CS262 Artificial Intelligence Concepts

Worksheet 5

1 Introduction

In the last practical, we introduced numeric recursion and list recursion. In this practical, we describe more sophisticated recursion to perform more complex tasks. Also, we introduce some useful Common Lisp data structures, *association lists*, and some more built-in functions.

2 Advanced list recursion

The list recursion function we implemented in Practical 5 use *rest*-recursion (apart from `flatten` function). These functions involve recursive calls to the *rest* of the list, and combine that result with the result for the *first* of the list.

However, not all problems that require us to process elements in a list can be solved using `rest`-recursion only. Consider a function called `powerset` that takes one argument which is a list containing no duplicates. The function returns a list of all possible lists that can be generated from the elements of the argument, ignoring order, and without any repeating elements. It should include the empty list and the original list. For example, `(powerset '(a b c))` returns
`((a b c) (a b) (a c) (a) (b c) (b) (c) nil)`.

Exercise

1. Define `powerset` by considering the following:

   Recursive case: The example shows what `powerset` returns for the list `(a b c)`. What should `powerset` return if we apply it to the *rest* of the list? In other words, what should `(powerset '(b c))` return?

   General form of the recursive case: What is the relation between the result of `(powerset '(a b c))` and that of `(powerset '(b c))`? What do we have to do to the latter result, in order to generate the former?

   Terminating case: What should `powerset` return when called with the empty list as an argument?

   Function definition: From the analysis above, define `powerset`.

2.1 "first-rest" recursion

Suppose we require a function `isin` that reports whether an expression appears at any level of embedding in a list. It will take two arguments, an expression and a list, and if the first argument appears at any level of the second argument, then `isin` returns `T`, and `NIL` otherwise. For example:

```
> (isin 'b '(a ((b) c) d))
T

> (isin 'e '(a ((b) c) d))
NIL
```

The function `member` is not of any use here since it determines only is an element occurs at the *top level* of a list. For instance, `(member 'b '(a b c d))` would return `T`, but
`(member 'b '(a ((b) c) d))` would return `NIL`.

The function `isin` cannot be written using simple list recursion since we need to get inside of the embedded lists and check their structure. That is, we need to call the function recursively on the *first* of the list as well on the *rest*. This technique is called *first-rest* recursion (or, *car-cdr* recursion). With this technique, `isin` can be defined as follows:

```
(defun isin (item lis)
   (cond ((null lis) nil)
         ((equal item (first lis)) t)
         ((atom (first lis)) (isin item (rest lis)))
         (t (or (isin item (first lis))
                (isin item (rest lis))))))
```

We can analyse the function above as follows:

1. The first two cases are terminating cases:
   (a) The function checks if `lis` is empty. If it is, it is obvious that `item` does not occur in `lis`, so return `NIL`.
   (b) If `lis` is not empty, check is the first element of `lis` is equal to `item`. If it is, then we can simply return `T`.
2. The third and fourth cases are recursive cases:
   (a) By the time this case is reached, we know that the first element of `lis` is not `item`, but we don't know if that first element is itself a list which contains `item`. Therefore, we check if it is an `atom`; if it is, then we simply need to check whether `item` is in the `rest` of the list.
   (b) The fourth case is the core of this *first-rest* recursion. We know here that first element of `lis` is a list, and `item` may be either somewhere is the first element of `lis` **or** in the `rest` of `item`. So in this case, we call `isin` recursively on **both** the `first` and the `rest` of `lis`. If either of these recursive calls returns `T`, then `T` will be returned by the function `or`.

As another example, consider the following function definition:

```
(defun numbers (lis)
    (cond ((null lis) nil)
          ((numberp (first lis)) (cons (first lis) (numbers (rest lis))))
          ((atom (first lis)) (numbers (rest lis)))
          (t (append (numbers (first lis)) (numbers (rest lis))))))
```

This function takes one argument which is a list, and returns a list of all the numbers in the argument. For example, `(numbers '((a 1 (2)) b 3))` returns `(1 2 3)`. Make sure you understand what is happening in each of the four cases. To confirm your analysis, the fourth condition can be stated as follows:

> *In the last case, the first element of* `lis` *is a list. We can assume that* `(numbers (first lis))` *will return a list of all numbers in the* `first` *of* `lis`*, and that* `(numbers (rest lis))` *will return a list of all numbers in the* `rest` *of* `lis`*. Since we want to return a single list with all the numbers, we* `append` *the results of these two function calls.*

In general, the following guidelines can be used to plan *first-rest* recursive functions.

