CS262 Artificial Intelligence Concepts

Worksheet 2

1 Defining our own functions

As all lisp programs are built up out of functions we need some way of defining our own functions. The special form `defun` provides the necessary facilities:

```
>(defun double (num)
      (* num 2) )
DOUBLE

>(setq x 3)
3

>(double x)
6
```

The general form of `defun` can be given thus:

(defun <function-name> ([parameter ...]) <function-body>)

This shows that a function must always have a name (that must be an atom), then follows the parameter list, and finally the function body specifies what the function does. Notice that the parameter list can contain any number of parameters, including none, (for an example of a function call with no parameters, remember the function `(talk)` in worksheet1). But the parameter list must always be present in the definition, and so a function with no parameters will be defined thus:

```
>(defun square-pi ( )
      (* pi pi) )
```

The function-body specifies what is to be performed; on execution each parameter is replaced with its bound value and the functions in the body are then evaluated.

2 Global and local variables

In worksheet 1, section 1.4 we used `setq` (or `setf`) to assign values to variables. These are **global variables** as they are visible to all functions that wish to use them. In contrast, the variable `num` used in function `double` is a **local variable**, as are all formal parameters in functions.

Thus, in the call:

```
>(double 45)
90
```

45 is bound to `num` during the evaluation of `double`. But afterwards, if we try:

```
>num
Error: The variable NUM is unbound.
```

This is because num is unbound **outside the scope of the function**.

Any variables used in a function that are not listed as parameters are known as "free variables" and will take whatever values have been assigned outside the function. The Lisp interpreter will normally issue warnings about such variables.

Good software engineering practice says that we should eliminate or at least minimise the number of global variables in any program. If we must use them, then they should be clearly distinguished by adding * to both ends of their names, e.g. *max-value*, *constant-k* etc.

$\Longrightarrow$ **Always use * in global variable names.**

3 More built-in list functions

Append is a very useful list processing function. It merges two or more lists into a single list.

```
>(append '(a b) '(c d e))
(a b c d e)

>(append '(3 4) '(5) '(6 (7 8)))
(3 4 5 6 (7 8))

>(list '(a b) '(c d e))
((a b) (c d e))

>(cons  '(a b) '(c d e))
((a b) c d e)
```

The three list functions `append`, `cons` and `list` are very useful and you should study how they work. `cons` always takes only 2 arguments and inserts the first argument at the beginning of the second. `list` puts its arguments as elements of a new list and `append` takes the elements of the lists it is given and makes a new list of them.

$\Longrightarrow$ **Make sure you understand the differences between `append`, `cons` and `list`.**

The function `reverse` reverses the order of the elements in its argument, but notice that it does not reverse any embedded lists.

```
>(reverse '(3 4 5 6 7))
(7 6 5 4 3)

>(reverse '((a b) g h (c d e)))
((c d e) h g (a b))
```

The function `last` takes one list and returns a list consisting of the last element:

```
>(last '(3 4 5 6 7))
(7)

>(last '((a b) g h (c d e)))
((c d e))
```

Exercises

1. Define a function called `sqr` that returns a list of the perimeter and the area of a square. A single parameter gives the length of one side:

```
>(sqr 5)
(20 50)
```

2. Write a function called `snoc` that is the opposite of `cons`. Instead of inserting an item at the beginning of a list, it inserts it at the end:

```
>(snoc 'a '(g h (c d e)))
(g h (c d e) a)
```

3. Define a function called `ends` that has one argument and returns a list containing the first and last elements in that argument:

```
>(ends '(3 4 5 6 7))
(3 7)

>(ends '((a b) g h (c d e)))
((a b) (c d e))
```

4 Predicates and tests

An important part of any programming language is the ability to make decisions and perform conditional tests. To enable conditional processing, a function needs to perform *tests*, and take different actions depending on the outcomes of those tests. For example, in case of `first` (i.e., `car`), you want to test whether the argument provided was a list. If not, we might want to raise an error, or take some different action.

We first find out how these tests can be performed, and proceed to implement conditional processing.

