CS262 Artificial Intelligence Concepts

Worksheet 1

1 Getting Started with Lisp

In this first practical you will learn about some basic functions provided by Lisp and how to call them. In later practicals you will learn about more complex Lisp data structures and predefined functions for operating on them, and how to define your own functions. Section 3 is a description of how to enter the Lisp environment, but first you should read and understand the following description of basic Lisp notions, and complete the exercises included along the way.

⟹ **Important points will be indicated thus.**

1.1 Functions

The Lisp interpreter is an interactive system, this means that when you are in the Lisp environment you can type in function calls and Lisp will execute them and respond immediately by putting the result on your screen. For example:

```
> (+ 2 3)
5
>
```

Here, the > is the Lisp environment prompt, which tells you that it is ready to accept a function. The user has typed `(+ 2 3)` and the Lisp interpreter has responded with `5`. Here we say that `+` is a *function*, and `2` and `3` are *arguments*, and that Lisp *returned* the result `5` – remember this terminology as we will use it from now on.

⟹ **In Lisp, programs = functions.**

Other Lisp arithmetic functions:

Function calls	Value returned	Description of operation
`(+ 1 2 3)`	`6`	Adds all the arguments
`(- 34 23)`	`11`	Subtract second argument from first
`(* 10 2)`	`20`	Multiply the arguments
`(/ 24 6)`	`4`	Divide first by second

You'll notice from the descriptions above that some functions (such as `+` and `*`) can take take any number of arguments, whereas others are restricted as to the number of arguments they can have, for example `/` and `-` can only have two arguments. It is possible to describe the general form of a function call in the following template notation:

(<function-name> [arguments ...])

The symbol `<function-name>` that follows the right parenthesis indicates that a function name is required. Square brackets around a symbol represents an optional component, and the three dots that there may be other optional components, remember though that some Lisp functions require specific numbers of components.

Exercise 1

1. Write a Lisp function call that adds 5 and 7 together.
2. Write a Lisp function call that divides 4 by 2.
3. Write a Lisp function call that multiplies 4, 6 and 7 together.

1.2 Embedded functions

It might be the case that you want to add two numbers together and then divide the result by another number. For example, you could add 38 and 85 and divide the result by 41 by doing the following:

```
> (+ 38 85)
123
> (/ 123 41)
3
>
```

However, instead of doing this in two steps, Lisp allows us to *embed* the first function call in the second, which leads to a single and much more compact function call:

```
> (/ (+ 38 85) 41)
3
>
```

The notion of embedding function calls is fundamental to Lisp programming. In the example above we can say that for the first time we have a *program* that consists of two instructions instead of one; one to add two values and another to divide the result of the addition by another value. Whenever a function is called any embedded functions are evaluated (ie. called or executed to produce value(s)) **before** the function is applied to the arguments.

$\Longrightarrow$ **Given (F A1 A2 A3 ...) the interpreter first evaluates all the As (arguments), then the function F is evaluated on (applied to) the argument values).**

Exercise 2

1. Multiply together the number 5 and the result of 3 minus 2.
2. Write a function call that will add 60 to 40 and divide the result by 4.

1.3 Types in Lisp

The functions discussed so far only accept numbers as arguments and perform arithmetic operations on them. However Lisp was specially designed for artificial intelligence programming and, in particular, for building programs that can perform non-numerical reasoning. In natural language processing, for example, we must be able to represent the words in a sentence and their meaning; these types of representations are referred to as symbolic expressions. In Lisp there are two types of symbolic expressions: *atoms* and *lists*.

$\Longrightarrow$ **All legal expressions in Lisp consist of only atoms and lists.**

An atom is a simple element, for example: `george 16 b10 + first-part very-long-atoms` are all legal atoms. As can be seen from this an atom is a string of numbers and/or punctuation marks. Atoms can help us represent non-numerical information, for example, if we want to represent chess pieces we could just use the atoms `king`, `queen` etc. Numbers are special atoms and can only be used to represent the corresponding numerical value.