1. Plan the terminating case(s): this can be either (or both)
   - If the list is empty, then return a value without a recursive call.
   - If the `first` of the list is an atom, and this happens to be something you are looking for (thus requiring no further investigation), then return the value without a recursive call.
2. Plan the *rest* recursive case(s): this is the case when the `first` of the list is an atom, and you need the result of the function called on the `rest` of the list. There are two general cases:
   - In some cases, you simply can ignore the `first` of the list as completely irrelevant. In such cases, simply call the function recursively for the `rest` of the list.
   - When the results of the `rest` recursive call have to be combined with a value that is derived from the `first` of the list, then provide an appropriate function to combine them (e.g., `list`, `append`, `cons`).
3. Plan the *first-rest* recursive case: this usually is the final case ("otherwise" or `T` case). Usually, the first item of the list is not an atom, and you need to recursively apply the function to the `first` of the list as well as to the `rest` of the list, and combine these results in an appropriate way.

Exercises

1. Define a function `count-atoms` which takes one argument, which must be a list, and returns the number of atoms contained at any level of embedding in the list. Example:

   `(count-atoms '((a) (b (c)) (d)))` returns `4`.
   `(count-atoms nil)` returns `0`.

2. Define a function called `delete-in`, which takes two arguments, an atom and a list, and returns a list which all the occurrence of the atom in the list is deleted. Example:

   `(delete-in 'a '(a b (d (c a)) a d))` returns `(b (d (c)) d)`.
   `(delete-in 'x nil)` returns `nil`.

3. Define a function called `skeleton`, which removes all the non-nil atoms from a list. Example:

   `(skeleton '((a (b)) (c d) (((e)))))` returns `((())(((()))))`.
   `(skeleton nil)` returns `nil`.

   Note: In GCL, since `()=nil`, you should obtain `((nil) nil ((nil)))` instead of `((())(((())))` in the first case.

3 Useful Common Lisp functions

One important feature of Common Lisp is its abundance of built-in functions. So far, we have used minimum number of these functions (e.g., `first`, `rest`, `cons`), but effective use of built-in functions increases both the efficiency and readability of Lisp programs. In this section, we introduce some useful Common Lisp functions.

3.1 Association lists

In Lisp, the most common data structure is simply a list. For example, suppose we are dealing with a database of people, and each person has four associated attributes—a name, an age, an address, and a nationality. We could represent each `person` item as a four-element list, e.g.,

```
("John Major" 51 "10 Downing Street, London" British)
```

This would then require four accessing functions:

```
(defun person-name (x) (first x))
(defun person-age (x) (second x))
(defun person-address (x) (third x))
(defun person-nation (x) (fourth x))
```

Note that `first`, `second`, `third`, `fourth` are all functions of CommonLisp which return the first, second, third and fourth element of a list, respectively.

This technique is acceptable for rather simple examples, but becomes messy for larger ones.

For simple tables of information, in which data is to be stored against a "key" (i.e. a data item labelling each entry in the table), an *association list* (or A-list) may be used, and can be accessed using the built-in function `assoc`. `assoc` searches a list in sequential order and returns the first item in it whose first element matches a given argument. The list should be set up in the form:

```
((<key1> <value1a> <value1b> <value1m>)
 (<key2> <value2a> <value2b> <value2m>)
 ....
 (<keyn> <valuena> <valuenb> <valuenm>))
```

For example, a list of offices and telephone numbers:

```
((Hardy C51 2434)
 (Lee E46 2420)
 (Price E48 2444)
 (Olivier C48 2447))
```

By assigning this association list to the variable `lecturers` using `setq`, we can obtain the following using `assoc`:

`(assoc 'Hardy lecturers)` returns `(Hardy C51 2434)`
`(assoc 'Olivier lecturers)` returns `(Olivier C48 2447)`

Notice that `assoc` returns the whole entry, including the key. We can then define functions to access necessary data from this database. For example, to obtain the phone number of a lecturer, we can define `phone-number`, with a name and the name of the database as arguments.

```
(defun phone-number (name database)
   (third (assoc name database)))
```

and this can be used as follows: `(phone-number 'Hardy lecturers)` returns `2434`.

To add a new entry to the association list, we can use the function `acons`, with three arguments, the key, the data, and the association list. The function `acons` returns a new association list by *cons*ing the key and the data together, and then *cons*ing the result onto the association list given. For example, if we want to add the new entry `(Hunt C47 2537)` to the association list `lecturers`:

```
> (acons 'Hunt '(C48 2537) lecturers)
((HUNT C47 2537) (HARDY C51 2434) (LEE E46 2420) (PRICE E48 2444)
(OLIVIER C48 2447))
```

Exercise

1. A binary tree can be represented as an association list. For example, the tree in Figure 1 can be represented as

   ```
   (setq b-tree '((55 25 70) (25 12 30) (12 6 15) (6 nil 7) (7 nil nil)
                  (15 nil nil) (30 nil nil) (70 62 80) (62 60 68) (60 nil nil)
                  (68 nil nil) (80 nil 99) (99 nil nil)))
   ```

   We can then use `assoc` to find the descendants of a node:

   `(assoc 6 b-tree)` returns `(6 nil 7)`.

