CS107 Handout 54
Autumn 2000 December 1, 2000

RSG in LISP

This handout was written by Julie Zelenski.

Given that programming languages are designed to have different strengths and weaknesses, directly comparing them doesn't often lead to interesting information. Trying to have such a discussion often just breaks down into religious arguments about which language is "better". "Better" isn't all that useful a concept in this case, you need to be able to balance the various tradeoffs. No one language can be the best tool for all tasks. CS107 is designed to help you gain the perspective to evaluate a language and pick the right tool for a particular job at hand.

LISP is much safer than C, but as a result is less efficient. LISP has many more built-ins and a lot of type flexibility which leads to greater expressiveness than C but it also doesn't allow some of the low-level manipulations you can do in C (such as directly manage your own memory and pointers) that makes it more restrictive. And no matter how much effort you put into it, LISP code is hard to read, period.

It can be interesting to take the same problem and try it to construct solutions in various languages as an exercise in comparison. To that end, here's a small example that shows how we might go about writing the RSG in LISP.

First off, let's assume we cheated and already had the grammar data pre-organized into a nested list structure. It turns out LISP has an incredibly extensive set of I/O routines (far beyond the `printf`/`scanf` facilities in C), but it is somewhat complicated to use, so we'll skip that and focus on the storing and expanding parts of the problem. Here is the simple poem grammar in the LISP data format:

```
(defconstant poem-grammar '(
   (start                              ;; non-terminal name is a symbol
      (("The" object verb " tonight.")))  ;; list of productions

   (object
         ((" waves")             ;; terminals are strings
          (" big yellow flowers")
          (" slugs")))

   (verb
         ((" sigh" adverb)
          (" portend like" object)
          (" die" adverb)))

   (adverb
         ((" warily")
          (" grumpily")))))
```

Just as before, you will start the whole process by expanding the `start` non-terminal and working recursively from there. Here are a few of the possible expansions of the poem grammar:

```
   ? (rsg poem-grammar)
   "The big yellow flowers sigh warily tonight."

   ? (rsg poem-grammar)
   "The slugs portend like waves tonight."

   ? (rsg poem-grammar)
      "The slugs die grumpily tonight."
```

Here is one possible LISP solution for the RSG. It is written very much in the functional vein— we write small functions that evaluate to an expression that we use as a piece of a larger expression when synthesizing the full answer. Just 6 short functions handle all the lookup/expansion work. No need to write a DArray first, the built-in list is a killer DArray replacement (there is also a hashtable among the built-ins if we wanted to get fancy). Notice there are more lines of comments than code—get used to that!

```
;; A recursive function to search a grammar for a non-terminal and
;; return its associated list of productions. Returns nil if not found.
;; Uses some of the cheesy car-cdr composition functions (see on-line manual
;; if you're curious) to pick various components out of the lists.
;; Expects a non-term (symbol) and a grammar in the standard format of
;; ((non-term ((production1) (production2))) where production is a list
;; of terminals (strings) and more non-terminals (symbols).

(defun find-productions (non-term grammar)
  (if (null grammar) '()
      (if (equal (caar grammar) non-term) (cadar grammar)
          (find-productions non-term (cdr grammar)))))


;; Given a list of anything, randomly chooses and returns one elem from list
;; Works for any non-empty list of any type of elements (even non-homogeneous)
;; Note how much easier using Lisp nth is compared to ArrayNth (no
;; casting, no chance to interpret the data wrong!)

(defun pick-one (list)
  (nth (random (length list)) list))


;; Given an elem (be it term or non) returns a string which represents
;; either just the terminal itself, or looks up the non-term's productions
;; and randomly chooses one to recursively expand using expand-production
;; and returns the flattened string from that

(defun expand-one (elem grammar)
  (if (stringp elem) elem
      (expand-production (pick-one (find-productions elem grammar)) grammar)))

;; Maps expand-one over every element in the production, then concatenates
;; the resulting strings into one combined string

(defun expand-production (production grammar)
  (concatenate-strings
      (mapcar #'(lambda(elem) (expand-one elem grammar)) production)))
```

```
;; Quick little function to merge together a list of strings into one
;; long string at the end of the expansion
(defun concatenate-strings (list-of-strings)
  (apply #'concatenate (cons 'string list-of-strings)))

;; Here ya go, the RSG:
(defun rsg (grammar)
  (expand-one 'start grammar))
```

What are the advantages and disadvantages of solving this problem in LISP instead of C? Which would you prefer to use? Would developing this program in LISP be a more or less pleasant experience than it was in C? How does Java fit into this comparison?