Tutorial Sheet: Lisp foundations

Christopher Buckingham

Note: Practical classes will build on and reference previous handouts so please bring them all with you, each week.

These exercises are building on your familiarity with the Lisp programming environment. The exercises are not designed to produce functions that you necessarily want to use later but are about providing you with the skills required for creating more sophisticated AI programs such as a rule-based expert system that the tutorials will tackle later in the module.

Today, you will increase your understanding of how to process lists, which is at the heart of Lisp programming. The assumption is that you have reviewed the accompanying "Introduction to Lisp" document with this lab class.

1 The double function

The *Introduction to Lisp* document gives an example of a function that takes a list and returns a list with all the elements doubled:

```
;; Takes a list of numbers and returns a list containing
;; each original list element multiplied by 2.
;;
(defun double (numbers)
  (let ((double-list nil))
    ;; double-list is the value returned by dolist
    ;; if you leave it out, dolist returns nil
    (dolist (element numbers double-list)
            (setf double-list (cons (* 2 element) double-list)))))
```

1. Type the function into emacs and load it into the Lisp interpreter.
2. Try giving different lists to it and watch the results.
3. Modify the function so that it returns a list in the same order as the one passed to it as an argument. *hint: try the lisp function called reverse.*
4. Modify the function so that it doubles all numbers *except* number 1, which is put into the new list as the word `one` instead. Hence the function will return `(one 4 6 4 one 8)` when passed the list `1 2 3 2 1 4`.

2 Exercise: Extract-all function

Define a function called *extract-all* that takes two parameters, an element and a list, and returns a new list with all occurrences of the element removed:

e.g. the function call `(extract-all 'n '(i n t e l l i g e n c e))` will return `(i t e l l i g e c e)`.

2.1 Hints

1. You need a local variable to hold the new list.
2. Go through the list one by one and see whether the element equals the one to be extracted.
3. If the element does not equal the one to be extracted, put it into the new list.

2.2 More efficient version

The `when` and `unless` functions can be used instead of `if` in many cases, if there is only one choice you are interested in.

2.3 When and unless

```
(when <test> <then forms>)

(unless <test> <else forms>)
```

Both *when* and *unless* can have any number of forms.
They return the value given by the last one.

```
(unless (= 2 4)
  'hello)
HELLO

(when (= 2 4)
  'hello)
nil
```

See if you can make the extract-all function a bit more efficient using when or unless.

3 Exercise: Extract function

Define a function called *extract* that takes two parameters, an element and a list, and returns a new list with *only the first occurrence* of the element removed:

```
>(extract 'n '(i n t e l l i g e n c e))
(I T E L L I G E N C E)
```

4 keep-vowels function

Define a function called *keep-vowels* that takes a list and returns a new list containing only the vowels:

```
>(keep-vowels '(i n t e l l i g e n t))
(E I E I)
```

5 Colour and colour changes

Suppose a Lisp function called `colour` takes two arguments, the first being a colour and the second a list of colours. It returns the number of times the first colour occurs in the list.

For example, the following call to the function would return a value as shown:

```
> (colour 'green '(purple green purple))
1
```

5.1 Colour change

Suppose a Lisp function called `mauve` takes a list of colours and returns the same list but with the name "purple" replaced by the name "mauve". For example, the following call to the function would return a value as shown:

```
> (mauve '(purple green purple))

(mauve green mauve)
```

Write the Lisp code that defines the required function.