Tutorial Sheet: Knowledge structures and accessors
CS2320, Introduction to Artificial Intelligence

Christopher Buckingham

Note: I have used the letter `l` to represent a variable name called `l` for list. It looks like the number `1`, though, so be careful when following the instructions. It should be obvious that it is a letter not a number because you can't have numbers as a variable name.

APOSTROPHES!!! Watch out for apostrophes when cutting and pasting. The wrong one causes lisp to behave very strangely and the wrong ones are part of pdf files because of the font mismatching. You NEED TO USE `'` AND NOT `` ` ``. Usually, the one you want is under the @ symbol but, as always, it depends on the keyboard.

1 Representing hierarchical knowledge: the "house" example

This tutorial starts thinking about knowledge representation with lists and how to access different parts of the knowledge from the lists. It is one way of approaching intelligent knowledge-based systems and so will begin to build the tools you require for your own chosen expert system.

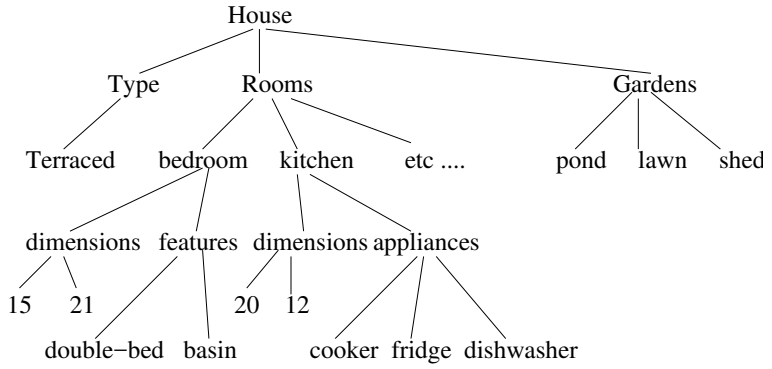

Figure 1: *House hierarchy*

An earlier lecture discussed how the properties of a house can be represented by a hierarchical structure and how this can also be represented easily in Lisp. Figure 1 shows a simple example.

Here is one way of translating the hierarchy to a lisp list that retains the structural information:

```
(setf *house
      '((type terrace)
        (rooms ((kitchen ((dimensions (20 12))
                          (appliances (cooker fridge dishwasher))))
                (bedroom ((dimensions (15 21))
                          (features ((double-bed washbasin)))))))
        (garden (pond lawn shed))))
```

Note that the variable is a global one and we put an asterisk in front of the name as a convention to make it more visible. It means that you can easily see in your function code when it is operating on a global variable as opposed to one that has been passed in by a parameter. Global variables are held in the Lisp interpreter external to any particular function and so can be operated on by any functions you define.

Generally speaking, global variables should be avoided because it is difficult to see how they are being changed. Also, the fact that they can be accessed by any function means they can be changed by mistake. It is better to limit the scope of variables by declaring them within a function or passing them into functions as parameters. However,

globals are sometimes more efficient, in which case we mark them out in some way so that they are very visible and distinct from other variables.

2 Setting up a variable in lisp

1. Create a file in Emacs called "house.lisp" and type the `setf` function above to set up the *house variable.
2. Check that the house has the correct brackets by going to each line of the lisp and hitting the tab key. This will line the lisp code up correctly, according to your brackets.
3. If any lines are incorrectly aligned, then your brackets are wrong. Check and amend them. If, on the other hand, you typed it in correctly, remove the last bracket on the line with the kitchen in it. What happens now if you press the tab key on the lines beneath?
4. Experiment with removing or adding brackets and lining the code up with the tab key. This is an important way of checking that the list structure is correctly representing the hierarchical relationships designed for it.
5. Note also that when you add a closing bracket, Emacs jumps to the earlier matching opening bracket, which is another way of making sure you have the right brackets in each place.

3 Functions to access the list structure: assoc

The `assoc` function will return information from an association list:

1. association lists are lists of property-value pairs, such as `((eyes "blue")(hair "brown")(age 22))`
2. `assoc` takes a specific property and an association list (alist for short), returning the property-value pair if the property exists in the list or `nil` otherwise:

   `(assoc 'age '((eyes "blue")(hair "brown")(age 22)))` will return `(age 22)`

   Note that you have to quote the property and the list if you don't want them to be evaluated. In other words, if they are meant to evaluate to themselves rather than represent variables or function calls.

   If you had told the lisp interpreter the following:

   `(setf prop 'age looks '((eyes "blue")(hair "brown")(age 22)))`

   then you would have produced variables that can be supplied to assoc: `(assoc prop looks)`. I hope you see the difference!

