School of Electrical Engineering and Computing

SENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS

(S1, 2020)

Functional Programming II

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Outline

- 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*

Thinking Functional

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

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

Thinking Functional

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

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

Thinking Functional

```
BubbleSweep(nums)
(
  ; if length(nums) < 2 ⇒ 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 ☺

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…

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

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.*
  - *E.g., 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).*

Pure Functional

- Which of the following are pure functions?

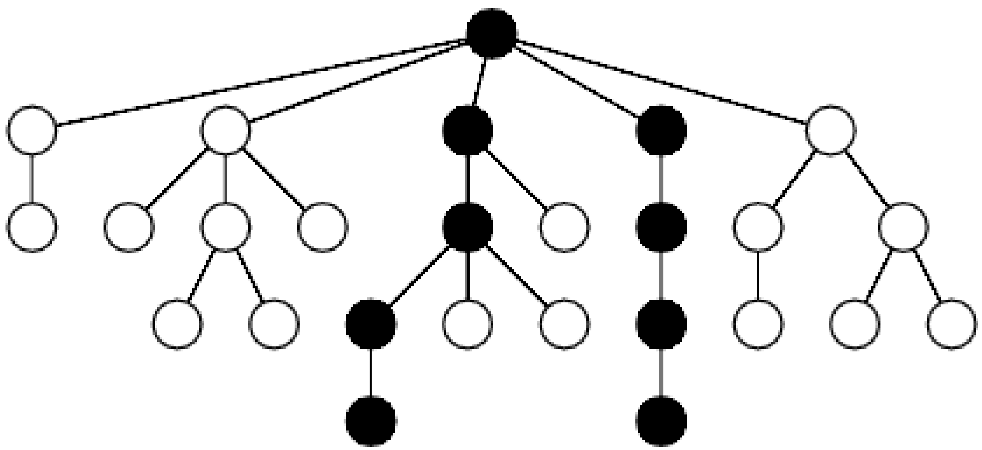
```
define func1 ()
    return 1

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



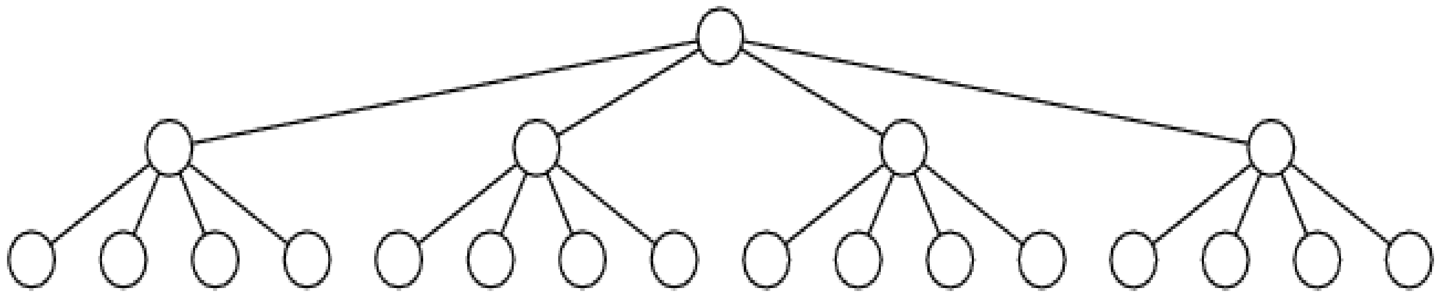
Nodes: functions

Edges: pushed args and/or popped results

Recursion

- *Recursion* is when a function calls itself
  - *Either directly or indirectly*
  - *There must be a terminating condition otherwise 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:* $A(x) \leftarrow B(x+1)$ *and* $B(y) \leftarrow A(y+1)$

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 <u>*recursive height*</u> 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) $\leftarrow$ if (x > 0) x+A(x-1) else 0, the recursive height of A for A(4) is 5*
  - *Given A(x) $\leftarrow$ if (x > 0) B(x-1) else 0 and*
    B(x) $\leftarrow$ if (x > 0) A(x-1) else 1,
    - the recursive height of A for A(4) is 3,
    - 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

- *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

- 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

- 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

- `(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

- Tail-recursive

```
( define ( trfactorial n prod )
  ( cond
    ( ( = n 0 ) prod )
    ( ( > n 0 )
      ( trfactorial
        ( - n 1 ) ( * n prod )
    ) )
) )
(factorial n) ≡ (trfactorial n 1)
```

Tail Recursion

- `(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
```

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…
  - *Common, but not standard*
- (**eval** *expr*) – treats the Scheme object as if it is Scheme code and executes it in the current context! For example,
  - (**eval** '(+ 1 2)) ⇒ 3
- **eval** supports dynamically constructed expression. obj can be modified like an argument.
  - ***Dangerous*** *(especially if expr is influenced by user input)*

```
( define ( eval-formula formula )
    ( eval `(let ((x 3)) ,formula) ) )
(eval-formula (* x x)) ⇒ 9
(eval-formula (+ x x)) ⇒ 6
```

References

- R. W. Sebesta, “Concepts of Programming Languages”, 9th Edn, Addison-Wesley, 2010 (Chapter 15) (also Edn.10)
- R. K. Dybvig, “The Scheme Programming Language”, 3rd Edition, MIT Press, 2003. http://www.scheme.com/tspl3/