Functional Programming

also see the notes on functionals

History

- 1977 Turing[1] Lecture John Backus described functional programming

  "The problem with 'current languages' is that they are word-at-a-time" [2]

  > **Notable exceptions then were Lisp and APL**

  > **Now ML, Haskell and others**

– 1 **Turing award is the Nobel prize of computer science.**

– 2 **"Word-at-a-time" translates to "byte-at-a-time" in modern jargon. A word typically held 2 to 8 bytes depending upon the type of computer.**

Meaningful Units of Work

◊ Work with operations meaningful to the application, not to the underlying hardware & software

» **Analogy with word processing is not to work with characters and arrays or lists of characters**

» **But work with words, paragraphs, sections, chapters and even books at a time, as appropriate.**

Requires Abstraction

- ◊ Abstract out the control flow patterns
- ◊ Give them names to easily reuse the control pattern
  - » **For example in most languages we explicitly write a loop every time we want to process an array of data**
  - » **If we abstract out the control pattern, we can think of processing the entire array as a single operation**

Example 1

◊ Consider the inner product of two vectors

< a1, a2, ... , an > ⊕ < b1, b2, ... , bn >

==> (a1*b1 + a2*b2 + ... + an*bn)

◊ In Java or C/C++, the following is an algorithm

```
result  = 0;
for (i = 1 , i <= n , i++) {
    result = result + a[i]*b[i];
}
```

◊ Note the explicit loop (or recursion) and introduction of variables **result**, **i** and **n** (have to explicitly know the length of the vectors

Example 1 – FP form

- innerProduct ::= **(/ +) o (α x) o trans**
- Note the following properties of functional programs
  - » **NO explicit loops (or recursion)**
  - » **NO sequencing at a low level**
  - » **NO local variables**
- In addition, functional programs have the following properties
  - » **functions as input – in the above**
    - > **+ (plus), x (times)**
  - » **functions as output – not shown in the above**
    - > **In FP frequently write functions that produce a new function using other functions as input**

Evaluating (/ +) o (α x) o trans

◊ Apply the function to a single argument consisting of a list of the actual arguments.

innerProduct : < < a1, ... , an > , < b1, ... bn > >

◊ Work from right to left – **o** is function compostion

f o g : x ==> f (g (x))

◊ Thus we execute **trans** first – which means the transpose of a matrix – swap rows and columns

trans : < < a1, ... , an > , < b1, ... bn > >

==> < < a1, b1> , < a2, b2 > , ... , < an, bn > >

Evaluating (/ +) o (α x) o trans - 2

- Now execute (α ×)
  - (α ×) – read as apply times to all – means apply the function × (times) to all items in the argument list

    (α ×) : < < a1, b1> , < a2, b2 > , ... , < an, bn > >

    ==> < a1 × b1, a2 × b2, ... , an × bn >
- Now execute (/ +)
  - (/ +) – read as reduce using + – means put the function + (plus) between the arguments and apply from left to right

    (/ +) : < a1 × b1, a2 × b2, ... , an × bn >

    ==> a1 × b1 + a2 × b2 + ... + an × bn
- And we have the inner product

Backus notation (BN) and Lisp

- Data structures – the list
  - Lisp – (a b c d)
    BN – < a, b, c, d >
    - The list is a fundamental structure we will see it again in Prolog
- Selector functions
  - Lisp – car / first, cdr / rest
    BN – tail (equivalent to rest), 1, 2, 3, ... as needed or implemented, select item from the list
- Constructor functions
  - Lisp – cons
    BN – [f-1 , f-2 , ... , f-n] – each f-i operates on the input to produce a list as output

Backus notation (BN) and Lisp – 2

◊ Choice – if … then … else …

» **Lisp – (cond (p.1 s.1-1 s.1-2 ... s.1-p)**
(p.2 s.2-1 s.2-2 ... s.2-q)
...
(p.n s.n-1 s.n-2 ... s.n-r)
)

» **BN – p.1 --> function.1 ;** **If p.1 then function.1 else**
p.2 --> function.2 ;
... ;
p.n --> function.n

Backus notation (BN) and Lisp – 3

◊ Function application

» **Lisp –** (f x1 … xn) (apply f (x1 … xn)) (funcall f x1 … xn)
BN –   f : < x1, … xn >

◊ Mapping functions

» **Lisp –** (map f …)   (mapcar f …)   (maplist f …)
BN –   (α f)

◊ Other functions

	Reduction	Function Composition	Binding	Constant
» **Lisp –**	(reduce f x)	(comp f g)	(bu f k)	literal
BN –	(/ f)	f o g	(bu f k)	k

Inner Product – 1 argument versions

◊ Lisp recursive version

```
(defun innerProduct ( a-b-pair )
 ( cond ( ( null  ( car  a-b-pair ) ) 0 )
        ( t  ( +  ( *  ( caar  a-b-pair ) ( caadr  a-b-pair ) )
                 ( innerProduct  ( list  ( cdar  a-b-pair)
                                ( cdadr a-b-pair) ) ) ) )
))
```

