(Functional (Programming (in (Scheme))))

Jianguo Lu

Programming paradigms

- Functional
  - No assignment statement
  - No side effect
  - Use recursion
- Logic
- OOP
- AOP
- …

What is functional programming

- It is NOT what you do in in IMPERATIVE languages such C, C++, Java, C#....
- Functions are first class objects.
  - everything you can do with "data" can be done with functions themselves
    - such as passing a function to another function.
  - "higher order" functions, functions that operate on functions. E.g., MAP/REDUCE.
- Recursion is used as a primary control structure.
  - In some FP languages, no other "loop" construct exists.
- FP is declarative--worries about *what* is to be computed rather than *how* it is to be computed.

FP

- There is a focus on LISt Processing.
  - Lists are often used with recursion on sub-lists as a substitute for loops.
- "Pure" functional languages eschew side-effects.
  - No statements
    - No assignment
    - No state
    - No assigning first one, then another value to the same variable to track the program state.
  - one program is one expression (plus supporting definitions).

History

- Functional programming began in the late 1950s. There were many dialects, starting with LISP 1.5 (1960), through Scheme (1975) to Common LISP (1985).
  - Lisp(1958)→ scheme(1975)→common Lisp(1985)→scheme RRS(1998)
  - LISP = LISt Processor
  - Scheme is simpler than Lisp
    - Scheme specification is about 50 pages, compared to Common Lisp's 1300 page draft standard.
- Another category of functional programming languages are

  ML(1973) →Miranda(1982)→ Haskell(1987).
- FP IS alive
  - XSLT also has characteristics of functional programming
  - MAPREDUCE
  - Python
- The mathematical basis of many functional programming languages is λ-calculus. It allows expressions that have functions as values.

Run Scheme interpreter kawa

- kawa is an Scheme interpreter written in Java
- Alternatively, you can use other implementations.
  - DrScheme is a good one. It has detailed debugging information. http://www.plt-scheme.org/software/drscheme/

- Download kawa jar file
  - ftp://ftp.gnu.org/pub/gnu/kawa/kawa-1.9.90.jar

- Start Kawa by:

  C:\440>java -jar kawa-1.9.90.jar kawa.repl

- Try the following programs in Kawa

```
> "hello world"
hello world

> (+ 2 3)
5

>(exit)
```

- Also installed in our school unix system

  luna:~>kawa

  #|kawa:1|# (+ 2 3)

  5

Run scheme programs in a file

- Instead of writing everything at the Scheme prompt, you can:
  - write your function definitions and your global variable definitions (define...) in a file ("file_name")
  - at the Scheme prompt, load the file with: (load "file_name")
  - at the Scheme prompt call the desired functions
  - there is no "formal" main function

The Structure of a Scheme Program

- All programs and data are expressions
- Expressions can be atoms or lists
- Atom: number, string, identifier, character, boolean
  - E.g. `"hello world"`

    `hello world`

- List: sequence of expressions separated by spaces, between parentheses
  - E.g. `(+ 2 3)`
- Syntax:

expression	→	atom | list
atom	→	number | string | identifier | character | boolean
list	→	(expr_seq)
expr_seq	→	expression expr_seq | expression

Interacting with Scheme

- Interpreter: "read-eval-print" loop

  > 1

  1

  – Reads 1, evaluates it (1 evaluates to itself), then prints its value

  > (+ 2 3)

  5

  + => function +

  2 => 2

  3 => 3

  Applies function + on operands 2 and 3 => 5

Evaluation

- Constant atoms - evaluate to themselves

  42 - a number

  3.14 - another number

  "hello" - a string

  ‘a‘ - character 'a'

  #t - boolean value "true"

  > "hello world"

  hello world

Evaluation of identifiers

- Identifiers (symbols) - evaluate to the value bound to them

```
(define a 7)

> a
7

> (+ a 5)      =>
   12

> +
+

>  x1      =>
 error
```

Evaluate lists

- Lists - evaluate as "function calls":

  (function arg1 arg2 arg3 ...)

