1 Delayed function calls

1. Assume you have a list of integers that you would want to turn into a list of their, say, factorials in a single stroke. And further assume that you would like to have a more general tool, which does the same trick not only with factorial but any unary function you provide to it, e.g. cube, square root, etc. What you need is a function that takes a function and a list as arguments, apply the function one by one to the elements of the list while storing the results in another list. Let us call this function MAPP – note the double 'P' not to clash with the built-in function MAP.

1.1. This is a task that would be straightforwardly implemented with recursion. Assume MAPP is given a function f and some list. If the list is empty, than its MAPP should be empty; if the list is non-empty the MAPP of it is simply the value obtained by applying f to the CAR of the list consed with the value obtained by calling MAPP with f and the rest of the list.

1.2. Here is a definition that attempts to achieve this:

```
(defun mapp (func lst) ;; WRONG!
  (if (endp lst)
    nil
    (cons (func (car lst)) (mapp func (cdr lst)))
    )
  )
```

1.3. The problem with the above definition is that it expects LISP to bind the function provided by the parameter func to the func that occurs in the function definition. LISP is designed *not* to do this;[1] when you call this function, say with (mapp factorial '(1 2 3 4)), or (mapp 'factorial '(1 2 3 4)), the func in the definition will not get replaced by the parameter you provided for the argument named func. Such replacements are done only with non-initial elements of lists – even not for all such elements, see 1.7. below.

1.4. What we want is achievable with the functions FUNCALL or APPLY.

1.5. FUNCALL wants its first argument to be something that would *evaluate* to:

i. a symbol with a function binding; or

ii. a function.

1.6. The rest of its arguments are treated as arguments of the function provided via the first argument; the function is applied to its arguments and the value is returned.

1.6.1. Assuming the definition of FACTORIAL is loaded,

```
(funcall factorial 8)
```

would lead to an error. The reason is that before getting fed into FUNCALL the argument FACTORIAL gets evaluated. Remember the rule of evaluation, which says that if a symbol is encountered at a non-initial position in a list, it gets evaluated to its value-binding. In this case LISP cannot find anything in the value-binding. The above specification of FUNCALL says that the first argument should be something that would *evaluate* to a symbol. Therefore the correct form is:

```
(funcall 'factorial 8)
```

This way FUNCALL gets a symbol – that's what 'FACTORIAL gets evaluated to – with a function-binding. This function is retrieved and applied to the remaining arguments – we have only one in the present case.

1.6.2. You can use the built-in functions as follows:

```
(funcall '+ 8 7 29)
(funcall 'member 'a '(z c a t))
```

1.6.3. The specification of FUNCALL in 1.5. has a second clause, which says that one can also provide an argument that evaluates to a function. Given a symbol, you can access its function-binding in two ways:

```
(function factorial)
(symbol-function 'factorial)
```

note the quote in the second form. The first form is the frequent one and like the QUOTE function, it has an abbreviated form:

```
#'factorial
```

Therefore you can directly send a function as an argument to FUNCALL as,

```
* (funcall #'+ 8 7 29)
* (funcall #'factorial 9)
```

1.7. Now you might think that with the below code we could get what we want, for instance with (mapp factorial '(1 2 3 4)) to get (1 2 6 24).

[1] A dialect of LISP, called Scheme, does this.

```
(defun mapp (func lst) ; STILL WRONG!
  (if (endp lst)
    nil
    (cons (funcall (function func) (car lst)) (mapp func (cdr lst)))
    )
  )
```

but again FUNCTION – QUOTE is no different – prevents the argument FUNC from getting bound to the parameter provided to the MAPP.
The correct code is:

```
(defun mapp (func lst)
  (if (endp lst)
    nil
    (cons (funcall func (car lst)) (mapp func (cdr lst)))
    )
  )
```

and you can call the function in two ways, both are fine:

```
(mapp 'factorial '(1 2 3 4))
(mapp #'factorial '(1 2 3 4))
```

2 Applicative programming

1. Lists are the basic data structures in LISP and in many other programming languages. Once you have your data in the form of a list, then you can do various transformations and/or checks on your data.