Predicates take arguments like functions, but they return information about their arguments. For example, the predicate `atom` takes one argument and returns `t` ("true") if it is, and `nil` ("false") if it isn't. Similarly, the predicate `listp` returns `t` when the argument is a list, and `nil` otherwise. Note that these two predicates are complementary.

```
>(atom 'tree)
T

>(atom '(tree))
NIL

>(listp 'tree)
NIL

>(listp '(tree))
T
```

While values `t` and `nil` are returned for these tests, in most cases, the important distinction in Lisp is between `nil` and non-`nil` values:

$\Longrightarrow$ **Any non-`nil` value is in practice considered to be "true".**

Like other functions, predicates can take arguments which are themselves function calls. For example:

```
>(listp (list 'a 'b 'c))
T

>(atom (cons 'a '(b c)))
NIL
```

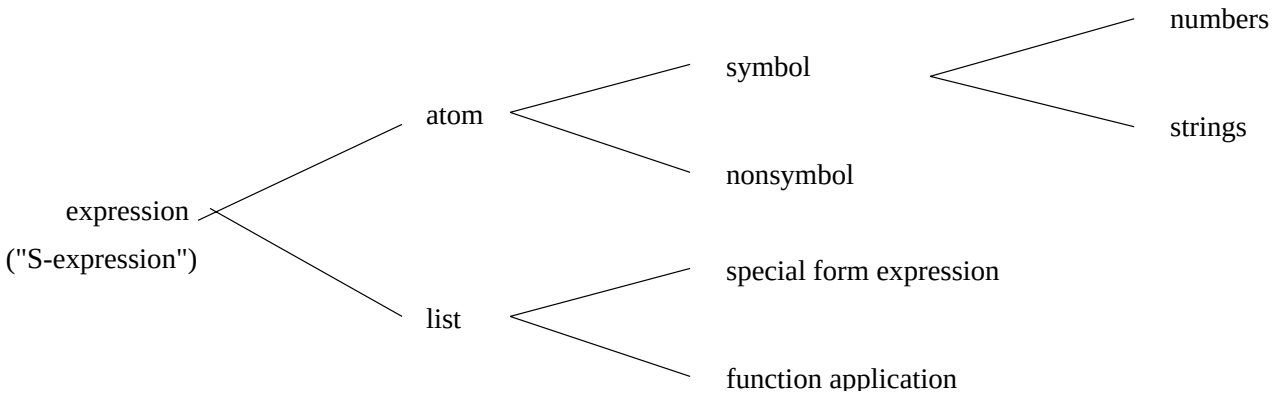

Figure 1: Lisp expressions

```
>(setq x '(a b c))
(A B C)

>(listp x)
T

>(atom (first x))
T
```

It is important to note that while `(atom nil)` evaluates to `t`, `(listp nil)` also evaluates to `t`. This is because:

$\Longrightarrow$ **the atom `nil` is equivalent to the empty list ().**

Notice, however, that `(atom t)` is `t` but `(listp t)` is `nil`.

Also both `t` and `nil` don't need to be quoted.

All numbers are also atoms.

It might be useful to remember the classification of data types in Lisp as illustrated in Figure 1.

Here are some more predicates in Common Lisp which are useful:

`numberp` returns `t` if its argument is a number and `nil` otherwise.

`zerop` returns `t` if its argument evaluates to 0 and `nil` otherwise.

`null` returns `t` if its argument evaluates to `nil` and `nil` otherwise.

There are some predicates which take more than one argument:

`equal` returns `t` if its two arguments evaluate to the same value and *nil* otherwise.

`<` returns `t` if the value of its first argument is less than that of its second argument, and `nil` otherwise.

`>` returns `t` if the value of its first argument is greater than that of its second argument, and `nil` otherwise.

`member` takes two arguments and returns `nil` if the first argument is not a member of the second argument (which must be a list); otherwise, it returns the tail of the second argument, beginning where the first argument appears.

It is important to notice that `member`, and many other functions that test for equality between two items, use the rather strict function `eq` by default. `eq` tests that two items are actually the same thing rather than they look identical. So for most general cases it is better to make such list comparison functions use `equal`. This can be done with the use of the keywork `:test` which stipulates the testing function to be used, e.g.

```
(member '(a b) '(w e (a b)) :test 'equal)
```

$\Longrightarrow$ **Use `:test 'equal` in list testing functions**

Exercises

1. What values will be returned from the evaluation of the following expressions?

```
(numberp 10)
(numberp '(25))
(numberp 'john)
(numberp -6.5743)
(zerop 5)
(zerop nil)
(zerop 0)
(zerop (- 23 (+ 15 8)))
(null nil)
(null '(a b c))
(null (numberp 'harry))
(equal t t)
(equal 5 6)
(equal 'g '(g))
(equal (first (rest '(a b c))) 'b)
(< 5 6)
(< 5 5)
(> 8 3)
(> (+ 4 5) (* 2 4))
(null (< (* 8 2) (+ 15 3)))
(member 'a '(d f a g h))
(member 'y '(l k z))
(member 'b '(d (b) g))
(cons 'x (member 'u '(s t u v)))
```

2. Define a function called `compare` which takes two arguments that are numbers. If the first number plus 10 is greater than twice the second number, then `compare` returns `t`; otherwise, it returns `nil`. Example: `(compare 5 5)` returns `t`.

