Tutorial Sheet: Introduction to Lisp

Christopher Buckingham
Computer Science
University of Aston

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

1 General background to tutorials and tutorial sheets

The idea behind the tutorials is for you to begin to understand how Lisp works and how to use it for modelling aspects of the world. The tutorial sheets must be read in conjunction with the lecture notes and your understanding of those lectures, otherwise you will not comprehend the context behind the sheets.

PLEASE NOTE the difference between l and 1 Sometimes, I use `l` as a variable name to represent any list, l, which is different to the number one: `1`. So make sure you don't enter a `1` when it should be an `l`; the tops are different but I do appreciate that it is not that easy to spot. The lisp interpreter certainly will, though, and you should get used to knowing what the likely problem is when it puts you into the debug mode.

1.1 The way to approach Lisp tutorials

Think, experiment, and play. Each tutorial sheet may take more than one hour to complete and you are expected to put in extra hours between the sessions accordingly. Don't forget there are 100 hours of total study time for this module, 70 of which are outside the contact hours (lectures and labs).

2 Simple list manipulation tasks

TASK In Emacs, create a file called `tutorial-exercises.lisp` and define a function called *tut* that loads in the actual file name called *tutorial-exercises.lisp* and save the file as *tutorial-exercises.lisp*.

- If you don't know how and why to do this, refer back to the previous tutorial because you had to do something very similar then.

TASK In the same file and after your function definition, set up a global variable called **opinion* and give it a value as follows:

```
(setf *opinion '(some of the earliest languages are still the greatest))
```

Note that global variables are ones that exist in the interpreter outside any particular function call. Local variables only exist while a function executes and so are not permanently accessible to other functions.

IMPORTANT! APOSTROPHES AND CUTTING AND PASTING FROM A PDF If you cut and past the line of code above from the pdf file into clisp, the character code for the apostrophe is not translated correctly. It will create an error so make sure you delete the apostrophe in your clisp and then type it again from your keyboard.

TASK Start clisp in the same directory as your lisp file and load the file into the lisp interpreter.

- Remember that the first time you do this, you have to use the `load` function with the complete file name and not your function call defined in the file because the lisp interpreter does not know the function you defined until the file has actually been loaded once (this puts the `tut` function into the interpreter memory).
  So, for the first time when you are in clisp you need:

  ```
  (load "tutorial-exercises.lisp")
  ```

  then after that, the Lisp interpreter will know about the function you defined called `tut` and you can call this to load the file in whenever you change anything in it, as follows:

  ```
  (tut)
  ```

TASK Type `*opinion` into the lisp interpreter and note what is returned.

TASK Type `*second-opinion` into the interpreter. Why has the interpreter returned an error?

Remember that you are now in the debug mode. Have a look at what the debugger is saying about the error because this is usually enough to know what has caused it.

Exit from the debug mode with `:Q`.

TASK Return to your Emacs file and set up a global variable called `*second-opinion` with the value

```
(lisp is a flexible language and good for symbol processing).
```

TASK Reload the file into the Lisp interpreter and Type `*second-opinion` again, noting the different result.

- Note that you can now run your short function call to reload the file because it is in the interpreter's memory.
  `(tut)`

TASK What happens if you type `(first *opinion)`?

TASK What happens if you type `(rest *opinion)`?

TASK What happens if you type `(first (rest *opinion))`?

TASK type `*opinion` and see whether it has changed as a result of the function calls.

TASK Write down the definition of the function `first` and `rest`: i.e. *exactly* what they return and whether they change their argument (the list passed in).

The next set of information and tasks looks at simple list manipulation commands.

- The `list` function: takes its arguments and puts them into a list. e.g. `(list 'a 'b 'c)` returns `(a b c)`.

  Remember that an apostrophe before a symbol or list tells the clisp interpreter **not** to evaluate it. The interpreter simply returns the symbol or list itself as the value and does not treat the symbol as a variable or the list as a function call.

- You will have seen that `first` returns an atom but `rest` returns a list. If you want to create a new list containing the first member of each of the opinion lists, you can use the `list` function: `(list (first *opinion) (first *second-opinion))`.
- **TASK** Create a new list containing the first member of each of the opinion lists.

  Note that the arguments to the `list` function call are themselves function calls but, as with all functions, they return a value and it is this value that the `list` function uses. If you are unsure what is happening, try calling each of the arguments in the lisp interpreter separately to see what it returns:

  ```
  (first *opinion)
  ```

- **TASK** What will be returned if you send the following arguments to `list`: '(a b) 'c 'd '(e f)?
- If you want to join two lists together, you use `append`, remembering that it *must* have arguments that are lists.
- **append function:** `(append list1 list2)` returns a list containing all the members of list list1 followed by all the members of list list2.
- **TASK** What happens if you append the rest of *opinion with the rest of *second-opinion?
- **TASK** What happens if you append the first of *opinion with all of *second-opinion?
  - Why is there an error? If you don't know, read the items above again because the answer is there.
- **TASK** How can you append the first of *opinion with all of *second-opinion without getting the error that occurred above?

Graham Taylor is an ex Watford, England, and Aston Villa football manager (plus some other clubs I have probably forgotten). He is famous for saying "Do I not like that" instead of "I do not like that".

TASK Set up a variable called `*taylor-speak` with the value `(do i not like that)`.

TASK Using different combinations of the `first` and `rest` functions, along with `list` and `append` as necessary, write the lisp code that will convert taylor-speak into better english and assign the result to a variable called `*proper-speak` which will obviously have the value `(i do not like that)`.

It might have been easier to do it using some other useful list-processing functions. Here are some:

member this function takes two arguments, the second of which *must* be a list. It returns nil if the first argument is not a member of the list. If the first argument is a member of the list, it returns the rest of the list that follows and includes the first element. e.g. `(member 'b '(a b c))` will return `(b c)`.

second this function returns the second member of a list. e.g. `(second '(a b c))` returns `b`.

third, fourth, fifth, . . . tenth all act like `second`, but returning their corresponding element.

It is easier to manipulate lists using these additional functions.

TASK Using the functions `first`, `second`, and `member`, create the `*proper-speak` list from `*taylor-speak`.

TASK Using any of the different list manipulation functions you now know, create a new list called `*third-opinion` from the `*opinion` and `*second-opinion` lists that returns the following list: `(lisp is a flexible language and still the greatest)`.