2. In most applications, data is read from a file, entered by user, or provided by a stream over a network. We will come to these topics, but for now, we will assume that data is given stored in a **global** variable in our program. One way to declare and assign a global variable is DEFPARAMETER. For instance, let us assume we have a set of grades:

```
(defparameter *grades*
    '(86 98 79 45 0 75 96 83 91 90 0 70 85 82 91 47 0 70))
```

3. The asterisk characters * around the word "grades" have no special meaning for LISP. It is a convention among LISP programmers to name global variables as such.

4. Having all the grades stored in a list, let us write a very simple function that would compute the sum of them. What about this one?

```
(defun total (lst) ; WRONG
  (+ lst))
```

5. Evaluating (total *grades*) would give an error. The reason this is unsuccessful is that the + operator works on numbers, but we provide a list as an argument. Using FUNCALL would not help as well. The construct to use in such situations is APPLY. It is like FUNCALL with the only difference that the arguments are provided in a list.

```
(defun total (lst)
  (apply #'+ lst))
```

6. Now, evaluating (total *grades*) should give you the number 1188.

7. Sometimes, one needs to transform a list into another list with an equal length and with a specific correspondence between the elements of the two lists. This is a mapping. LISP provides the built-in MAPCAR for this purpose. For instance assume you would – for some strange reason – take the square root of the grades, and collect them in a list. LISP has the built-in SQRT that can be applied to numbers. To apply this wholesale on a list, say *grades*, just do:

```
(mapcar #'sqrt *grades*)
```

8. There is no limit to the complexity of the functions you can use with MAPCAR. Let us first define a function that gives the letter grade corresponding to the numerical value. You can easily do this by COND. Give it the name LETTER. Doing (mapcar #'letter *grades*) should give you:

```
(BA AA CB FF FF CB AA BB AA AA FF CC BA BB AA FF FF CC)
```

9. Or, you can define a function APPEND-LETTER, which pairs each grade with the letter grade, returning everything in a list. What you get in the end should look like:

```
((86 BA) (98 AA) (79 CB) (45 FF) (0 FF) (75 CB)
 (96 AA) (83 BB) (91 AA) (90 AA) (0 FF) (70 CC)
 (85 BA) (82 BB) (91 AA) (47 FF) (0 FF) (70 CC))
```

10. Remember that we had to use APPLY to get the sum of a list of numbers. Another way to do this is to use REDUCE. This is somewhat similar to MAPCAR. It takes a function with two arguments and a list; then it reduces the entire list to a single value, by applying the function first to the first two elements, then to the result of this application and the third element, then the result of this application and the fourth element, and so on until it reaches the end of the list. For instance, the following function calls will give you the sum and the mean of the grades, respectively.

```
(reduce #'+ *grades*)
(/ (reduce #'+ *grades*) (length *grades*))
```

11. Usually we would not want to include the 0 grades in computing the mean, especially if we know that these are coming from people who did not participate in the exam or the course. A useful pair of list filtering functions are REMOVE-IF and REMOVE-IF-NOT. To filter out zero grades, just do:

```
(remove-if #'zerop *grades*)
```

12. If we want to have a function that computes the mean of the grades, with first filtering the zero values, we might have the following:

```
(defun class-mean (grades-list)
  (float (/
          (reduce #'+ (remove-if #'zerop grades-list))
          (length (remove-if #'zerop grades-list)))))
```

13. The code is inefficient as we compute the very same thing twice. There is a very useful construct to store values that are to be used more than once. The construct is LET and here is its syntax:

```
(let ((<variable_1> <value_1>)
      (<variable_2> <value_2>)
          .
          .
          .
      (<variable_n> <value_n>))
      <body>
     )
```

where <body> is just an ordinary function body.