- First element must evaluate to a function
- Recursively evaluate each argument
- Apply the function on the evaluated arguments

```
> (- 7 1)
6

> (* (+ 2 3) (/ 6 2))
15

>
```

Operators

- Prefix notation
- Any number of arguments

```
> (+)
0

> (+ 2)
2

> (+ 2 3)
5

> (+ 2 3 4)
9

> (- 10 7 2)
1

> (/ 20 5 2)
2
```

Preventing Evaluation (quote)

- Evaluate the following:

  > (1 2 3) ;Error: attempt to apply non-procedure 1.

- Use the quote to prevent evaluation:

  > (quote (1 2 3))

  (1 2 3)

- Short-hand notation for quote:

  > '(1 2 3)

  (1 2 3)

Identifiers and quotes

```
(define a 7)

a                 => 7
'a     => a
 (+ 2 3)          => 5

'(+ 2 3)          =>
       (+ 2 3)

((+ 2 3))         =>
       Error
```

Forcing evaluation

```
(+ 1 2 3)              => 6
'(+ 1 2 3)             => (+ 1 2 3)
(eval  '(+ 1 2 3))     => 6
```

- eval evaluates its single argument
- eval is implicitly called by the interpreter to evaluate each expression entered:

  "read-eval-print" loop

List operations

- Scheme comes from LISP, LISt Processing
- List operations: cons, car, cdr, …
- cons – construct a list from head and tail

```
(cons  'a  '(b c d))                => (a b c d)
(cons  'a  '())                     => (a)
(cons  '(a b)   '(c d))   =>
                                    ((a b) c d)
(cons  'a  (cons 'b  '()))          =>
                                    (a b)
```

List operations

- car – returns first member of a list (head)

```
(car '(a b c d))                    => a
(car '(a))                                  => a
(car '((a b) c d))                          =>
                    (a b)
(car '(this (is no) more difficult))       =>
                     this
```

- cdr – returns the list without its first member (tail)

```
(cdr '(a b c d))                            => (b c d)
(cdr '(a b))                                => (b)
(cdr '(a))                                  =>
                    ( )
(cdr '(a (b c)))                    =>
                    (b c)  ?
                    ((b c))

(car (cdr (cdr '(a b c d))))        =>
                    c
(car (car '((a b) (c d)))) ➔
                    a
```

list operations

- null? – returns #t if the list is null ()

  #f otherwise

  (null? ()) => #t

- list – returns a list built from its arguments

```
(list 'a 'b 'c)  => (a b c)
(list 'a)                  => (a)
(list '(a b c)) =>
                  ((a b c))
(list '(a b) 'c)=>
                  ((a b) c)
(list '(a b) '(c d))       =>
                  ((a b) (c d))
(list '(+ 2 1) (+ 2 1))    =>
                  ((+ 2 1) 3)
```

List operations

- length – returns the length of a list

```
(length '(1 3 5 7))                    => 4
(length '((a b) c))                    =>
                    2
```

- reverse – returns the list reversed

```
(reverse '(1 3 5 7))      => (7 5 3 1)
(reverse '((a b) c))      =>
                    (c (a b))
```

- append – returns the concatenation of the lists received as arguments

```
(append '(1 3 5) '(7 9)) => (1 3 5 7 9)
(append '(a) '())                      => (a)
(append '(a b) '((c d) e))             =>
    (a b (c d) e)
```

Type Predicates

- Check the type of the argument and return #t or #f

(boolean? x) ; is x a boolean?

(char? x) ; is x a char?

- (char? #\a) => #t
- Characters are written using the notation #\<character>

(string? x) ; is x a string?

(string? “xyx”) => #t

(number? x) ; is x a number?

(number? 2) => #t

(list? x) ; is x a list?

(procedure? x) ; is x a procedure?