Inner Product – 1 argument versions – 2

- Lisp functional version

```
( defun  innerProduct  ( a-b-pair )
   ( reduce   '+   ( mapcar   '* ( first   a-b-pair)
                                   ( second   a-b-pair ) ) ) )
```

mapcar does transpose due to having multiple arguments

- Backus notation

innerProduct ::= (/ +) o (α x) o trans

Matrix multiplication

◊ Lisp 2-argument version

```
( defun matProd ( a  b )
   ( mapcar ( bu  ‘prodRow  ( trans b ) )  a ) )

(defun prodRow ( bt  r )  ( mapcar ( bu  ‘ip  r ) bt ) )
```

> ip is the inner product (see previous slide)

◊ Backus notation version

matProd ::= (α α ip) o (α distl) o distr o [trans o 2 , 1]

Library of functions

- Depending upon the application area other functions are created.
  - **For example trans – transpose a matrix**
- Some are created using existing functionals
  - **For example innerProduct**

Library of functions – 2

- ◊ Others are created “outside” of the system for efficiency reasons
  - » **For example trans may be more efficient to implement outside of Lisp**
    - – **Although as compiler knowledge grows compilers produce more efficient code than “coding by hand”**
    - – **Machine speeds increase so many functions execute fast enough**
- ◊ The file **`functionals.lsp`** contains additional library functions. It can be downloaded from the www resources page for the course

Binding function – bu – 1

- Given a binary function it is often useful to bind the first parameter to a constant – creating a unary function
    - > **Also called currying after the mathematician Curry who developed the idea**
  - » **(bu '+ 3) – creates a unary "add 3" from the binary function "+"**

    (mapcar (bu '+ 3) '(1 2 3)) ==> (4 5 6)

  - » **Cons x before every item in a list**

    (mapcar (bu 'cons 'x) '(1 2 3)) ==> ((x.1) (x.2) (x.3))

  - » **Note that mapcar expects a function definition as the second argument, so we use bu to help construct the function**

Binding function – bu – 2

- We could define the function 3+

  ```
  ( define 3+  ( x )  ( + 3 x ) )
  ```

  - » **and use**

  ```
  (mapcar  ‘3+ ‘ (1 2 3))    ==>  (4 5 6)
  ```

  - » **but this adds to our name space**

- For use-once functions we can use lambda expressions

  ```
  (mapcar  #‘(lambda (x) (+ 3 x))  ‘(1 2 3))  ==>  (4 5 6)
  (mapcar (function
                ( lambda (x) (+ 3 x) ) )  ‘(1 2 3)) ==> (4 5 6)
  ```

Binding function – bu – 3

- The previous slide solutions are seen as being clumsy and more difficult to read compared to the following – bu has a clear meaning – with the above you have to reverse engineer to understand

  (mapcar (bu '+ 3) '(1 2 3)) ==> (4 5 6)

- Can define functions using bu

  (defun 3+ (y) (funcall (bu '+ 3) y))

  In such cases we would write

  (defun 3+ (y) (+ 3 y))

  We do not normally use bu to define named functions

Binding function – bu – 4

◊ BU is defined as follows

```
(defun bu (f x)
   #'(lambda (y) (funcall f x y))
)
```

> The long form

```
(defun bu (f x)
   (function (lambda (y) (funcall f x y)))
)
```

◊ BU uses a function as input and produces a function as output

Binding function – bu – 5

- How does Lisp represent the output of bu?
- In gcl (Gnu Common Lisp) you can see what takes place
  - » (bu ‘+ 3)

```
(LAMBDA-CLOSURE ( ( X 3) ( F + )) ()
 ( (BU BLOCK #<@001E8D10>) )
  (Y)
  (FUNCALL F X Y)
)
```

- We see the **parameter and body** from the definition of bu together with the bindings ((X 3) (F +))
- The closure adds the bindings to the environment so the body uses those bindings when it executes.

The Functional rev

◊ **rev** – reverse the order of the arguments of a binary function

```
(defun rev (f)
  #' (lambda (x y) (funcall f y x))
)
```

◊ Earlier we wrote

```
(mapcar (bu 'cons 'a) '(1 2 3)) ==> ((a.1) (a.2) (a.3))
```

◊ Suppose we want **((1.a) (2.a) (3.a))** then we write

```
(mapcar (bu (rev 'cons) 'a) '(1 2 3))
  ==> ((1.a) (2.a) (3.a))
```

Other Functionals in the notes – 1

- In `functionals.lsp` and the notes on functionals the following functionals are described
- **(comp unaryFunction1 unaryFunction2)**
  - > **Compose two unary functions**
- **(compl unaryFunction1 unaryFunction2 ... unaryFunctionN)**
  - > **Compose a list of unary functions**
- **(trans matrix)**
  - > **See slides on developing functional programs**

Other Functionals in the notes – 2

◊ (distl anItem theList)

> Distribute anItem to the left of items in theList

(distl ‘a ‘(1 2 3)) ==> ((a 1) (a 2) (a 3))

◊ (distr anItem theList)

> Distribute anItem to the right of items in theList

(distr ‘a ‘(1 2 3)) ==> ((1 a) (2 a) (3 a))