14. Here is the mean function with LET binding:

```
(defun class-mean (grades-list)
  (let ((real-grades (remove-if #'zerop grades-list)))
    (float (/ (reduce #'+ real-grades) (length real-grades)))))
```

15. Example: A Collatz sequence is obtained by starting with an integer n; if n is odd, add $3n+1$ to the sequence, if n is even add $n/2$ to the sequence. If you obtain 1 stop, otherwise go on as before with the lastly added integer. The Collatz Conjecture states that no matter which integer you start the sequence, you are guaranteed to reach 1 and stop after a finite number of iterations. Let us write a function that computes the Collatz sequence for a given integer.

```
(defun collatz-generate (n)
  (if (= n 1)
    '(1)
    (let ((new-value (if (evenp n)
                         (/ n 2)
                         (+ (* n 3) 1))))
      (cons n (collatz-generate new-value)))))
```

16. Let us define Collatz (sequence) length of an integer to be the number of steps needed to reach 1 from that integer. This is one less than the length of Collatz sequence. So,

```
(defun collatz-length (n)
  (- (length (collatz-generate n)) 1))
```

17. Knowing how numbers are correlated with their Collatz length would be interesting; does the lenght grow as the number grows, for instance? Or is there a fluctuating pattern? To do this let us first find a way of generating a sequence of integers within a given range; after this it would be easy to MAPCAR it to what we want to investigate. We already wrote a function for generating ranges; this time we will write a more sophisticated one. We will use keyword arguments in doing that. First the code, then we will look at it in detail.

```
(defun ranger (&key (start 0) (end 9) (step 1) (acc nil))
  (cond ((>= start end) (cons start acc))
        (t (ranger :acc (cons end acc)
                   :start start
                   :end (- end step)
                   :step step))))
```

this range function generates a range including its start and end points. Keyword arguments allow you to refer to the parameters by names – don't forget the ':' before the keywords – so that you do not have to remember the order of the parameters, as you should for optional parameters.[2]

18. Now we have enough machinery to list the Collatz lengths of the first one million integers:

```
(mapcar #'collatz-length (ranger :start 1 :end 1000000))
```

19. Check the maximum Collatz length in this range:

```
(apply #'max
    (mapcar #'collatz-length (ranger :start 1 :end 1000000)))
```

20. MAX breaks down for 1,000,000 range – it should respond 350 for 100,000. We define our own maximum function, which operates on lists:

```
(defun maxx (lst &optional (max nil))
  (cond ((endp lst) max)
        ((null max) (maxx (cdr lst) (car lst)))
        (t (maxx (cdr lst) (if (> (car lst) max)
                               (car lst)
                               max)))))
```

21. Now we can check the maximum Collatz length in a range of one million:

```
(maxx (mapcar #'collatz-length (ranger :start 1 :end 1000000)))
```

22. We can observe how the maximum Collatz length changes with the size of the range. Here is how to do it with LAMBDA:

```
(mapcar
    #'(lambda (range)
        (maxx (mapcar #'collatz-length
                  (ranger :start 1 :end range))))
    '(10 100 1000 10000 100000 1000000))
```

giving (19 118 178 261 350 524).

23. It may be hard to count 0's in specifying various sizes of ranges – it is easy to err. Let us handle that task with mapcar as well.

```
(mapcar #'(lambda (x) (expt 10 x)) '(1 2 3 4 5 6))
```

Now we can have

```
(mapcar #'(lambda (range)
            (maxx (mapcar #'collatz-length
                          (ranger :start 1 :end range))))
        (mapcar #'(lambda (x)
                    (expt 10 x)) '(1 2 3 4 5 6)))
```

24. We have gained some knowledge on Collatz sequences, but we still do not know about individual numbers. We can pair numbers with their Collatz lengths:

```
(mapcar #'(lambda (x) (list x (collatz-length x)))
    (ranger :start 1 :end 1000000))
```

25. Let us find the number with the maximum Collatz length. MAXX on its own is not helpful here; it justs finds the maximum number in a list of numbers. What we need is a way to find the pair(s) with the maximum second second element. One way is to write a function similar to MAXX. Let us do this. But let us do this in a way that will have a general use. We will parametrize MAXX in such a way that, the caller/user of the function will decide where MAXX should look for items to compare. In the above version it looks at top most elements, by checking the CAR of the list in every recursion. Here is our new, more "customizable" MAXX.

