CS262 Artificial Intelligence Concepts

Worksheet 6

1 Introduction

Although recursion is powerful enough to perform complex list processing procedures, sometimes iteration can make algorithms simpler. In this practical, we first cover how to use local variables in Lisp, and proceed to defining functions involving iteration using `loop`.

2 More about local variables

So far we have seen how to define a global variable using `setq` or `setf` and we have seen local variables used in function parameters. The concept of global variable is the same as in Ada, in the sense that they are assigned values outside the context of any function. Global variables are useful when you have several functions in the program that require the same value. However, they should be used with great care.

Defining local variables

When writing a function which performs multiple actions, it is often necessary to store the results of one action for use in a subsequent action. In Common Lisp, the `let` statement allows us to:

- declare one or more local variables
- (optionally) gives them initial values
- executes a sequence of Lisp actions using these settings

The format of using the `let` statement is:

```
(let ((variable-1 initial-1)
      (variable-2 initial-2)
      ....
      (variable-N initial-N))
     action-1
     action-2
     ...
     action-N)
```

The first argument to `let` is a *local-variable list*. It declares the local variables and their initial values. The subsequent arguments are actions which are taken within the scope of these local variables. As a trivial example, consider a function called `rectangle` which accepts a list containing the length and width of a rectangle and prints the area, perimeter, and diagonal of the rectangle. Since we will need to use the two dimensions in three calculations, we start by extracting them from the arguments and assigning them to local variables.

```
(defun rectangle (dimensions)
    (let ((length (first dimensions))    ;;; local variable length
          (width (second dimensions)))  ;;; local variable width
         (print (list 'area (* length width)))
         (print (list 'perimeter (* 2 (+ length width))))
         (print (list 'diagonal (sqrt (+ (* length length) (* width width)))))))
```

`sqrt` is a built-in function which calculates the square root of a number.

You can use `setq` to assign a new value to a local variable within `let`. However, note that this assignment is only valid within the scope of the `let` statement. Consider the following example.

```
(defun show-vars ()
   (let ((var 'hello))   ;;; local variable VAR initialised as HELLO
        (print var)      ;;; prints the local value of VAR
        (setq var 'hi)   ;;; local VAR re-assigned to HI
        (print var))     ;;; prints the local value of VAR
                         ;;; end of the scope of LET
   (print var)           ;;; prints the global value of VAR
   (setq var 'cheers)    ;;; global VAR re-assigned to CHEERS
   (print var))          ;;; prints the global value of VAR
```

See how this works:

```
>(setq var 'bye)        ;;; sets the global variable VAR to BYE
BYE

>var                    ;;; check the value of VAR
BYE

>(show-vars)

HELLO
HI
BYE
CHEERS
CHEERS

>var                    ;;; value of global VAR changed inside SHOW-VARS
CHEERS
```

Notice the difference between local and global variables.

Exercise

1. Define a function called `longer-list` which accepts two arguments, each of which must be lists. The function returns the list that contains the most elements. If the lists contain the same number of elements, `longer-list` return the word `equal`. Example:

   `(longer-list '(a) '(x y))` returns `(x y)`
   `(longer-list '(1 a) '(b (c d)))` returns `equal`

   Hint: You should create two local variables and initialise one variable to the length of the first list (using `length`) and the other variable to the length of the second list.

3 Iteration

Although recursion is a very powerful technique in programming, in some cases, a simpler algorithm can be obtained by iteration. In Common Lisp, we can use a `loop` statement to repeatedly evaluate a sequence of actions until it is forces to exit from the loop. The template for `loop` is as follows:

```
(loop <action-1>
      <action-2>
      ...
      <action-n>)
```

This evaluates from `action-1` to `action-n` in order and repeats this process until a `return` is called to exit from the loop.