(procedure? car) => #t

Boolean Expression

```
(< 1 2)                          => #t
(>= 3 4)                         => #f
(= 4 4)                          => #t
(eq? 2 2)                        => #t
(eq? '(a b) '(a b))              => #f
(equal? 2 2)                => #t
(equal? '(a b) '(a b))           => #t    ; recursively equivalent
```

– "eq?" returns #t if its parameters represent the same data object in memory;

– equal? compares data structures such as lists, vectors and strings to determine if they have congruent structure and equivalent contents

```
(not (> 5 6))                    => #t
(and (< 3 4) (= 2 3))            => #f
(or  (< 3 4) (= 2 3))            => #t
```

Conditional Expressions

- if – has the form:

  (if <test_exp> <then_exp> <else_exp>)

  (if (< 5 6) 1 2)

  => 1

  (if (< 4 3) 1 2)

  => 2

  Recall the java expression (x<y)?1:2

- Anything other than #f is treated as true:

  (if 3 4 5)

  => 4

  (if '() 4 5)

  => 4

  It is not a typed-language

- if is a special form - evaluates its arguments only when needed (lazy evaluation):

  (if (= 3 4) 1 (2))

  => Error

  (if (= 3 3) 1 (2))

  => 1

Condition expression

- cond – has the form:

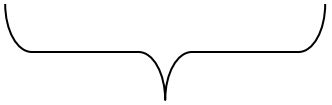
```
(cond
   (<test_exp1> <exp1> ...)
   (<test_exp2> <exp2> ...)
   ...
   (else <exp> ...))
```

```
(define n -5)
(cond ((< n 0) "negative")
      ((> n 0) "positive")
      (else "zero"))
=> "negative"
```

Recursive definition

- How do you THINK recursively?
- Example: define factorial

factorial (n) = 1 * 2 * 3 * ...(n-1) * n



factorial (n-1)

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=1 \quad \text{(the base case)} \\ n * \text{factorial}(n-1) & \text{otherwise (inductive step)} \end{cases}$$

Factorial example

```
(define (factorial n)
      (if (= n 1)
     1
     (* n (factorial (- n 1)))))

(factorial 4)
  => 24
```

Fibonacci example

- Fibonacci:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

- Implement in Scheme:

```
(define (fib n)
    (cond
      ((= n 0) 0)
      ((= n 1) 1)
      (else    (+ (fib (- n 1)) (fib (- n 2))))))
```

Length example

- Length of a list:

$$\text{len(lst)} = \begin{cases} 0 & \text{if list is empty} \\ 1 + \text{len (lst-without-first-element)} & \text{otherwise} \end{cases}$$

- Implement in Scheme:

```
(define (len lst)
  (if (null? lst)
    0
    (+ 1 (len (cdr lst)))))
```

```
|(length (a b c d))
| (length (b c d))
| |(length (c d))
| | (length (d))
| | |(length ())
| | |0
| | 1
| |2
| 3
|4
```

Sum example

- Sum of elements in a list of numbers:

  (sum '(1 2 3)) => 6

sum(lst) =

- 0 — if list is empty
- first-element + sum (lst-without-first-element) — otherwise

- Implement in Scheme:

```
(define (sum lst)
  (if (null? lst)
      0
      (+ (car lst) (sum (cdr lst)))))
```

Member example

- Check membership in a list:

```
(define (member? x lst)
  (cond
        ((null? lst) #f)
        ((equal? x (car lst)) #t)
        (else (member? x (cdr lst)))))
```

Recursion

- When recurring on a list lst ,
  - ask two questions about it: (null? lst) and else
  - make your recursive call on (cdr lst)

- When recurring on a number n,
  - ask two questions about it: (= n 0) and else
  - make your recursive call on (- n 1)

Local definition

- let - has a list of bindings and a body
  - – each binding associates a name to a value
  - – bindings are local (visible only in the body of let)
  - – let returns the last expression in the body

```
> (let ((a 2) (b 3))      ; list of bindings
    (+ a b))              ; body - expression  to evaluate
5


> (+ a b)
          => error
> a                                  => Error: variable a is not bound.
> b                                  => Error: variable b is not bound.
```

Notice the scope of *a* and *b* is within the body of let.

