Introduction to Artificial Intelligence
CISC-481/681

Program 0: Simple Lisp Programs

Reading:

- Handouts Summarizing Common Lisp Functions
- Paul Graham's ANSI Common Lisp: Read Chapters 2, 3, 4, 5.1–5.4, 7.1–7.3, 14.5. This is about all you'll need for the entire course.

Lisp Practice

Exercise 1 Write a function called **palindromep** that is given a list and returns t if the list has the same sequence of elements when read left to right as when read right to left. It should return NIL otherwise. This is a 2-line program, if you use the correct two functions. One is an equality function (either EQ, EQL, EQUAL, or =). Please explore and understand those functions—we will use them often in the future.

```
> (palindromep '(a b c b a))
T
> (palindromep '(go spot go))
T
> (palindromep '((x y z) (e f g) (e f g) (x y z)))
T
>(palindromep '(madam im adam))
NIL
> (palindromep '(x))
T
```

Exercise 2 Write a function called **presentp** that takes two arguments, an atom and a list. it should return non-nil if the atom appears anywhere in the list, and NIL if it does not appear. **presentp** is like member except that the atom can appear anywhere in the list, not just at top level. You should use recursion in your solution.

```
> (setf formula '(sqrt (/ (+ (expt x 2) (expt y 3)) 2.0)))
(sqrt (/ (+ (expt x 2) (expt y 3)) 2.0))
> (presentp 'x formula)
T
> (presentp 'z formula)
NIL
```

Exercise 3 Write a Lisp function **duplicate-entries** that takes a list and returns true if it contains multiple copies of some entry, and NIL otherwise. Be sure to handle nested lists properly. You can do this recursively, or my using the mapping functions. For example, `maplist`. Did you know that `member` takes an optional argument, `:TEST`, followed by a function to be used for testing equality? For example, `(member thing1 thing2 :TEST #'equal)`. This will often be VERY useful.

```
> (duplicate-entries '(a b a c d))
T
> (duplicate-entries '(a b (a) c d))
NIL
> (duplicate-entries '((a b) b c (a b)))
T
```

Exercise 4 Define a function **occurrences** that takes a list and returns an assoc-list indicating the number of times each `eql` element appears, sorted form the most common element to the least common element (hint: Common Lisp defines an extremely efficient `sort` function).

```
> (occurrences '(a b a d a c d c a))
((A . 4) (C . 2) (D . 2) (B . 1))
```

Exercise 5 Why does `(member '(a) '((a) (b)))` return nil?

Exercise 6 What do the global lisp variables `*print-level*` and `*print-length*` do?

A Simple Example Lisp Program

Study $CLASSHOME/prog1.lisp[1]. The main function here is `generate`, and the style of programming used is called *data-driven* because the data (the list of rewrite rules pointed to by the variable `*grammar*`) drives what the program does next. It is a natural and easy-to-use style in Lisp, leading to concise and extensible programs, because it is always possible to add a new piece of data with a new association without having to modify the original program. Here are some examples of `generate` in use (try this yourself):

[1]From Peter Norvig's *Paradigms of AI Programming*

```
CL-USER 2 : 2 > (generate 'sentence)
(THE TABLE SAW A BALL)
CL-USER 3 : 2 > (generate 'sentence)
(THE BALL LIKED THE MAN)
CL-USER 4 : 2 > (generate 'noun)
(BALL)
CL-USER 5 : 2 > (generate 'noun-phrase)
(A WOMAN)
CL-USER 6 : 2 > (generate 'verb-phrase)
(TOOK THE WOMAN)
```

Another advantage of representing information in a declaritive form—as rules or facts rather than as Lisp functions—is that it can be easier to use the information for multiple purposes. We can easily write a program that generates not only the sentence, but the complete parsed syntax of the sentence:

```
CL-USER 8 : 2 > (generate-tree 'sentence)
(SENTENCE (NOUN-PHRASE (ARTICLE THE)
                       (NOUN WOMAN))
          (VERB-PHRASE (VERB LIKED)
                       (NOUN-PHRASE (ARTICLE THE)
                                    (NOUN TABLE))))
```

Or we can generate all the possible rewrites of a phrase

```
CL-USER 9 : 2 > (generate-all 'noun-phrase)
((THE MAN) (A MAN) (THE BALL) (A BALL) (THE WOMAN) (A WOMAN) (THE TABLE)
 (A TABLE))
CL-USER 10 : 2 > (length (generate-all 'sentence))
256
```

However, the function `generate-all` will only work for finite languages (note that the bigger grammar in the file defines an infinte language using recursive rules).

Exercise 7 Switch to the bigger grammar in the file and generate a few sentences. Create your own grammar (for another natural language, or a subset of a computer language) and generate several sentences or programs.

Excercise 8 Rewrite `generate` so that it still uses `cond` but it avoids calling `rewrites` twice.

Excercise 9 Write a version of `generate` that explicitly differentiates between terminal symbols (those with no rewrite rules) and nonterminal symbols. For example, write a predicate function `non-terminal-p` and rewrite `generate` to use your new function.

Your Own Short Lisp Program

Excersise 10 Write a Lisp function ask-and-tell that processes simple sentences. It should allow you to enter lists that represent assertions, such as (Mary likes hiking) or (Steve hates pretzels), as well as questions, such as (Does Mary like hiking) or (Does Steve like pretzels). The assertions should all have a proper name (e.g., Mary) in the first position, the word likes or hates in the second position, and either an activity (e.g., hiking) or an object (e.g., pretzels) in the last position. The questions should be similar, except that they begin with the word Does. The function should remember the assertions you give it, and answer your questions appropriately. If you repeat an assertion, it should let you know that; if you enter a contradictory assertion, it should alert you to that, and confirm that you're sure before making the change. E.g.,

```
>(ask-and-tell '(mary likes hiking))
"OK"
>(ask-and-tell '(steve hates pretzels))
"OK"
>(ask-and-tell '(does mary like hiking))
"YES"
>(ask-and-tell '(does mary like pretzels))
"I DON'T KNOW"
 >(ask-and-tell '(mary likes hiking))
"I KNOW THAT ALREADY"
>(ask-and-tell '(does steve like pretzels))
"NO"
>(ask-and-tell '(mary hates hiking))
"YOU'VE CHANGED YOUR MIND.  ARE YOU SURE?"    no
"OK"
>(ask-and-tell '(does mary like hiking))
"YES"
>(ask-and-tell '(mary hates hiking))
"YOU'VE CHANGED YOUR MIND.  ARE YOU SURE?"  yes
"OK"
>(ask-and-tell '(does mary like hiking))
"NO"
```

One easy way to do the database is to use Lisp's built-in hash table operations (see chapter 4.8 in Graham). Put the hashed DB in a global variable.