CS262 Artificial Intelligence Concepts

Worksheet 4

In this worksheet we look at input/output functions in Lisp.

1 Printing

The function `print` takes one argument and prints the value of its argument onto the screen *and* returns the value of the argument. For example:

```
>(print 'hello)

HELLO
HELLO

>(print "hello")

"hello"
"hello"

>(print (first '(hello john)))

HELLO
HELLO

>(cons 'hello (print '(john)))

(JOHN)
(HELLO JOHN)
```

In each of these examples, the first line of the output was produced by the call to `print` function, and the second line was the value returned from the expression. The distinction between the value returned by a function and a printed value is an example of a very basic distinction in Lisp: the distinction between values of functions and *side effects* of functions.

$\Longrightarrow$ **Every function in Lisp returns a value, but some functions perform additional actions, which are called side effects.**

Printing is a side effect of `print`. We have already came across two other operators in Lisp which have side effects: `setq` and `defun`. The side effect of `setq` is to assign a value to a variable. The side effect of `defun` is to define a new function. While each of these three operators retuns a value, we use them mainly because of the side effects they perform.

`princ` behaves very similar to `print`, but prints strings without quotation marks. Compare:

```
>(print "Chris travelled to Germany last year.")

"Chris travelled to Germany last year."
"Chris travelled to Germany last year."

>(princ "Chris travelled to Germany last year.")
Chris travelled to Germany last year.
"Chris travelled to Germany last year."
```

Notice that `print` gives a new-line before printing and a space afterwards, while `princ` only prints the argument.

1.1 Multiple actions in a function

So far all the function definitions we used contain only one action. However, we can actually use `defun` to write a function that performs multiple actions. You can simply list one action after another to carry out these actions in order. For example:

```
(defun print-sum (num1 num2)
    (princ num1)
    (princ " plus ")
    (princ num2)
    (princ " equals ")
    (princ (+ num1 num2))
    (terpri)
    "Thank you.")
```

The function `terpri` produces a new line. Notice that in this function, the fact that a string evaluates to itself is exploited as the returned value of `print-sum` to print out `"Thank you."`.

```
>(print-sum 3 6)
3 plus 6 equals 9
"Thank you."
```

1.2 Better control over layout

The function `format` offers a range of facilities that can remove the need for lots of `princ` statements. For the above example, we could have used:

```
(defun print-sum (num1 num2)
    (format t "~a plus ~a equals ~a ~\% Thank you" num1 num2 (+ num1 num2)))
```

This gives the same result as for above except that the final line is not in quotes. The general form of `format` is as follows:

(format <destination> string [lisp-objects ...])

The destination is set to `t` to make `format` print, otherwise it is set to `nil` and the print string is returned as a value. The given string is printed with object values placed, in order, according to the directive `~a`. Another directive, `~\%`, is used to generate a new-line. For an example of the use of `format` see the `sums.lisp` program in worksheet 1.

2 Input

The most useful input function is `read`. Consider the example below, `read-try`, which takes one argument and reads two inputs from the keyboard. The function adds together its argument and the first value it reads and then makes a list of that sum and the second value it reads.

```
(defun read-try (num)
   (list (+ num (read)) (read)))
```

As you can see, `read` does not take any arguments. Each time `read` is called it monitors the keyboard, gets whatever Lisp expression the user types, and returns that expression. In the example below, the user types in the underlined lines:

```
> (read-try 8)
14
books
(22 books)
```

Notice that when you type in an input to `read`, it is not quoted: this is because `read` does not evaluate its input. Notice also that Lisp does not print any prompt on the screen when it is awaiting input for `read`.

3 Saving interactive sessions

The function `dribble` is used to record to file an interactive session with the interpreter. You may find this useful for preparing assignments or debugging. `dribble` is turned on by giving a file name and turned off by repeating without any argument. For example:

```
>(dribble "file-x")
Starts dribbling to file-x (1996/10/2, 2:4:20).
NIL

>(setq a 23)
23

>(* 3 a)
69

>(dribble)
Finished dribbling to file-x.
NIL

>a
23
```

Exercises

1. Write a function called `average-number` which takes no arguments, but takes three inputs and returns the average of those inputs.

2. Write a function that accepts one argument, that must be a list. The function should prompt for an input from the user and then check to see if the input is a member of the argument list.