School of Electrical Engineering and Computing

# SEENG2200/6220 Programming Languages and Paradigms

Topic 10
Functional Programming
– Part 2

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au



## Topic 10 Overview

Thinking Functional
Recursion
Lazy Evaluation





## Thinking Functional

One of the greatest challenges which faces students new to functional programming is…
- Learning to match parentheses ☺

Another challenge is learning to “think” in a functional style





## Thinking Functional

Start with a broad statement of the problem
- This becomes the main function
- Parameters are only tentative at this point

Break the problem into sub-problems
- These are the helper functions the main function will call
- Again, parameters are tentative

**Only impose sequential evaluation where absolutely necessary**





## Thinking Functional

Recursively break these problems into smaller and smaller problems until each problem is a simple, unambiguous operation
- Calls to built-in functions

From the bottom up, decide what parameters each function needs to fulfill its responsibilities
- Some parameters will be the return values of other functions





## Thinking Functional

Example – BubbleSort…

```
BubbleSort(nums)
(
  ; do bubble-sweep
  ; until sorted
)
```

- Do-until is not a functional construct

7

## Thinking Functional

```
BubbleSort(nums)
(
  ; BubbleSweep(nums)
  ; IF (Sorted? nums)
    ; nums
    ; BubbleSort(nums)
)
```

- Uses recursion.
- Not purely functional (but close enough)



8

## Thinking Functional

```
Sorted?(nums)
(
  ; if length(nums) < 2 ⇒ #t
  ; else if nums[0] > nums[1] ⇒ #f
  ; else ⇒ Sorted?(cdr(nums))
)
```



9

## Thinking Functional

```
BubbleSweep(nums)
(
  ; if length(nums) < 2 ⇒ nums
  ; else
    ; if nums[0] > nums[1]
      ; swap nums[0] ⇔ nums[1]
    ; ⇒ nums[0] . BubbleSweep(cdr(nums))
)
```

- The swap/sweep combination relies on sequential modification of the nums list – not very functional



10

## Thinking Functional

```
BubbleSweep(nums)
(
  ; if length(nums) = 1 ⇒ nums
  ; else if nums[0] > nums[1]
    ; ⇒ nums[1] .
      ; BubbleSweep(nums[0].cddr(nums))
  ; else ⇒ nums[0] .
      ; BubbleSweep(cdr(nums))
)
```

- This version swaps as it builds the argument lists – very functional ☺



11

## Thinking Functional

In this case, passing the list of numbers provides enough information for each sub-problem – so the parameters are good as they are

- But note the way Scheme handles zero-or-more arguments in recursive function calls

Now, turn it into Scheme code…



12

## Thinking Functional

```
(DEFINE (Sorted? nums)
  ; (DISPLAY nums) (NEWLINE)
  (COND
    ( (< (LENGTH nums) 2) #t )
    ( (> (CAR nums) (CAR (CDR nums))) #f )
    ( ELSE (Sorted? (CDR nums)) )
  )
)
```

## Thinking Functional

```
(DEFINE (BubbleSweep nums)
  ; (DISPLAY nums) (NEWLINE)
  (COND
    ( (< (LENGTH nums) 2) nums )
    ( (> (CAR nums) (CAR (CDR nums)))
      (CONS
        (CAR (CDR nums))
        (BubbleSweep
          (CONS (CAR nums) (CDR (CDR nums)))
    )))
    ( ELSE
      (CONS (CAR nums) (BubbleSweep (CDR nums)))
)))
```





## Thinking Functional

```
(DEFINE BubbleSort (LAMBDA nums
  (LET ( (sweep (BubbleSweep nums)) )
    (DISPLAY sweep) (NEWLINE)
    (IF (Sorted? sweep)
      sweep
      (APPLY BubbleSort sweep)
  ))
))
```

- `nums` is a list of numbers (the argument list)
- `(BubbleSort nums)` sorts one item – a nested list
- `APPLY` applies `BubbleSort` to the original list





## Thinking Functional

Pure Functional

- A function with no *side effects*
  - Doesn't change any state external to itself
- If the result changes, the only thing that affects this change is the function's input.
  - Doesn't rely on any state external to itself.
- Simple to run in parallel
  - Running a function over every value on a list can be done in parallel.
  - i.e. ***sin(x)*** where ***x*** takes on the value of each element of a ***List<Integer>*** or, in Scheme (assuming ***pi*** is defined as 3.14......) (sin 1 pi 3 3 pi 5 6 7).
    - Because ***sin(pi)*** is always ***0***, it does not matter if it is evaluated before, after, or at the same time as ***sin(5)***.





## Thinking Functional

Which of the following are pure functions?

```
define func1 ()
    return 1

define func2 ()
    return today's date

define func3 (x, y, z)
    return x + y + z

define func4 (x, y, z)
    return 2

define func5 ()
    return random number

define func6 (x)
    if x < 8
        return func1()
    return func2()
```





## Function Call Tree

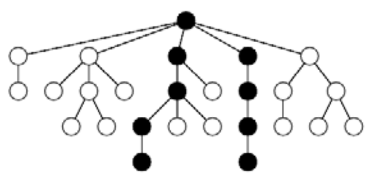
The sequence of function calls during the execution of a program can be represented by a *tree* data structure

- The main function of the program is the *root* of the tree
- If function A calls function B then there is a node labelled B which is a child of the node labelled A
- The height of the tree correlates with the size of the stack needed to execute the program





## Function Call Tree



Nodes =
- functions

Edges =
- pushed args
- popped results

## Recursion

*Recursion* is when a function calls itself

- Either directly or indirectly
- There must be a *terminating condition* else the function will **never be evaluated**!

