CS262 Artificial Intelligence Concepts

Worksheet 3

1 Introduction

In constructing a definition of a function, it is possible to rely on the definition of that same function for other parameter values. One way to do this to define the function for some *base case*, which is usually a simple value such as 0 or 1, and to define how a more complex case (e.g. an arbitrary integer) is defined in terms of simpler values. A function with such a design is called a *recursive function* and is useful in solving problems that require some operations to be carried out repeatedly.

In this practical we learn how to design recursive functions for both numeric recursion and list recursion.

2 Numeric recursion

Consider a function which takes an integer (including 0) and returns the sum of all the integers between 0 and that number. If we call that function `summ`, `(summ 7)` should return `28`, since

$$7 + 6 + 5 + 4 + 3 + 2 + 1 + 0 = 28$$

A recursive function would have the following recipe to solve this problem:

1. If the argument is 0, then it must return 0 since the sum of all the integers between 0 and 0 is obviously 0.
2. Otherwise, if the argument is n, 'the sum of the integers between 0 and n' equals 'n plus the sum of the integers between $n - 1$'.

This can be translated into the Lisp function definition below:

```
(defun summ (n)
    (cond ((zerop n) 0)
          (t (+ n (summ (- n 1))))))
```

Notice that there are two cases in this cond-statement. The first case is called a *terminating case*, or *base-case*, since it returns a value without a recursive call. The second case is called a *recursive case*. Here's how `(summ 7)` is being computed internally:

```
(summ 7)
(+ 7 (summ 6))
(+ 7 (+ 6 (summ 5)))
(+ 7 (+ 6 (+ 5 (summ 4))))
(+ 7 (+ 6 (+ 5 (+ 4 (summ 3)))))
(+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (summ 2))))))
(+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (summ 1)))))))
(+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (summ 0))))))))
(+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))))
(+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1))))))
(+ 7 (+ 6 (+ 5 (+ 4 (+ 3 3)))))
(+ 7 (+ 6 (+ 5 (+ 4 6))))
(+ 7 (+ 6 (+ 5 10)))
(+ 7 (+ 6 15))
(+ 7 21)
28
```

You can see the effect of recursion if you use the *tracing* facility in Common Lisp. We can trace a function call by typing `(trace` <*function-name*>`)` at the Lisp prompt. Then, each time the function is called, a list of arguments to the function is printed, and each time the function exits, the value that function returns is printed. For example:

```
>(trace summ)
(SUMM)

>(summ 7)
  1> (SUMM 7)
    2> (SUMM 6)
      3> (SUMM 5)
        4> (SUMM 4)
          5> (SUMM 3)
            6> (SUMM 2)
              7> (SUMM 1)
                8> (SUMM 0)
                <8 (SUMM 0)
              <7 (SUMM 1)
            <6 (SUMM 3)
          <5 (SUMM 6)
        <4 (SUMM 10)
      <3 (SUMM 15)
    <2 (SUMM 21)
  <1 (SUMM 28)
28

>(untrace summ)
(SUMM)
```

From this printout, we can see the depth of the recursion. `trace` is useful for debugging, but do not rely too much on it: the output can be confusing. Once you are satisfied with the tracing, you can switch it off by an `untrace` call as shown above.

3 Designing a recursive function

In writing a recursive function, we require the terminating case(s) and the recursive case(s).

1. **Planning the terminating case(s).** A terminating case prevents the function from calling itself recursively forever. The terminating case returns a result, which provides the basis for computing the results of the recursive calls.
   - When a terminating case is evaluated, it should return the correct answer, given the current argument(s) to the function.
2. **Planning the recursive case(s).** Each time the function calls itself, the argument should get closer to the terminating case. For example, in `summ`, the recursive calls were made with smaller and smaller integers. This guarantees that at some point, `summ` will be called with `0`. So always make sure that the recursive call is leading to the terminating case.
   - We need to find out the *recursive relation*, i.e., the relation between the result of the recursive call and the answer for the current call. In `summ`, the recursive relation was `(summ n)` = `n` + `(summ (- n 1))`.
   - Finding the recursive relation is not always easy. In such a case, figure out some of some sample function calls, and their corresponding recursive calls. Then characterise what the relation is between the pairs of function calls.