[2] The function has a flaw; it may not work as expected for some ranges with step different than 1, can you see why? Any ideas to fix it?

```
(defun maxx (lst &key
                 (max nil)
                 (hook #'(lambda (x) x)))
  (cond ((endp lst) max)
        ((null max) (maxx (cdr lst) :max (car lst) :hook hook))
        (t (maxx
            (cdr lst)
            :max (if (> (funcall hook (car lst))
                        (funcall hook max))
                     (car lst)
                     max)
            :hook hook))))
```

26. With our new MAXX in our hands we can now find the number with the largest Collatz length in a given range, say 1000.

```
(maxx (mapcar #'(lambda (x)
                  (list x (collatz-length x)))
                (ranger :start 1 :end 1000)) :hook #'cadr)
```

27. Finally, we would like to know the numbers in a given range that have the same Collatz lenght. The solution is left as an exercise.

3 User interaction

4 Search (optional)

4.1 Directed graphs

1. A directed graph is a set of ordered pairs of nodes. Pairs are understood as involving a link from the first component to the second component. Simple graphs are best visualized as graph diagrams:

2. The directed graph in Figure 1 can be represented as the LISP list:

```
((S A) (S B) (S C) (A F) (C B) (B A))
```

3. We can economize on representation by representing each node as a single list consisting of itself – as the car of the list – and collecting all the nodes that are directed from it in the cdr of the list. For instance,

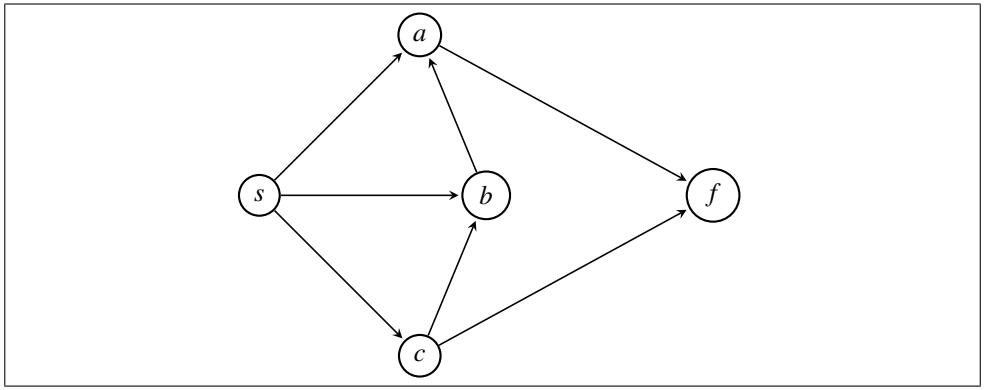

Figure 1: A directed graph.

```
((S A B C) (A F) (C B) (B A))
```

4.2 Searching for paths in a graph

1. We write a program to find paths given a starting point and destination.

```
(defun extend-path (path &optional (graph *graph1*))
  (let ((neighbors (cdr (assoc (car path) graph))))
    (mapcar #'(lambda (node)
                (cons node path))
            (remove-if #'(lambda (node) (member node path))
                       neighbors))))

(defun search-path (start end graph
                           &key
                           (mode 'bf)
                           (agenda (list (list start)))
                           result
                           (current (car agenda)))
  (cond ((endp agenda) result)
        ((equal end (car current))
         (search-path start end graph
                      :agenda (cdr agenda)
                      :result (cons (reverse current) result)
                      :mode mode))
        (t
          (search-path start end graph
                       :agenda (if (equal mode 'df)
                                   (append
                                    (extend-path current graph)
                                    (cdr agenda))
                                   (append
                                    (cdr agenda)
                                    (extend-path current graph)))
                       :result result
                       :mode mode))))
```

References

Churchland, P. S. and Sejnowski, T. J. (1992). *The Computational Brain*. MIT Press.

Crane, T. (2003). *The Mechanical Mind*. Routledge, New York.

Pohl, I. and Shaw, A. (1981). *The Nature of Computation: An Introduction to Computer Science*. Computer Science Press, Rockville, Maryland.

Pylyshyn, Z. (1984). *Computation and Cognition: Toward a foundation for Cognitive Science*. MIT Press, Cambridge, MA.

Tanenbaum, A. S. (1999). *Structured Computer Organization*. Prentice Hall, NJ, 4th edition.

Touretzky, D. S. (1990). *COMMON LISP: A Gentle Introduction to Functional Programming*. Benjamin/Cummings Publisihing Co., CA.