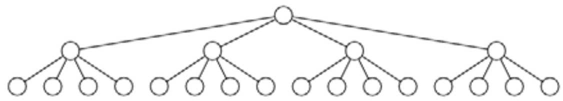
*Mutual recursion* is when there is a *cycle* of calls among a set of (more than one) functions.

- For example: A(x) ← B(x+1) and B(y) ← A(y+1)





## Recursion



If each call to the function calls itself *m* times and terminates after *n* recursive calls, then…

- the total number of calls is $O(m^n)$
- the maximum stack size is $O(n)$

If *m* = 1 it is called *linear recursion*





## Recursion

Recursion can be used to simulate imperative-style iteration in functional programming languages

As traditional computer architectures do not handle function calls well, the number of recursive calls is an important way of measuring the efficiency of a program





## Recursion

The *recursive width* of a function A is the number of times calls to A resulting (directly or indirectly) from one call to A

- A ← A() + A() + A() has recursive width 3
- A ← B() + C(); B ← A(); C ← A() has recursive widths 2 on A, 1 on B and 1 on C
- Recursive width can be calculated from the function definition without knowing the run-time arguments





## Recursion

The *recursive height* of a function A is the number of times A is called on the longest path from the root to a leaf of the function call tree

- Given A(x) ← if (x > 0) x+A(x-1) else 0, the recursive height of A for A(4) is 5
- Given A(x) ← if (x > 0) B(x-1) else 0 and
  B(x) ← if (x > 0) A(x-1) else 1,
  the recursive height of A for A(4) is 3, whereas
  the recursive height of B for A(4) is 2 –
  the total recursive height is 5
- Recursive height can only be calculated when the run-time arguments are known





## Tail Recursion

While recursive function calls are (in general) much less efficient than procedural iteration structures, they can be more readable

Functional language compilers can convert *some* recursive structures into more efficient forms

In particular, compilers are good at optimising *tail recursion*

## Tail Recursion

25

*Tail recursion* is when a recursive function either:
- returns a value, without recursion, or
- the last action of the function is to call itself with new arguments.

Tail-recursive functions can be evaluated by transforming the recursive function calls into a loop and storing a single progressive result
- **Constant stack space!**
- Good functional “compilers” do this automatically



## Tail Recursion

26

Functional

```
GCD(u,v) ←
  if v=0 then u
  if v≠0 then GCD(v, u mod v)
```

…becomes imperative…

```
while v≠0 {
  t1 ← v; t2 ← u mod v;
  u ← t1; v ← t2;
}
return u
```



## Tail Recursion

27

Many non-tail-recursive functions can be made tail-recursive by adding *accumulating parameters*

Non-tail-recursive

```
( DEFINE ( factorial n )
  ( COND
    ( ( = n 0 ) 1 )
    ( ( > n 0 )
      ( * n ( factorial ( - n 1 ) ) )
    )
) )
```



## Tail Recursion

28

Tail-recursive

```
( DEFINE ( trfactorial n prod )
  ( COND
    ( ( = n 0 ) prod )
    ( ( > n 0 )
      ( trfactorial
        ( - n 1 ) ( * n prod )
    ) )
) )
(factorial n) ≡ (trfactorial n 1)
```



## Tail Recursion

29

(`factorial 5`) results in the following call trace:

```
5 * (factorial 4)
5 * 4 * (factorial 3)
5 * 4 * 3 * (factorial 2)
5 * 4 * 3 * 2 * (factorial 1)
5 * 4 * 3 * 2 * 1 * (factorial 0)
5 * 4 * 3 * 2 * 1 * 1

The sum is 120.
```



## Tail Recursion

30

`(trfactorial 5 1)` results in the following call trace:

```
trfactorial 5 1
trfactorial 4 5
trfactorial 3 20
trfactorial 2 60
trfactorial 1 120
trfactorial 0 120

The sum is 120.
```

## Lazy Evaluation

Lazy evaluation is the delaying of the evaluation of part of a program until it is actually needed

- A bit like a function call – the code is there but isn't executed until it is called on
- A bit like short-circuit logic – the code is there but isn't called unless needed
- But more so ... the code may not even be compiled/interpreted until its value is needed
- **The code may be evaluated in the context of where it actually needs to be evaluated, rather than being evaluated where it is defined!**





## Lazy Evaluation

Arguments passed to a function under lazy evaluation can be seen as a *promise* to provide the value *when needed*

- When the called function needs each argument the promise is *forced* and the argument evaluated
- If the argument is never actually needed (such as inside a selection function) then it is never evaluated

Lazy evaluation is harder to implement but can be more efficient





## Lazy Evaluation

Scheme supports lazy evaluation

- **`(delay`** ***`expr`*****`)`** – returns a promise to evaluate the expression
- **`(force`** ***`promise`*****`)`** – evaluates the promise

Scheme uses *memorization* – **a promise is only evaluated once** and the resulting value is remembered (sometimes called *pass-by-need*)

```
( define p ( delay ( + 1 x ) ) )
( define x 1 ) ( force p ) ⇒ 2
( define x 21 ) ( force p ) ⇒ 2
```





## Lazy Evaluation

Many scheme interpreters also support…

**`(eval`** ***`obj`*****`)`** – treats the Scheme object as if it is Scheme code and executes it in the current context!

- Example: **`(eval '(+ 1 2))`**
- Common, but not standard
- **Dangerous** (especially if ***obj*** is influenced by user input)





## References

R. W. Sebesta, "Concepts of Programming Languages", 9th Edition, Addison-Wesley, 2010 (Chapters 10 and 15)

"Revised[5] Report on the Algorithmic Language Scheme" . http://www.schemers.org/Documents/ Standards/R5RS/