Let

- Factor out common sub-expressions:

  $f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$

  $a = 1+xy$

  $b = 1-y$

  $f(x,y) = xa^2 + yb + ab$

- Locally define the common sub-expressions:

```
(define (f x y)
    (let ((a (+ 1 (* x y)))
          (b (- 1 y)))
    (+ (* x a a) (* y b) (* a b))))

(f 1 2)                    => 4
```

‘let’ can define functions

- (let ((f +))
  (f 2 3)) =>

5

- (let ((f +) (x 2))
  (f x 3)) =>

5

- (let ((f +) (x 2) (y 3))
  (f x y)) =>

5

- The variables bound by let are visible only within the body of the let
  - (let ((+ *))
    (+ 2 3))

6

(+ 2 3) =>

5

Input and output

- read - returns the input from the keyboard

```
> (read)
234          ; user types this
234          ; the read function returns this

> (+  2 (read))
3            ; user input
5            ; result
```

- display - prints its single parameter to the screen

```
> (display "hello world")
hello world
> (define x 2 )
>(display x)
2
```

- newline - displays a new line

Input and output

- Define a function that asks for input:

```
(define (ask-them str)
   (display str)
   (read))

> (ask-them "How old are you? ")
How old are you? 22
22
```

Input and Output

- Define a function that asks for a number (if it's not a number it keeps asking):

```
(define (ask-number)
   (display "Enter a number: ")
   (let ((n (read)))
            (if (number? n)
                     n
                     (ask-number))))
```

```
> (ask-number)
Enter a number: a
Enter a number: (5 6)
Enter a number: "Why don't you like these?“
Enter a number: 7
7
```

Interactive factorial program

- An outer-level function to go with factorial, that reads the input, computes the factorial and displays the result:

```
(define (factorial-interactive)
   (display "Enter an integer: ")
   (let ((n (read)))
     (display "The factorial of ")
     (display n)
     (display " is ")
     (display (factorial n))
     (newline)))
```

```
> (factorial-interactive)
Enter an integer: 4
The factorial of 4 is 24
```

Higher order functions

- A function is called a higher-order function if it takes a function as a parameter, or returns a function as a result
- In Scheme, a function is a first-class object – it can be passed as an argument to another function, it can be returned as a result from another function, and it can be created dynamically

```
(let ((f +)) (f 2 3))
```

Examples of higher-order functions

- map
  - Takes as arguments a function and a sequence of lists
  - There must be as many lists as arguments of the function, and lists must have the same length
  - Applies the function on corresponding sets of elements from the lists
  - Returns all the results in a list

$$f\ (E_1\ \ E_2\ .....\ \ E_n)\ \rightarrow\ ((f\ E_1)\ \ (f\ E_2)\ \ .....\ \ (f\ E_n))$$

```
(define (square x) (* x x))
(map square '(1 2 3 4 5))                    => (1 4 9 16 25)
```

- (map abs '(1 -2 3 -4 5 -6)) =>
  (1 2 3 4 5 6)

- (map + '(1 2 3) '(4 5 6)) =>
  (5 7 9)

lambda expression

- You can also define the function in-place:

```
(map (lambda (x) (* 2 x)) '(1 2 3))          =>
   (2 4 6)
```

–
```
(map (lambda (x y) (* x y))
     '(1 2 3 4)
     '(8 7 6 5))    =>
(8 14 18 20)
```

- A simple version of map definition

```
(define (map F Lst)
  (if (null? Lst)
      Lst
      (cons (F (car Lst))
            (map F (cdr Lst)))
) )
```

Lambda expression

- A lambda expression (lambda abstraction) defines a function in Scheme.
- Informally, the syntax is:

```
(lambda (<parameters>) <body>)
```

- For example:

```
(define addOne (lambda (p) (+ p 1)))
```

it has the same effect as:

```
(define (addOne p) (+ p 1))
```

Logo for Racket, a functional language based on Scheme

Higher order function: reduce (also called fold)

- F: a binary operation, that is, a two-argument function.
- E0: a constant.
- We want to express the following transformation:

  (E1 E2 ..... En) => E0 F E1 F E2 F ..... F En

- in scheme notation:

  (E1 E2 ..... En) =>

  (F E0 (F E1 (F ..... (F En-1 Fn) .....)))

```
(define (reduce F E0 L)
  (if (null? L)
      E0
      (F (car L)
         (reduce F E0 (cdr L)))
) )
```

- **Example:**
  - **(reduce * 1 '(1 2 3 4))**
  - **➔ 1*1*2*3*4**

exercise: what is the result of this expression?