3.1 Numeric iteration

Suppose we want to write a function `add-integers` which adds up all the integers between 1 and some specified number. For example, `(add-integer 5)` would add up 1, 2, 3, 4 and 5, and returns 15, which is the result of this addition. We can use `let` to define local variables, and `loop` to perform iteration.

```
(defun add-integers (num)
    (let ((count 1)                ;;; Initialise local variables
          (total 1))               ;;;   COUNT and TOTAL
         (loop                     ;;; Beginning of LOOP
            (cond ((equal count num) (return total)))  ;;; Exit test
            (setq count (+ 1 count))                  ;;; Update COUNTER
            (setq total (+ total count)))))           ;;; Update TOTAL
```

In this definition, local variable `count` is used as the *control variable*, which is incremented by 1 at each iteration, and `total` is used as the *result variable*, which keeps track of the result of computation at the end of each iteration. Once `count` reaches the limit, provided by the parameter `num`, it exits the loop and return the appropriate value as the result.

Here is another example. This function multiplies two integers provided as arguments.

```
(defun int-multiply (x y)
    (let ((result 0)
          (count 0))
         (loop
            (cond ((equal count y) (return result)))
            (setq count (+ 1 count))
            (setq result (+ result x)))))
```

Exercises

1. Analyse `int-multiply`. What is the control variable and what is the result variable? Suppose we call `(int-multiply 5 0)`, how many times will it loop?
2. Write your own version of `nth` called `my-nth`. This takes two arguments, an integer (non-negative) and a list, and returns the element of the list in the position specified by the integer (0-based). For example:

   `(my-nth 4 '(a b c d e f))` returns `e`.
   `(my-nth 0 '(a b c d e f))` returns `a`

3.2 List iteration

Iteration can be performed not only for numbers, but equally for lists. Suppose we want to write a function, called `double-list`, which takes a list of numbers as its argument, and returns a new list in which each of the numbers has been doubled. For example,

`(double-list '(5 15 10 20))` returns `(10 30 20 40)`

This function can be defined using list iteration:

```
(defun double-list (lis)
    (let ((newlist nil))
```

```
(loop
   (cond ((null lis) (return newlist)))
   (setq newlist (append newlist (list (* 2 (first lis)))))
   (setq lis (rest lis)))))
```

Note how the control variable and result variables are updated. The control variable `lis` (which initially is the parameter) becomes shorter and shorter through iteration, until it becomes empty and satisfies the condition for the exit from the loop. The result variable `newlist` is initialised as `nil`, and accumulates the result of computation at each iteration, until finally returned as the overall result. The computation process for `(double-list '(5 15 10 20))` can be illustrated as follows:

	`newlist`	`lis`
Initial value	`()`	`(5 15 10 20)`
Iteration 1	`(10)`	`(15 10 20)`
Iteration 2	`(10 30)`	`(10 20)`
Iteration 3	`(10 30 20)`	`(20)`
Iteration 4	`(10 30 20 40)`	`()`
Returned value	`(10 30 20 40)`	

Exercises

1. Using iteration, define a function `list-sum`, which takes as an argument a list of numbers, and returns the sum of those numbers. Example:

   `(list-sum '(5 10 -4 27))` returns `38`.
   `(list-sum '())` returns `0`.

2. Define a function called `list-first`, which takes a list of embedded lists, and returns a new list, consisting of the first item of each embedded list. Example:

   `(list-first '((a b c) (train) (45 96)))` returns `(a train 45)`.

3. Define your own version of `member`, called `my-member`, which takes two arguments. The first argument can be an atom or a list and second argument is a list. The function checks whether the first argument is a top-level element of the second argument. If so, it should return the tail of the second argument beginning with the first occurrence of the first argument. If not, returns `nil`. Example:

   `(my-member 'jack '(mary john jack jill tom))` returns `(jack jill tom)`.
   `(my-member 'a '(x y (a) (a b)))` returns `nil`.

4. Write a function called `list-intersect`, which tales two lists as arguments, and returns a list that is an intersection of the two lists. In other words, it returns a list of all the elements that appear in both argument lists. Be careful not to duplicate elements in the intersection. Example:

   `(list-intersect '(a b a c b) '(a a b c d))` returns `(a b c)`.