Here is another example of a recursive function, `multiply`, which is equivalent to the built in function `*`. That is, `multiply` takes two arguments (numbers) and multiply them.

Step 1 — Terminating case. If one of the arguments, say the first argument, is 0 then it just returns 0.

Step 2 — Recursive case. Recursive relation between `(multiply x y)` and `(multiply (- x 1) y)`.

- Recursive examples:

	`(multiply x y)`	`(multiply (- x 1) y)`
(i)	`(multiply 3 4)`= 12	`(multiply 2 4)`= 8
(ii)	`(multiply 1 4)`= 4	`(multiply 0 4)`= 0

- Characterise recursive relation:
`(multiply x y)` can be obtained from `(multiply (- x 1) y)` by adding `y`.

Function definition.

```
(defun multiply (x y)
    (cond ((zerop x) 0)
          (t (+ y (multiply (- x 1) y)))))
```

Exercises

1. Define the function `factorial`, which computes the factorial of its argument. It takes one argument `n`, which must be an integer that is greater or equal to 0. the function returns the factorial of `n`, which is the result of multiplying `n` × (`n` - 1) × (`n` - 2) × ... × 1. For example,
`(factorial 3)` returns `6`, since $3 \times 2 \times 1 = 6$.
By definition, the factorial of 0 is 1.
Before doing this problem, work out the missing parts of the recursive table below for `factorial`:

   (a) Terminating case.
   $n = 0$ `(factorial 0)`=__.

   (b) Recursive case.
   Relation between `(factorial n)` and `(factorial (- n 1))`.

   - Recursive examples:

	`(factorial n)`	`(factorial (- n 1))`
(i)	`(factorial 5)`=__	`(factorial 4)`=__
(ii)	`(factorial 1)`=__	`(factorial 0)`=__

   - Characterising recursive relation:
   `(factorial n)` can be obtained from `(factorial (- n 1)` by _____.

2. Define the function `power`. It takes two arguments, integers `m` and `n`, and returns the value of `m` raised to the power of `n`. For example:
`(power 2 3)` returns `8`, since $2^3 = 2 \times 2 \times 2 = 8$.
By definition, any number to the power of 0 is 1.
Again, first work out the recursive table for `power`:

   (a) Terminating case.
   $n = 0$ `(power 4 0)`=__.

   (b) Recursive case.
   Relation between `(power m n)` and `(power m (- n 1))`.

   - Recursive examples:

	`(power m n)`	`(power m (- n 1))`
(i)	`(power 5 3)`=__	`(power 5 2)`=__
(ii)	`(power 3 1)`=__	`(power 3 0)`=__

- Characterising recursive relation:
  `(power m n)` can be obtained from `(power m (- n 1))` by _____.

4 List recursion

We have so far seen the recursion on numbers. Similarly, recursive processing can be performed on lists. The structure of list recursion is very much similar to that of numeric recursion. For example, the if we want to write a function `list-sum` which takes a list of numbers as its argument and returns the sum of all the numbers in the list, e.g., `(list-sum '(4 2 6 3))` returns `15` since $4+2+6+3=15$, we can use list recursion as follows:

1. The sum of all the numbers in the empty list is 0
2. For any non-empty list of numbers,
   the sum of all the numbers in the list
   equals
   the sum of the first number in the list
   plus
   the sum of all the numbers in the `rest` of the list.

From the observation above, we can define `list-sum` as follows:

```
(defun list-sum (lis)
    (cond ((null lis) 0)
          (t (+ (first lis) (list-sum (rest lis))))))
```

Here's how `(list-sum '(4 2 6 3))` is computed internally:

```
(list-sum '(4 2 6 3))
(+ 4 (list-sum '(2 6 3)))
(+ 4 (+ 2 (list-sum '(6 3))))
(+ 4 (+ 2 (+ 6 (list-sum '(3)))))
(+ 4 (+ 2 (+ 6 (+ 3 (list-sum nil)))))
(+ 4 (+ 2 (+ 6 (+ 3 0))))
(+ 4 (+ 2 (+ 6 3)))
(+ 4 (+ 2 9))
(+ 4 11)
15
```

As you can see, each time `list-sum` is called with a non-empty list, the function calls itself recursively on the `rest` of its argument before returning a value. Finally when `(list-sum nil)` (equivalent to `(list-sum '())`) is called, it returns a value, 0, without calling `list-sum` further. This allows the completion of each recursive call and it is propagated all the way back up.