(reduce + 0 (map (lambda (x) 1) '(1 2 3)))

Different ways to define length function

```
(define (len lst)
  (if (null? lst)
     0
     (+ 1 (len (cdr lst)))))

(define len ( lambda (lst)
            (if (null? lst)
               0
               (+ 1 (len (cdr lst))))))

(define len (lambda (lst)
          (reduce + 0
                (map (lambda (x) 1) lst))))
```

Higher order function: apply

- apply
  - takes a function and a list
  - there must be as many elements in the list as arguments of the function
  - applies the function with the elements in the list as arguments

    (apply * '(5 6)) => 30

    (apply append '((a b) (c d))) => (a b c d)

- Comparison to eval:

  (eval <expression>) or (eval (<func> <arg1> <arg2>...))

  (apply <func> (<arg1> <arg2>...))

Higher order function: compose

- Compose takes two functions as parameters.
- It also returns a function

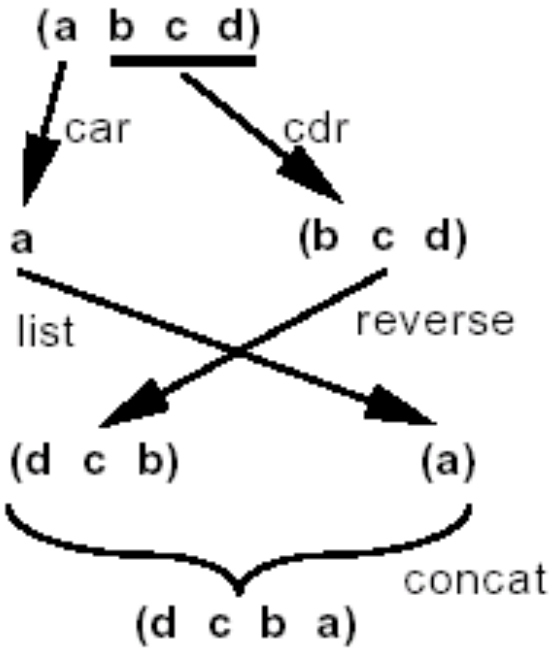
```
(define (compose f g)
   (lambda (x)
            (f (g x))))

((compose car cdr) '(1 2 3)) =>
2

((compose (lambda (x) (+ 1 x))
          (lambda (x) (* 2 x)))
  5 )        =>
11
```

Define reverse function

```
(define reverse ( lambda (L)
                    (if (null? L)
                      '( )
                      (append (reverse (cdr L)) (list (car L))) )))
```



Append example

```
> ( define ( append L1 L2 )     ; built-in!
    (if   ( null? L1 )
          L2
          ( cons ( car L1 )
                 ( append ( cdr L1 ) L2 ) )
  ) )


> ( append '( ab bc cd )
           '( de ef fg gh ) ) =>
(ab bc cd de ef fg gh)
```

Number list example

```
>
( define ( numberList? x )
  ( cond
    ( ( not ( list? x ) ) #f )
    ( ( null? x ) #t )
    ( ( not ( number? ( car x ) ) ) #f )
    ( else ( numberList? ( cdr x ) ) )
) )

> ( numberList? ' ( 1 2 3 4 ) )
#t
> ( numberList? ' ( 1 2 3 bad 4 ) )
#f
```

Insertion sort example

```
(define (insert x l)
        ( if (null? l)
          (list x)
          (if (<= x (car l))
            (cons x l)
            (cons (car l) (insert x (cdr l))))))
```