$\Longrightarrow$ **Numbers evaluate to themselves.**

Lists are just as important as atoms (Lisp stands for "LISt Processing language"). The following are examples of lists:

```
(2 4 6)
(cat (dog fred) bike house)
(george)
()
(+ 3 2)
(egg ((bacon bacon) sausage cornflakes))
```

A list is a sequence of items or *elements* enclosed in a pair of parentheses. Each element itself may be either an atom or another list. A list with no elements is called an *empty list* or *nil*. Finally you must remember that the order of a list is very important, and that the parentheses of the list must be balanced, that is, the number of opening parentheses is always equal to the number of closing parentheses.

Exercise 3

1. How many elements do the above lists have?
2. Consider the following list: `((a c)(b (g r) h) j g)`:
   (a) What is the first element?
   (b) What is the first element of the list that is the second element?

1.4 Variables, values and bindings

Now consider the following session:

```
> '(+ 28 25)
(+ 28 25)
> '(cat dog)
(CAT DOG)
> (quote (cat dog))
(CAT DOG)
>
```

$\Longrightarrow$ **The quote mark tells the Lisp interpreter to treat an expression as a *literal expression* and *not* evaluate it.**

For example, the quote in `'(+ 28 25)` tells Lisp not to evaluate this as a call to some function named + (which does addition) but instead to treat it simply as a list of three atoms. The quote mark is actually a (very convenient) shorthand for the function QUOTE, thus '(c d f) is **identical** to (quote (c d f)).

There is a function called SET that assigns values to atoms, but we normally use SETQ or SETF. These take 2 arguments, the first being an atom (the variable) and the second being its value.

```
> (setq a '(sheep fish game))
(SHEEP FISH GAME)
> a
(SHEEP FISH GAME)
> (setq b '(+ 32 13))
(+ 32 13)
> a
(SHEEP FISH GAME)
> b
(+ 32 13)
>
```

Notice that the first argument is not quoted. This is another shorthand (i.e. `(setq a 'b)` is the same as, but simpler than, `(set 'a 'b)`).

We refer to the connection between atoms and their values as **bindings**. Thus an error message like **"The variable x is unbound"** means that no value has been set for `x` (either by the user or by a function). Several values can be set up by `setq`:

```
> (setq a 3 b 2 c 4)
4
> a
3
> (+ a b)
5
> (+ a b c)
9
> (+ 2 b 10)
14
>
```

1.5 Functions for operating on lists

Lists enable us to represent complex information, but we need to have functions to allow us to construct and extract the various elements of a list. For example, if we have a list that represents the positions of footballers on a playing field, then we need functions that will allow us to extract the position that a particular player is in. The table below introduces four basic functions for operating on lists:

Function calls	**Value returned**	**Description of operation**
`(car '(c d f))`	`c`	Return the first element of a list
`(cdr '(c d f))`	`(d f)`	Return the list with the first element removed
`(cons 'c '(d f))`	`(c d f)`	Insert the first argument at the beginning of the list that's the second argument
`(list 'c '(d f))`	`(c (d f))`	Make a list out of the arguments

These functions are very important and you should be particularly sure that you understand their operation before going on to the next practical sheet.

1.5.1 Extracting information from lists: *car* and *cdr*

The first two functions allow us to examine the contents of lists. The function `car` always takes one argument, which must be a list, and returns the first element in the list. The function `cdr` accepts one argument, which must be a list, a returns the list with the first element removed (we call this the *tail* of the list).

Consider the expression: `(car (cdr '((a b) c d)))`

Remember that Lisp always evaluates the arguments of a function before the function itself, thus before evaluating the `car` it will evaluate the `cdr`. The `cdr` yields the list `(c d)`, the `car` of which is just `c`.

1.5.2 Building lists: *cons* and *list*

The function `cons` (stands for "constructs") inserts its first argument at the front of the list that is its second argument, and returns the resultant list.

```
> (cons (car '(cat dog fox)) (cdr '(cow bird sheep)))
(CAT BIRD SHEEP)
```

Again the first thing to do is to evaluate the arguments of this function. `(car '(cat dog fox))` returns `cat` and `(cdr '(cow bird sheep))` returns `(bird sheep)`. Inserting `cat` into `(bird sheep)` yields `(cat bird sheep)`.