4 DO loops

The special form `do` incorporates features of `let` and `loop`, and can be used for various forms of iteration. Below is the template for `do`-loop.

```
(do ((<var-1> <init-var-1> <update-var-1>)
     (<var-2> <init-var-2> <update-var-2>)
     ...
     (<var-n> <init-var-n> <update-var-n>))
    (<exit-test> <return-value>)
    <action-1>
    <action-2>
    ...
    <action-m>)
```

This construct can be interpreted as follows:

1. `<var-1>`,`<var-2>`,...,`<var-3>` become local variables for the duration of the `do`-loop.
2. These variables are given initial values `<init-var-1>`, `<init-var-2>`,...,`<init-var-n>`.
3. The `<exit-test>` is evaluated and if it returns a non-nil value, then the `do`-loop finishes, returning as its result the evaluation of the `<return-value>`.
4. If `<exit-test>` evaluates to nil, the *body* of the `do`-loop, i.e., `<action-1>`, `<action-2>`,..., `<action-m>`, is evaluated in order.
5. Then the values of the local variables are all updated, simultaneously, with the values produced by evaluating the corresponding `<update-var-n>` expressions.
6. Continues loop until exit.

The equivalent construct using `let` and `loop` looks like this:

```
(let ((<var-1> <init-var-1>)
      (<var-2> <init-var-2>)
      ...
      (<var-n> <init-var-n>))
     (cond (<exit-test> (return <return-value>)))
     <action-1>
     <action-2>
     ...
     <action-m>
     (setq <var-1> <update-var-1>)
     (setq <var-2> <update-var-2>)
     ...
     (setq <var-n> <update-var-n>))
```

As you can see, using `do`-loops makes the program more concise, without the need for `return` and `cond` statements.

Recall the function `int-multiply` we defined earlier.

```
(defun int-multiply (x y)
    (let ((result 0)
          (count 0))
         (loop
            (cond ((equal count y) (return result)))
            (setq count (+ 1 count))
            (setq result (+ result x)))))
```

This can be rewritten using `do`-loop as follows:

```
(defun int-multiply (x y)
    (do ((result 0 (+ result x))
         (count 0 (+ count 1)))
        ((equal count y) result)))
```

Notice that this function does not have anything which corresponds to `<action>` in the template. Since most iterations simply update the local variables, it is often the case that `do`-loop contain no body.

Exercises

1. Rewrite `list-sum` and `my-member` in the previous Exercises using `do`-loop.

4.1 Conditional updates

In the use of `do`-loops we have seen so far, the variables are updated in the same way on each pass through the loop. Suppose we want to check something about the data and make such updates dependent on that property of the data. For instance, if we want to write a function called `save-large`, which go through a list of numbers and save all the numbers greater than 100:

```
(defun save-large (lis)
    (do ((oldlist lis (rest oldlist))
         (result nil (cond ((> (first oldlist) 100) (cons (first oldlist) result))
                           (t result))))
    ((null oldlist) (reverse result))))
```

Notice that in this function, we used the `cond`-statement to update the value for `result`. If the next element is greater than 100, then add that to `result`; if not, then reset `result` to its previous value.

Exercises

1. Define a function called `save-atoms`, which goes through a list and returns a list of all the elements of the argument list that are atoms. Make sure that the atoms in the result appears in the same order as they appear is the original list. Example:

    `(save-atoms '(x y (a b) z (c)))` returns `(x y z)`.

2. Define a function `sortnums` which takes one argument, a list containing numbers. The function returns a new list with two embedded lists—the first contains the negative numbers and the second contains the positive numbers and zero(s). again, the order of the numbers should be the same as that appears in the original list. Example:

    `(sortnums '(3 -3 0 -7 1))` returns `((-3 -7) (3 0 1))`.

3. Write a function called `rectangle` which takes two positive integers, say `m` and `n`, as arguments, and produces a rectangle consisting of the letter `o`. Example:

    ```
    >(rectangle 3 4)
    oooo
    oooo
    oooo
    oooo
    NIL
    ```