Functions as Arguments

Funcall, Apply and Eval

Mapping Functions

Mapcar, MapList and Reduce

The Problem

◊ Suppose you want to write the following functions

> Sum first n integers

```
» (defun sumN  (n )
    (cond ( ( equal n 0 )  0)
          ( t  (+  n  ( sumN (1- n )) )) ))
```

> Sum of the squares first n integers

```
» (defun sumN2 ( n )
    (cond ( ( equal n 0 )  0)
          ( t  (+  (* n n )  ( sumN2 (1- n )) )) ))
```

> Sum of the cubes first n integers

```
» (defun sumN3 ( n )
    (cond ( ( equal n 0 )  0)
          ( t  (+  (* n n n )  ( sumN3 (1- n )) )) ))
```

Abstract the commonalities and only supply the variations write one function for all cases

Abstraction requires

- Passing a unary function
  - » **Identity for sum of integers**
  - » **Square for sum of squares**
  - » **Cube for sum of cubes**

- Ability to evaluate the function
  - » **(funcall ‘+ 1 2 3) --> 6**
    - > **Requires the parameters of the function to evaluate to appear at the level of the function**
  - » **(apply ‘+ (1 2 3)) --> 6**
    - > **Requires the parameters of the function to evaluate to be a list**

The abstraction

- Using funcall

```
» (defun sumInt (func n)
    (cond ((equal n 0) 0)
          ( t ( + (funcall func n)
                  (sumInt func (1- n))))
  ))
```

- Using apply

```
» (defun sumInt (func n)
    (cond ((equal n 0) 0)
          ( t ( + (apply func (list n))
                  (sumInt func (1- n))))
  ))
```

Using the Abstraction

- Now we can define or use any unary function to obtain the sum of that function applied to the first N integers.
- For example

```
» (defun double (int) (+ int int))
» (sumInt 'double 10) --> 110

» (defun square (int) (* int int))
» (sumInt 'square 10) --> 385

» (defun identity (int) int)
```

– **identity – do nothing with the integer before summing**

```
» (sumInt 'identity 10) --> 55
```

Why need funcall or apply?

- Without funcall or apply the function would look like the following.
  - ```
    (defun sumInt (func n)
      (cond ((equal n 0) 0)
            ( t ( + (func n)
                    (sumInt 'func (1- n))))
    ))
    ```
- Why does the above not work?
    ```

Why need funcall or apply? – 2

- Without funcall or apply the function would look like the following.
  - ```
    (defun sumInt (func n)
       (cond ((equal n 0) 0)
             ( t ( + (func n)
                     (sumInt 'func (1- n))))
    ))
    ```
- Why does the above not work?
  - **In the expression (func n) the symbol-function of func is used to find the function to apply.**
  - **But because func is an argument the definition is in func's symbol-value**
    ```

More Abstraction -1

◊ Here is the original abstraction

```
(defun sumInt (func n)
  (cond ((equal n 0) 0)
        ( t ( + (funcall func n)
                (sumInt func (1- n))))
))
```

◊ Can abstract further

```
(defun funInt
      (binaryFunc unaryFunc n base)
  (cond ((equal n 0) base)
        (t (funcall binaryFunc
                    (funcall unaryFunc n)
                    (funInt binaryFunc
                       unaryFunc (1- n) base)))
))
```

More Abstraction -2

- Now can do the following
  - `(funInt '+ 'double 10 0) --> 110`
  - `(funInt '* 'double 10 1) --> 3715891200`
  - `(funint '* 'double 10 0) --> 0    ; why?`

- Too much abstraction make functions too complex
  - **Judgement and experience dictate when abstraction has gone too far**

Evaluate an S-expression

- **funcall** and **apply** are based on the function **eval**
- **eval** evaluates an S-expression
  - » **(setq x (cons '+ '(2 3))) --> (+ 2 3)**
  - » **(eval x) --> 5**
  - » **(eval (eval ' x)) --> 5**
- Note: **eval** makes it possible to execute S-expressions created by a Lisp program.
- Lisp contains the essentials for Artificial Intelligence
  - » **Dynamically construct S-expressions which represent functions**
  - » **Dynamically execute them**

Mapping functions

- Programs such **sumInt** are representative of the more general case of applying functions to lists of arguments
- Lisp provides useful abstractions

Mapcar

- ◊ (mapcar function arg1 arg2 ... argN)
  - » **apply the function to the first of each of a list of argments**
  - » **recursively apply to the second of each argument, etc**
  - » **collect all the results in a list**

- ◊ ((function (car arg1) (car arg2) ... (car argN))
  (function (cadr arg1) (cadr arg2) ... (cadr argN))
  (function (caddr arg1) (caddr arg2) ... (caddr argN))
  (function (cadddr arg1) (cadddr arg2) ... (cadddr argN))
  ...)

Maplist

◊ (maplist function arg1 arg2 ... argN)

» **apply the function to the arguments**

» **remove the first item from each argument**

» **collect all the results in a list**

◊ ((function arg1 arg2 argN)
(function (cdr arg1) (cdr arg2) ... (cdr argN))
(function (cddr arg1) (cddr arg2) ... (cddr argN))
(function (cdddr arg1) (cdddr arg2) ... (cdddr argN))
...)

Reduce

- (reduce function list)
  - **apply the function to the first two items in the list**
  - **recursively apply to the result and the next item on the list**
- (function

```
...
 (function
   (function
     (function (car list) (cadr list))
      (caddr list))
       (cadddr list))
        ...
         (cad...dr list))
```