You can `cons` a list into a list:

```
> (cons '(cat dog fox) '(cat dog fox))
((CAT DOG FOX) CAT DOG FOX)
```

The `list` function creates a new list by simply wrapping parentheses around the arguments. Note the difference between the two calls below:

```
> (list 'a '(b c))
(A (B C))
> (cons 'a '(b c))
> (A B C)
```

Also:

```
> (cons 'd nil)
(D)
> (list 'd nil)
> (D NIL)
```

Also note the implication of using or omitting the quote:

```
> (cons '(+ 2 3) '(4))
((+ 2 3) 4)
> (cons (+ 2 3) '(4))
> (5 4)
```

Exercise 4

1. Write a function call that will take the list `(c d e)`, and return the first element.
2. Write a function call that will take the list `(1 2 c)` and return it with the number `1` removed (that is, return the tail of the list).
3. Write a function call that inserts the atom `c` into the front of the list `(e f)`.
4. Write a function call that takes the lists `(3 2)` and `(b c)` and produces the list `((3 2)(b c))`.
5. Write a function call that adds 4 to 2 and produces 6. However, get the 4 from the list `(4 3)` before you add it to 2.
6. Write a function call that returns the second element of the list `(horse dog cat)`.

2 An example program

The following Lisp program is available for you to run and get some feel for how Lisp operates. **Don't worry about all the Lisp features at this stage, but just try loading and running the program.**

First enter GCL, as explained in the next section, and then type

>(load "/dcs/gmc/public/lisp/sums.lisp")

Then type:

>(talk)

and follow instructions. Afterwards you could look over the listing, given below, and notice some of the lisp aspects. Try running some of the functions within the program on their own, as suggested in the comments.

```
;;; File:   sums.lisp
;;; Title:  A simple Lisp example for CS262
;;; Author: M.H.Lee
;;; Date:   Sept. 1996
;;; Version: 2.0
;;;
;;; This is an interactive program that presents to the
;;; user some simple arithmetic problems. The answers
;;; are checked and scores are reported.
;;;
;;; Although simple, several features of lisp are illustrated.
;;; You can try (test) functions on their own,
;;; e.g. try (problem 3 4 '+) and (number 4)
;;; Experiment with variations of your own.
;;;
;;; First set up the global variables:

(setf  *operator*  '+
*trials*     0
*hits*       0
*fails*      0
*name*     nil  )

;;; The program is run by typing: (talk)
;;; This is a function with no arguments.
;;;
;;; TALK asks for various data and then loops around the
;;; function PROBLEM.
;;; LET sets up any local variables.

(defun talk ()
  (let ((state) (num-sums) (hardness 1))
         (format t "What is your name ?  ")
         (setq *name* (read))
         (format t " Hello ~a, how are you? " *name*)
         (setq state (read))
         (format t " Lets try some sums ~a ~\%" *name*)
   (loop (format t " How many sums would you like? (0 to exit)  ")
         (setq num-sums (read))
         (cond ((equal num-sums 0) (return (format t "Cheerio ~a " *name*))))
         (format t " How hard? (1 to n) ")
         (setq hardness (read))
         (do ((totsum 0 (+ 1 totsum)))
 ((eq totsum num-sums) (return))
                  (problem (number hardness)(number hardness) *operator*))
         (format t "In ~a trials you had ~a right and ~a wrong ~\%"
               *trials*  *hits*     *fails*)
 (format t "Do you still feel ~a ? ~\%" state)
         (format t "Now lets try some harder ones ~\%")
         (setq *operator* '*)
   )))

;;; The function PROBLEM presents a single arithmetic problem and
;;; checks the answer.
;;; Notice that the value of OP can be used in different ways, either
;;; for printing (i.e. +) (in line 3) or applied as a function (that
;;; adds numbers) (lines 4 & 8).

(defun problem (x y op)
   (incf *trials*)
   (format t " What is ~a ~a ~a, ~a ? " x op y *name*)
   (cond ((equal (read) (apply op (list x y))) (incf *hits*)
        (format t " Very good ~\%") )
         (t (incf *fails*)(format t " No! - type y to try again ")
            (cond ((eq (read) 'y) (problem x y op))
                  (t  (format t " The answer is ~a ~\% " (apply op (list x y) )))
   )     )
))

;;; This function returns a random number, where the
;;; maximum number of digits is given by the argument SIZE.
;;; Notice how recursion is used to generate numbers of *any* size.

(defun number (size)
        (cond ((= 1 size)  (digit))
              ((= 2 size)  (merge-digits (digit) (digit)))
              ((< 2 size)  (merge-digits (number (- size 1)) (digit)))
              (    t       (format t "error in NUMBER -> ~a" size))
              )  )

;;; Generate a random digit, i.e. a number of max size 10

(defun digit ()
        (random 10) )

;;; Merge-digits takes two digits, a, b, and returns 10a + b

(defun merge-digits (a b)
          (+ (* a 10) b) )

;;; end of file
```