3. Define a function called `palp`, which takes a list and tests whether that list is a palindrome. (Note: a palindrome is a list that reads the same backward and forward.) If the list is a palindrome, `palp` returns `t`; `nil` otherwise. Example: `(palp '(a b c c b a))` returns `t`. `(palp '(look kool))` returns `nil`.

4. Define a function called `negnum`, which takes one argument that is a number, and returns a two-element list. The first element of the list is `t` if the number is 0 and `nil` otherwise. The second element of the list is `t` if the number is negative and `nil` otherwise. Example: `(negnum -5)` returns `(nil t)`. `(negnum 5)` returns `(nil nil)`.

5 Conditionals

Using the predicates described in the previous section, we can start using conditionals.

Suppose we want to write a function, `which-make` which takes one argument and returns `vauxhall` if the argument was `nova`, returns `ford` if the argument was `fiesta`, and `unknown` otherwise. This can be achieved by the following:

```
(defun which-make (car)
    (cond ((equal car 'nova) 'vauxhall)
          ((equal car 'fiesta) 'ford)
          (t 'unknown)))
```

This function uses the `cond` statement, which is similar to the case-statement in Ada. A `cond` takes one or more arguments, each of which must be a list. These arguments are called *cases*, and each one contains at least two arguments: the first argument is a test to perform, and the remaining elements are actions to perform when the test returned a **non-nil** value (not necessarily `t`).

There are three cases in the example above:

1. `((equal car 'nova) 'vauxhall)`
2. `((equal car 'fiesta) 'ford)`
3. `(t 'unknown)`

Each case is evaluates one at a time, from first to last. As soon as a test is "true" (non-nil), the action(s) are evaluated in that case. Then the evaluation of the cond-statement terminates there, i.e., no further cases are attempted, and Lisp returns the value of the final action taken. For example:

```
>(which-make 'fiesta)
FORD
```

Here, the value `fiesta` is assigned to the parameter `car`. The first case in the cond-statement is evaluated: the test `(equal car 'nova)` returns `nil`, so this case is discarded. In the second case, the test `(equal car 'fiesta)` returns `t` (a non-nil value) so the corresponding action is evaluated; in this case, the evaluation of the literal `'ford`, which evaluates to `ford`. Since no more cases are considered, this is the final action taken, and the function `which-make` returns `ford`. Another example:

```
>(which-make 'clio)
UNKNOWN
```

In this case, the test for the first two cases both fails (returns `nil`). However, it is captured by the test for the third case, which is `t`. Since `t` always evaluates to `t` (a non-nil value), this test always succeeds, and the corresponding action is taken.

Such a case is referred to as an *else-case*, since it is the case where the action is performed if all the previous tests have failed. This is analogous to *others* case in the Ada case construct.

We can summarise the cond-statement as follows:

```
(cond ( <test-1> <action-1-1> <action-1-2> ...)
      ( <test-2> <action-2-1> <action-2-2> ...)
      ...
      ( <test-n> <action-n-1> <action-n-2> ...)
)
```

Exercises

1. Define `carlis`, which takes one argument. If the argument is a non-empty list, then it returns the first element of the list. However, if the argument is the empty list, the `carlis` returns the empty list. If the argument is an atom, `carlis` simply returns just that atom. Example:

   `(carlis '(tree weed)` returns `tree`
   `(carlis 'flower)` returns `flower`
   `(carlis nil)` returns `nil`

2. Define `checktemp`, which takes one argument, a temperature, and returns an atom which describes how warm it is. A global variable, `hightemp`, stores a high temperature. Another global variable, `lowtemp`, stores a low temperature. If the temperature is above `hightemp`, `checktemp` returns the atom `high`; if the temperature is below `lowtemp`, `checktemp` returns `low`. If it is in between, it returns `medium`. The variables `hightemp` and `lowtemp` should be set to 90 and 30, respectively, using `setq`. Example:

   `(checktemp 100)` returns `hot`

3. Define a function `classify`, which determines the type of its argument. If the argument is a non-empty list, return the literal `list`. If the argument is a number, return `number`. If the argument is an atom, return `atom`. If the argument is `nil`, return `nil`. Be careful to order your tests properly. Example:

   `(classify 'a)` returns `atom`
   `(classify '(x y))` returns `list`
   `(classify nil)` returns `nil`
   `(classify 5)` returns `number`