   Define a function called `bsearch`, which takes three arguments, `key`, `root` and `tree`. The parameters `key` and `root` are numbers and `tree` is a binary tree in the form of an association list. The function `bsearch` should search through `tree` starting at `root` to determine if `key` is anywhere in `tree`.

3.2 More functions on lists

nth As introduced briefly in an earlier section, you can use `first`, `second`, `third`, etc., to get the element at each position in a list. This goes up only to `tenth`. However, you can use `nth` to obtain the n-th element (0-based) of a list. So `(nth 0 '(a b c))` returns `a` and `(nth 1000000 big-list)` returns the 1000000th element of `big-list`.

length This takes a list as an argument and returns the number of (top-level) elements in the list. Example:`(length '(a b (c d) e))` returns `4`.

last Takes a list as an argument and returns the list which contains only the last element of the original list. Example:`(last '(a b c d))` returns `(d)`.

butlast This function takes a list as an argument, and creates and returns a list with the same elements as its argument, but missing the last element. Example: `(butlast '(a b c d))` returns `(a b c)`.

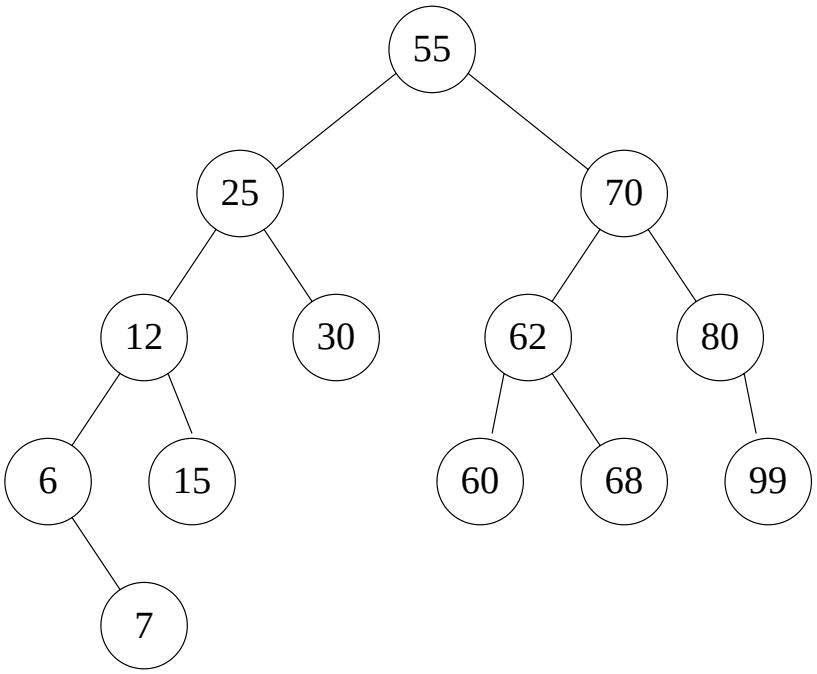

Figure 1: A binary tree.

remove This function takes an item and a list, and returns a list with the item removed from the original list. Example: `(remove 'c '(a b c d))` returns `(a b d)`.

sort This sorts a list according to a comparison function. At the moment, we restrict the use of this function to sorting lists of numbers in ascending or descending order. To sort a list of numbers, say `(5 3 7 2 9)` in ascending order, we can call `(sort '(5 3 7 2 9) #'<)`. `<` is the comparison function here, indicating that two items in the list are compared using `<` and reordering the list so that two neighbouring elements maintain this relation. Don't worry about `#` now, and don't forget to put it with a quote before the comparison function. To sort a list of numbers in descending order, the comparison function `>` is used.

Exercise

1. Write a function called `read-and-sort`, which takes an argument which is either `a` or `d`. If `'a` is given as the argument, `read-and-sort` reads five numbers from the user, and returns a list containing these five numbers sorted in ascending order. If `'d` is given as the argument, then it is sorted in descending order. Example:

```
(read-and-sort 'a)
10
8
1
5
3
(1 3 5 8 10)
```

2. Define a function `sort-list` which takes a list. Each element of the list is a list containing a number as its first element, e.g., `((5 john 45) (9 mary 18) (2 tom 22) (6 jerry 31))`. `sort-list` returns the list sorted, in ascending order, by the first elements of each element. Example:

   `(sort-list '((5 john 45) (9 mary 18) (2 tom 22) (6 jerry 31)))` returns `((2 tom 22) (5 john 45) (6 jerry 31) (9 mary 18))`.