` only, we defer more substantive testing of `chart-cell()` until we have a way of altering the cell contents.

(c) Finally, while filling in the cells of the chart during parsing, we will need a function to, again given a row and column index, add to the contents of a cell. However, each cell in the chart has the semantics of a set (although implemented as a regular list), i.e. there must not be duplicate elements. Implement the function `chart-adjoin()` that takes three arguments, a row and column index plus some object, and adds the object to the specified cell, unless it is already there.

```
? (chart-adjoin 1 2 'S)
→ (S)
? (chart-cell 1 2)
→ (S)
? (chart-adjoin 1 2 'VP)
→ (S VP)
? (chart-cell 1 2)
→ (S VP)
? (chart-adjoin 1 2 'S)
→ (S VP)
```

? `(chart-cell 1 2)`
→ `(S VP)`

Note: Since `chart-adjoin()` is one of the functions primarily called for its side effect, it may be legitimate for your implementation to return a value that is slightly different from the examples above; likewise, wherever we use lists to model sets, the order of elements—by definition—does not matter. To implement the set semantics in adjoining an element to a cell, consider re-using some of the general set operations that we developed earlier in the course; see '`sets.lsp`'.

2 Another Set Operation (15 Points)

(a) In '`sets.lsp`', implement the function `cross-product()` that takes two sets as its arguments; the return value of `cross-product()` is a list containing all pairs of elements from the two input sets, e.g.

? `(cross-product '(1 2 3) '(A B))`
→ `((1 A) (1 B) (2 A) (2 B) (3 A) (3 B))`

(b) *Optional* For your amusement and ours, but no extra credit, implement another variant of `cross-product()`. If your first solution uses a recursive control structure, then use `loop()` now, or vice versa.

3 The Cocke-Kasami-Younger Algorithm (5 + 10 + 25 = 40 Points)

For your convenience, here is the schematic version of the CKY parsing algorithm as it was presented in the lecture:

```
for (0 ≤ i < |input|) do
  chart_[i,i+1] ← {α | α → input_i ∈ P};
for (1 ≤ l < |input|) do
  for (0 ≤ i < |input| − l) do
    for (1 ≤ j ≤ l) do
      if (α → β1 β2 ∈ P ∧ β1 ∈ chart_[i,i+j] ∧ β2 ∈ chart_[i+j,i+l+1]) then
        chart_[i,i+l+1] ← chart_[i,i+l+1] ∪ {α};
```

As is often the case, our pseudo-code notation uses syntactic conventions to condense operations that will typically need to be made explicit in an actual implementation, e.g.

- '$input_i$' refers to the word at position i in the input sequence (a list of symbols);
- '$chart_{[i,j]}$' refers to the chart cell at row index i and column j;
- '$|input|$' denotes the length of the input sequence, i.e. the number of input words;
- 'for $(0 \leq i < 42)$ do' means a loop of a variable i ranging from 0 (inclusive) to 42 (exclusive);
- '$\alpha \rightarrow \beta \in P$' means that the grammar contains a rule rewriting α as β;
- '$chart_{[i,j]} \leftarrow \{\alpha\}$' means that the chart cell at $[i, j]$ is assigned the value $\{\alpha\}$;
- '$X \wedge Y$' (e.g. as the test in an if statement) means that both X and Y need to be true.

(a) In the body of `cky-parse()`, write three nested `loop()` constructs, using counter variables l, i, and j, respectively, to capture the basic control structure of the CKY algorithm. To validate the control structure, put the following into the body of the innermost `loop()`:

```
(format t "[~a,~a] <-- [~a,~a] + [~a,~a]~%" i (+ i l 1) i (+ i j) (+ i j) (+ i l 1))
```

At this point, you should be ready to evaluate, say,

? `(cky-parse '(kim adores snow in oslo))`

and compare the printed output to the lower left corner of the slide on CKY parsing. Make sure that the first and last few lines of your output match what is shown on the slide (i.e. all cell indices are identical and occur in the same order).

(b) Next, implement the body of `cky-initialize()`, the function that initializes the chart with 'lexical' categories. For each word of the input sequence, `cky-initialize()` looks up all rules of the grammar that have that word as their right-hand side member; for each such rule, the left-hand side category is added to the corresponding cell in the chart.

Test the functionality of `cky-initialize()` by parsing *Kim adores snow in Oslo* again. As `cky-parse()` returns the final chart, you should be able to see non-`nil` elements along the diagonal at this point.

(c) Finally, we need to substitute the actual cell computation (instead of our `format()` placeholder) in the body of `cky-parse()`. For increased modularity, implement the part of the algorithm that computes new cell combinations (i.e. the bottom-most two lines of the pseudo-code presentation) as a separate function `cky-augment()`. This auxiliary function takes six arguments, corresponding to the row and column index of the cell that is about to be augmented (the indices to the 'left' of the assignment operator '←'), and row and column indices, respectively, of the two cells corresponding to the right-hand side elements of each combination. Informally, we can think of the former cell as the 'target' for `cky-augment()`, and consider the latter two cells the 'source' of the computation.

Internally, `cky-augment()` retrieves the contents of the two 'source' cells (as specified by its third to sixth arguments), cross multiplies the sets of categories from these two cells, and then looks for rules that have one of the pairs in the cross product of categories on their right-hand side. For each such rule, `cky-augment()` adjoins the left-hand side of the rule (i.e. the category that can be derived by combining two categories from the 'source' cells) to the 'target' cell (as specified by the first two arguments).

Note: In looking up all rules that have a specific right-hand side, you might consider another auxiliary function, e.g. `rules-rewriting-as()`, that determines the list of rules from the grammar that have the exact RHS that we are looking for, at each point. Revisit `rules-deriving()` as a possible role model for this function; generally, 'programming by example' can be a viable paradigm when the task you are trying to accomplish is similar to one you have solved (or had solved for you) earlier.

4 Generalized Chart Parsing (30 Points)

We will make the final part of this assignment available on Friday, February 29. Retrieve it from the course web page and move from the CKY algorithm to a generalized chart parser. The file '`active.lsp`' already provides the skeleton of our implementation, and we will provide more instructions on how to fill in the remaining '`???`' placeholders.

5 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, 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.