3 The Lisp Environment

The Common Lisp package which we use in CS262 is GNU Common Lisp (GCL), version 1.1.

3.1 Starting up

To start the Lisp interpreter, type *gcl* at UNIX prompt.

```
thor% gcl
GCL (GNU Common Lisp) Version(1.1) Fri Nov 25 10:04:21 GMT 1994
Licensed under GNU Public Library License
Contains Enhancements by W. Schelter

>
```

You can see that you are now in GCL by the prompt >. Since Lisp is an interactive language, you can execute any Lisp function, expression, declaration, etc. at the prompt. For example,

```
> (+ 2 3)
5
```

3.2 Exiting Lisp

You can exit from GCL either by typing **(bye)** (don't forget the brackets!) at the prompt, or more simply, by **CTRL-d** (i.e., pressing the *control key* and *d* at the same time).

3.3 Recovering from error

Invalid inputs (such as wrong syntax, use of undefined functions, use of invalid parameters) will often result in raising errors. In such cases, GCL will go into the error mode, indicated by the change in prompt to >>. For example,

```
>(+ 5 a)

Error:  The variable A is unbound.
Fast links are on:  do (use-fast-links nil) for debugging
Error signalled by EVAL.
Broken at +.  Type :H for Help.
>>
```

Although you can carry on as usual when this happens, it is better to recover from this error first. To recover from errors, type **:q** at the prompt.

```
>>:q

Top level.
>
```

Sometimes, you want to interrupt an execution of a program since it may have gone into an infinite loop, or simply doesn't respond (often because of mismatch of brackets). You can do this any time by typing **CTRL-c**, and this takes you into the error mode. From there, you can use the above procedure to return to the normal mode.

3.4 Loading a file

As mentioned previously, Lisp is interactive; you can even define a function at the prompt. But this will be a very tedious and erroneous task if you need to define complicated functions or programs. In such cases, as in Ada, you can write a program outside Lisp using an editor and load it into GCL. You can do this by using the function **load**. For instance, to load a file **test.lisp**:

```
>(load “test.lisp”)
Loading test.lisp
Finished loading test.lisp
T

>
```

When choosing which editor to use, make sure that the editor has a facility to show matching brackets; as you will find out soon, Lisp uses a lot of brackets! For this reason, either *nedit* or *emacs* is recommended.

3.5 On-line documentation

The documentation is rather brief in GCL but you can get information on a built-in function by typing:

```
>(help 'member)
```

And typing:

```
>(help* “mem”)
```

gives details of all functions containing the string “mem” somewhere in their name. If you wish to locate a function for a particular purpose or need to see better documentation, the first place to look is in a good textbook: Winston, or Wilensky.

A standard reference on Common Lisp is “Common Lisp: The Language”, by Guy L Steele, Digital Press.