6 Logical functions

In addition to the predicates, we can use *logical functions* to combine predicates to make more elaborate tests. Here are three useful logical functions, `not`, `or`, and `and`.

not This takes one argument and if the argument evaluates to `nil`, it returns `t`; if the argument evaluates to a non-nil value, it returns `nil`.

or This takes one or more arguments. It evaluates its arguments from left to right, and returns the first value it encounters that is non-nil. If all the arguments evaluate to `nil`, then `or` returns `nil`.

and This takes one or more arguments. It evaluates its arguments from left to right, if it encounters an argument which evaluates to `nil`, `and` immediately returns `nil` without evaluating any more arguments. If every argument evaluates to a non-nil value, `and` returns the value of its final argument.

Exercises

1. Think how each of these expressions would be evaluated.

```
(not nil)
(not t)
(not '(a b c))
(not (atom '(a b c)))
(or t nil)
(or nil nil)
(or (numberp 'a) '(a b c) (+ 6 5))
(and 5 nil)
(and 'a 'b)
(and (listp nil) (atom nil))
```

2. Define `lisnump` which takes one argument. If the argument is a number, or a list, it returns `t`. Otherwise, it returns `nil`. Implement this without using `cond`.

3. Define `samesign` which takes two numbers as arguments, and returns `t` if both arguments have the same sign. Implement this without using `cond`. Example:

   `(samesign 0 0)` returns `t`
   `(samesign -2 -3)` returns `t`
   `(samesign -2 5)` returns `nil`

4. Define `classify-sentence`, which takes as an argument a list encoding of sentences. It classifies sentences as either *questions*, *active sentences*, or *passive sentences*. The following rule can be used for this classification:

   - *Questions* always begin with "why" or "how".
   - Any sentence that contains both "was" and "by" is *passive*.
   - every other sentence is *active*.

   For example:

   `(classify-sentence '(john gave mary a book))` returns `active`
   `(classify-sentence '(why did john give mary a book?))` returns `question`
   `(classify-sentence '(mary was given a book by john))` returns `passive`

7 Error raising

If you evaluate an expression that violates the rules of Lisp in some way, the Lisp system will signal an error. Since Lisp is not a strongly typed language like Ada, on many occasions you should also arrange for your own function definitions to indicate errors when they detect that something has seriously gone wrong. We can raise an error by the function `error`, which takes a double quoted string as an argument. For example, consider the following definition of double:

```
(defun double (num)
    (* num 2))
```

Since it relies on the user to provide a correct input, when the user supplies a non-number, it complains:

```
>(double '(a b c))

Error: (A B C) is not of type NUMBER.
Fast links are on: do (use-fast-links nil) for debugging
Error signalled by *.
Broken at *.  Type :H for Help.
>>
```

Although this will indeed make you realise that something has gone wrong, it isn't informative enough especially if this function call appeared within a complicated series of function calls. To make it more informative, we want to raise our own errors to a wrong input to `double`.

```
(defun double (num)
    (cond ((not (numberp num))
           (error "Error at DOUBLE: The argument must be a number."))
          (t (* num 2))))
```

Now to the same incorrect input as before, it responds with the error message:

```
>(double '(a b c))

Error: Error at DOUBLE: The argument must be a number.
Fast links are on: do (use-fast-links nil) for debugging
Error signalled by COND.
Broken at ERROR.  Type :H for Help.
>>
```

Exercises

1. Define a function `addit` which takes two arguments, an item and a list and searches for the item in the list. If it finds the item in the list, it returns `found`. If it does not find the item then it adds the item onto the end of the list. But if the first argument is an empty list, avoid adding it to the and of the old list and just return the old list untouched. Make the function raise an error whenever the second argument provided is not a list. Example:

   `(addit 'a '(c d e))` returns `(c d e a)`
   `(addit 'a '(a b))` returns `found`
   `(addit nil '(a b))` returns `(a b)`
   `(addit 'a 'b)` raises an error with a meaningful error message

2. Define a function `combine` which takes two arguments and does the following:

   - If either argument is `nil`, it returns `nil`.
   - If both arguments are numbers, it returns the sum of the numbers.
   - If both arguments are atoms (but the conditions above do not apply, i.e., neither argument is `nil` and at least one argument is not a number), then the function returns a list of the arguments.
   - If both arguments are lists (but neither one is `nil`), the function appends them returns the resulting list.
   - Otherwise, it inserts the argument that is an atom into the argument that is a list and returns the resulting list.

   Hint: Do not assume that your function will have five cases or that they should be coded in the order of the five statements above.