Again, the design of list recursion is analogous to that of numeric recursion. We can use the same template to approach the solution.

Exercise

1. Define a function `list-length`. Given a list, it returns the number of elements in that list. You should count only the top-level elements, not elements within the nested lists. Example:

   `(list-length '(a (b c) d))` returns `3`.
   `(list-length nil)` returns `0`.

As in the numeric recursion, work out the recursion table below and write the function definition based on it.

(a) Terminating case.
`(list-length nil)`=__.

(b) Recursive case.
Relation between `(list-length lis)` and `(list-length (rest lis))`.

- Recursive examples:

	`(list-length lis)`	`(list-length (rest lis))`
(i)	`(list-length '(a (b c) d))`=__	`(list-length '((b c) d))`=__
(ii)	`(list-length '(j))`=__	`(list-length nil)`=__

- Characterising recursive relation:
`(list-length lis` can be obtained from `(list-length (rest lis))` by _____.

5 Multiple terminations and recursions

Up to now, we have only seen recursive functions which had just one terminating case and one recursive case. However, some recursive functions require more than one terminating or recursive case. The following function `greaternum` takes two arguments, a list of numbers and a number, and returns the first item in the list that is greater than the second argument. If all the numbers in the list are smaller than the second argument, it returns the second argument, since it is the greatest number.

```
(defun greaternum (lis num)
    (cond ((null lis) num)
          ((> (first lis) num) (first lis))
          (t (greaternum (rest lis) num))))
```

Here are a couple of examples:

```
(greaternum '(4 5 2 7 8) 6) returns 7.
(greaternum '(23 45 12 4) 50) returns 50.
```

Notice that this function contains two terminating cases.

1. `((null lis) num)`
2. `((> (first lis) num) (first lis))`

These correspond to the following observations.

1. If the empty list is given as the first argument (i.e., list of numbers), simply return the second argument, since there is nothing to compare.
2. If the first item of the list is greater than the second argument, return the first item.

When the input falls into neither of these cases, then the recursive case is called, discarding the first item in the list (since it already failed to be greater than the second argument).

Another example. The function `ismemberof` below takes two arguments, an item and a list, and tests if the item occurs anywhere in the list. Example:

`(ismemberof 'c '(a b c d))` returns `t`.
`(ismemberof 'john '(mary chris james))` returns `nil`.

Here is the function definition.

```
(defun ismemberof (item lis)
    (cond ((null lis) nil)
          ((equal item (first lis)) t)
          (t (ismemberof item (rest lis)))))
```

Each case in this function definition can be read as follows:

1. (Terminating case 1) If the list is empty, then there is no way you can find the item in the list. Return `nil`.
2. (Terminating case 2) If the first item of the list is the same as the item you are looking for, then you've found it. Return `t`.
3. (Recursive case) Otherwise, repeat the search but without the first item of the list (since it has already failed to be the item being sought for).

Exercises

1. Define a function called `negatives`, which takes a list of numbers and returns a new list that contains only the negative numbers. Example:

   `(negatives '(9 -2 -5 0 6))` returns `(-2 -5)`.

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

   `(first-items '((mary jane ann) (thomas john) (4 3 1 5)))` returns
   `(mary thomas 4)`.

3. Define a function called `union` which takes two lists and returns the union of the two. The union of two lists is all the elements that are in either of the two lists, but with no duplicates. You may assume that the lists given as arguments will not contain any duplicates. Example:

   `(union '(a b c d) '(c d e))` returns `(a b c d e)`.

4. Define a function called `flatten`, which takes a list which may or may not contain embedded lists, and returns the list which contains all atoms in the list and embedded lists. Example:

   `(flatten '(a (b c) (d (e f) (g)) (h)))` returns `(a b c d e f g h)`.