For example, if we set up the following list, l (note: the letter 'l' not the number '1'):

```
(setf l '((a 1)(b 2)(c 3)))
```

`assoc` will return a pair as follows:

```
(assoc 'b l)
(b 2)
```

TASK Type in the list, l, as defined above and experiment with assoc to return each member of the list.

TASK What happens if you type `(assoc c l)`?

TASK What happens if you type `(assoc 'd l)`?

If we wanted to write a function for getting the property from a list (it would be a bit unnecessary in this case but I am doing it just to remind you how functions are written), then you would do it as follows:

```
(defun get-property-value (property association-list)
  (second (assoc property association-list)))
```

So if you want to know the value of the `c` property in the list `l`, then you would call this function for the list defined above as follows:

```
(get-property-value 'c l)
```

and it would return 3:

```
> (get-property-value 'c l)
3
```

Remember that the property to find must have an apostrophe in front of it so that lisp doesn't treat it as a variable and try to evaluate it. Unless, of course, you had a variable that was supposed to be evaluated to provide the right property. So, for example, if you had a variable called `property` and assigned it a value:

```
(setf property 'c)
```

then you would not use the apostrophe in the function call because you want the variable to be evaluated to get its value:

```
> (get-property-value property l)
3
```

3.0.1 Reminder of the generic format for defining functions

The general approach to defining functions is as follows:

```
(defun function-name (list of parameters)
  (function code))
```

Note that the function returns whatever is the value of the last statement it evaluates.

3.1 Accessing parts of the house list

Once you have created the house list structure, you need to access different parts of it easily. For example, you might want a function that returns a list of rooms, as shown below:

```
>(get-rooms *house)

(ROOMS ((KITCHEN ((DIMENSIONS (20 12))
                  (APPLIANCES (COOKER FRIDGE DISHWASHER))))
        (BEDROOM ((DIMENSIONS (15 21))
                  (FEATURES ((DOUBLE-BED WASHBASIN)))))))
```

TASK Write the Lisp code that defines the `get-rooms` function.

Then you can use this function to create a more specific one that returns a particular room:

```
>(get-room 'kitchen *house)

(KITCHEN ((DIMENSIONS (20 12)) (APPLIANCES (COOKER FRIDGE DISHWASHER))))
```

TASK Write the Lisp code that defines the `get-room` function, using the `get-rooms` function as part of the `get-room` definition.

Hint When applying functions to a list returned by another function, be sure that the list is in the correct format; `assoc` will not work directly on the list returned by `get-rooms`. If you look back to what `get-rooms` returned, you will see it is *not* an association list (the first member is the symbol "rooms", which is the property of the property-value pair). It is the value associated with the property, `rooms`, that is the association list you need.

Functions created to "get at" parts of list structures are called *accessors*. They are crucial for allowing your expert system to retrieve and operate on particular chunks of your knowledge.

Your task for this tutorial is to elaborate on the house knowledge structure and define functions you think will be useful for accessing different parts of it.

TASK Add more detail to the house structure such as additional rooms and other features that might be important for choosing whether to live there (e.g. proximity of shops, transport links, state of repair, heating).

TASK Define accessor functions that enable you to retrieve information about the house and your extra features. This should be done in the same way as you did the get-rooms and get-room functions: by defining functions that retrieve higher-level components (e.g. rooms) first so that functions retrieving lower-level components (e.g. a particular room) can use the higher-level functions in their definitions. Example function calls and their returned values from the given house list structure above might be:

```
>(get-garden *house)
(GARDEN (POND LAWN SHED))

>(get-kitchen-appliances *house)
(APPLIANCES (COOKER FRIDGE DISHWASHER))

>>(get-dimensions 'kitchen *house)
(DIMENSIONS 20 12)
```

3.2 If you get this far ...

TASK Write a function called `rate-rooms` that takes a house and produces a rating for it, based on how well it matches the number of bedrooms. It takes 2 arguments, the first being the ideal number of bedrooms, and the second being the house to rate. It then compares the ideal number of bedrooms with those the house has, and generates a score. For example, a perfect match (i.e. the ideal and actual number of bedrooms are the same) might generate a score of 10, with a reduction in the score depending on how many more or less than ideal is the number of bedrooms in the house. You need to think of a sensible metric for reducing the score and implement it within the function. When called the function might perform as follows for a five-bedroom house:

```
(rate-bedrooms 2 *house)
4
```

You might want to use the following functions:

`length` that returns the length of a list: `(length l)`.

`*`, `+`, `-` that are maths functions operating on numbers in the expected way: `(* 4 2)` `(/ 4 2)` `(- 4 2)` `(+ 4 2)` that return 8, 2, 2, and 6 respectively.

You should also work out the algorithm for rating matches before trying to do it in lisp, so that you are clear in your own mind about the problem representation.