Session 1 – seminar trace

---

the following interaction occurs in the evaluation/debug window in the Allegro Lisp IDE...

the compiler/evaluator is active in the debug window, it will immediately evaluate & print the values of variables, etc

```
> *current-case-mode*
:case-sensitive-lower
```

it also evaluates function calls

```
> (min 10 5 2 7)
2
```

numbers (spelled using a series of digits) are eval'd to a numeric value – this looks like they are simply echoed

```
> 123
123
```

symbols are eval'd to themselves. To specify a lone symbol it must be preceded by a single quote character. NOTE: **symbols are not strings**

```
> 'mango
mango
```

if you forget the quote, Lisp will try to treat the symbol as a variable

```
> mango
Error: Attempt to take the value of the
       unbound variable `mango'.
```

typically we collect symbols together into lists, these lists must be preceded by a quote

```
> '(mango melon peach)
(mango melon peach)
```

lists can contain mixed types of value and can be nested

```
> '(I ate 1 & she ate 3 too)
(I ate 1 & she ate 3 too)

> '((car (wheels 4) (seats 5))
              (bike (wheels 2) (seats 1)))
((car (wheels 4) (seats 5)) (bike (wheels 2) (seats 1)))
```

often we structure lists to specify packets of information & manipulate collections of related information

```
> '(isa tom cat)
(isa tom cat)

> '((isa tom cat)
    (isa spike dog)
    (chases dog cat))
((isa tom cat) (isa spike dog) (chases dog cat))
```

Lisp function calls also look like Lists, first returns the first item in is argument

```
> (first '(cat dog rat frog))
cat
```

second does something similar - functions are defined upto tenth

```
> (second '(cat dog rat frog))
dog
```

rest returns all but the first item

```
> (rest '(cat dog rat frog))
(dog rat frog)
```

`->` pronounced association lookup is a function in the utils collection. An association list works a little bit like a hash table

```
> (-> '((one 1)(two 2)(three 3)) 'two)
(2)
```

we use the dot constructor "." to bind values more closely into association lists (this is often preferred)

```
> (-> '((one . 1)(two . 2)(three . 3)) 'two)
2
```

Lisp declares variables as you use them. Variables are bound (similar to assignment) using ***setf***. The following sets a variable call *numbers*

```
> (setf numbers '((one . 1)(two . 2)(three . 3)(four . 4)))
((one . 1) (two . 2) (three . 3) (four . 4))

> numbers
((one . 1) (two . 2) (three . 3) (four . 4))

> (-> numbers 'three)
3

> (setf x (-> numbers 'three))
3
```

```
> (setf x (* x 2))
6
```

the next few examples use predefined definitions (see the associated .cl file)

```
> (-> globe-data 'Asia)
((India (Capital . New-Delhi) (Population . 980))
 (Sri-Lanka (Capital . Colombo) (Population . 15)))

> (-> (-> globe-data 'Asia) 'Sri-Lanka)
((Capital . Colombo) (Population . 15))

> (-> globe-data 'Asia 'Sri-Lanka)
((Capital . Colombo) (Population . 15))

> (-> globe-data 'Asia 'Sri-Lanka 'Population)
15


> (population 'India)
980

> (combined-population 'Botswana 'Zimbabwe)
12.5


> (pop-from-country (second *countries*))
11
```

mapcar is a mapping function which repeatedly applies a fuction to a list of data & collects the results. Note the use of #' to pass a function object as a piece of data

```
> (mapcar #'first *countries*)
(Botswana Zimbabwe India Sri-Lanka)

> (mapcar #'second *countries*)
((Capital . Gaborone) (Capital . Harare) (Capital . New-
Delhi)
 (Capital . Colombo))

> (mapcar #'pop-from-country *countries*)
(1.5 11 980 15)
```

reduce is a mapping function which uses its function arg to accumulate a single result over a list of data

```
> (reduce #'+ '(1 2 3 4 5))
15

> (reduce #'+ (mapcar #'pop-from-country *countries*))
```

```
1007.5
```

reduce can be provided with ***:key*** argument to filter its data before applying its function

```
> (reduce #'+ *countries* :key #'pop-from-country)
1007.5

> numbers
((one . 1) (two . 2) (three . 3) (four . 4))

> (reduce #'* numbers :key #'rest)
24
```