6 5 3 1 8 7 2 4

```
(define (isort l)
        (if (null? l)
            ()
            (insert (car l) (isort (cdr l)))))
```

Insertion sort in an imperative language

```
void    isort    (int[ ] A) {
    int j;
    for (int i = 1; i < A.length; i++) {
            int a = A[i];
            for (j = i -1; j >=0 && A[j] > a; j- -)
                                A[j + 1] = A[j];
            A[j + 1] = a;
     }
}
```

DFA simulation

- Start state: q0
- Transitions

  ((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)

  ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2)

- Final state: q0
- DFA

  '(q0

  (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)

  ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))

  (q0))

Define move function

```
(define move
 (lambda (dfa symbol)
  (let ((curstate (car dfa)) (trans (cadr dfa))
      (finals (caddr dfa)))
   (list
    (if (eq? curstate 'error)                    ;get the next state or ‘error
      'error
      (let ((pair (assoc (list curstate symbol)  trans)))
                                  ; pair is the next state, say ((q0,0),  q2), or false when not found
       (if pair (cadr pair) 'error)))                    ;  boolean cond is true if not false
    trans
    finals))))
```

```
'(q0
 (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)
((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
(q0))
```

- cadr: car and cdr. Second element
- caddr: third element
- assoc: search an element in an association list

assoc function

- assoc: search an association list (table)
  - An association list is a list of lists.
  - Each sublist represents an association between a key and a list of values.
- Examples
  - Scheme>(assoc 'julie '((paul august 22) (julie feb 9) (veronique march 28)))
    - (julie feb 9)
  - Scheme>(assoc 'key2 '((key1 val1) (key2 val2) (key0 val0)))
    - (key2 val2)
  - Scheme>(assoc '(feb 9) '(((aug 1) maggie phil) ((feb 9) jim heloise) ((jan 6) declan)))
  - ((feb 9) jim heloise)
  - (assoc (list curstate symbol) trans)
  - (assoc '(q0 0) trans) → ((q0,0), q2)

Simulate function

```
(define simulate
 (lambda (dfa input)
  (cons (car dfa)
   (if (null? input)
      (if (infinal? dfa) '(accept) '(reject))
      (simulate (move dfa (car input)) (cdr input))))))
```

- Run the program

```
(simulate
 '(q0
  (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)
   ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
  (q0))
'(0 1 0 0 1 0))
```

- Output: the trace of the states and ‘accept’ or ‘reject’

```
(q0 q2 q3 q1 q3 q2 q0 accept)
```

infinal state function

```
(define infinal?
    (lambda (dfa)
      (memq (car dfa) (caddr dfa))))
```

- dfa

```
'(q0
  (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)
   ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
  (q0))
```

- memq: return the first sublist of list whose car is obj. If obj does not occur in list, then #f (not the empty list) is returned
  - memq 'a '(a b c)) ==> (a b c)
  - (memq 'b '(a b c)) ==> (b c)
  - (memq 'a '(b c d)) ==> #f

Assignment

- a calculator for simple arithmetic expressions
  - when typing (calculator '(1 + 2)), scheme interpreter will print 3.
- The calculator will accept expression of any length, and the expression associates to the right.
  - In (calculator '(1 + 1 - 2 + 3)), the expression is interpreted as (1+ (1 - (2+3))), hence the value is -3.
- One bonus mark for left association

  1+1-2+3 is evaluated as 3

Interpreter and compiler

- Interpreter
  - Program is executed directly. No translated code is generated.
  - Slow
  - For small programs
- Compiler
  - Program is translated into code that is closer to machine (intermediate code, or assembly code, or machine code)
  - Faster
  - For bigger programs
  - hybrid implementation for Java and C#

Functional programming

- Pythorn
- XSLT
- MAPREDUCE

More about assignment

(define expr '(+ 1 2))

(eval expr)

(define operator (car expr))

operator

(operator 2 3)

- +
- procedure application: expected procedure, given: +

(define operator (eval (car expr)))

operator

(operator 2 3)

- #<procedure:+>
- 5