CSC 461 Quiz #3

Study Guide

Mary Moore

Functional Programming Concepts

- paradigms:
  - imperative
    - normal stuff
    - java, c++, c, c#, python
    - tell the computer how to do stuff
    - usually has pass by reference or similar
    - has void functions
  - functional/declarative
    - math based
    - written as a string of output plugged in as input to functions
    - R, F#, LISP, scala, prolog, excel
    - tell the computer what you want
    - very close to traditional algebra
    - often exclusively pass by value (may even restrict to wrappers)
    - no void functions (in pure form)
- why functional?
  - better designed for data mining
  - map
    - apply a function to all elements in a collection
  - filter
    - pull a subset out of a collection based on some criteria
  - reduce
    - combine elements in a collection
  - these three functions are the primary concerns of functional languages
  - no side effects
    - easier to debug
    - easier to test
    - easier to convert to concurrency
    - limits the possibility of uninitialized values
  - data is not mutable
- basics of functional programming
  - similarities to imperative
    - binding and scope still apply
    - naming still applies
    - type systems still apply
    - functions still apply (even more so!)
  - differences with imperative
    - prefers immutable data
    - prefers recursion over iteration
    - uses even MORE polymorphism
    - almost always has first class functions

- what functional programming is not great at…
  - multi-level initialization
    - due to everything done immediately
  - random access
    - the recursion inhibits this
  - memory use
    - no mutable data means no storage reuse
- monads
  - some things *still* require state
    - random generators, I/O
    - could return the state and send it to every call… but who wants to do that!
  - monads are the solution!!
    - containers
    - handle the "return and reuse state in the call" for you by returning the same type as the container
    - don't need to worry about how they work, just that they are a wrapper for a type of function composition
- functional languages *can* have OOP!
  - even more than imperative in some ways
  - prefer a lot more struct based formats tho with lots of operator overloading
- trickiest parts about functional
  - no side effects ⇒ bundle your information
  - no loops in pure form ⇒ convert iteration to tail recursion

Questions:

1. **Assume we have the following list, how do we negate the list, and then sum the values evenly divisible by 5?**
   List = [1, 5, -4, 10, -15, 12]
   neg = list.map(-x)
   filtered = neg.filter(x%5 == 0)
   sum = filtered.reduce(x+y)

2.
3.

Scala Code

- generation:
  - high level
- paradigm:
  - functional
- location:
  - big data
  - high small message throughput
  - highly concurrent applications (ex. twitter)

- philosophy:
    - scala ⇒ scalable language
    - extensibility is critical
    - encourages no side effects
    - encourages immutable data
    - allows functions to be used interchangeably
    - do NOT rewrite java code!
    - flexible objects
    - write once, run anywhere (w/same limitations as java)
    - meant for experienced programmers
        - python → teaching languages
        - java → fix problems w/c++
        - scala → let the programmer do what they want
- designed by Martin Odersky
    - about a decade old
- runs on JVM
    - scala & java are closely related
    - kotlin is based on scala's syntax
    - c# does smth like this as well with managed code and c for speed
- green flags
    - more or less, you can use java or scala (whichever is best for the task at hand) in the *same* project
        - means this shares many of the same strengths & weaknesses of java…
    - many built in libraries
    - better built for recursion
    - incredibly easy concurrency support
    - reusable & interchangeable functions is a primary concern
- red flags
    - slower than true compiled languages
    - can reverse compilation
    - security issues
- beige flags
    - functional programming feels *weird* when you first start
    - immutable data is preferred
        - a.k.a. prefers pure functions (no side effects and no mutations)
    - substantial type inference typing
        - regularly do not name the type, but the type is static!
    - garbage collection
        - the language handles clean dynamic memory allocations for you 🙂
        - tends to hit at the worst time 🙁
- how scala works
    - scala worksheet
        - should be used for 'scratch' purposes
        - list the evaluation at each step with "evaluation worksheet"

        - ends file name with .sc
    - scala console
        - also should be used for 'scratch purposes'
        - type in expressions, ending in enter
    - scala file
        - ends file names with .scala
- program execution
    - def
        - keyword to define a function (like python)
    - args
        - argc array
        - scala follows UML stype variables
    - unit
        - make this a 'void' function declaration
    - object
        - contents are static
- main v.s. app
    - to just *have* main there for you: object xyz extends App
    - app is a trait (interface) in scala and encapsulates main
- scala structure
    - file name and object/class MUST match
    - the package name is the list of folders, separated by a .
    - you MUST use a unique package name when submitting
- modules
    - to import all ⇒ import name.name.lastGroup.*become ??????
    - to import part ⇒ import name.name.lastGroup.{item1, item2, …}
    - I/O import one ⇒ import scala.io.StdIn

Questions:

1. **Scala is built for concurrency. Why are "no side effects" helpful for this?**
   the data is more atomic, as now you only update on a return

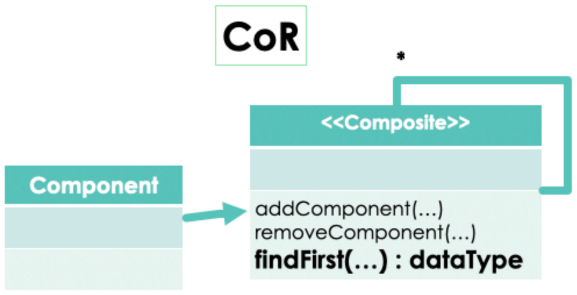
2.

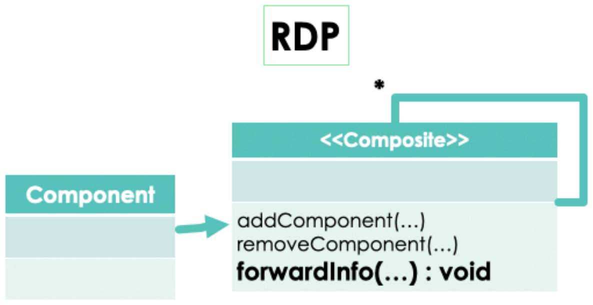
Composite & Techniques

- one object is composed of smaller pieces
- it is a subset of "has a" relationships in that this is only applied to objects that are incomplete without their subparts and the subparts disappear if the "root" object is deleted
- all subcomponents are derived from the same parent (or interface)!
- both RDP & COR require the structural format of composite to work properly
    - difference lies in how we 'stop'
        - RDP → stops only after *all* subcomponents are hit
        - COR → stops *early* if a subcomponent marks the task as 'handled'

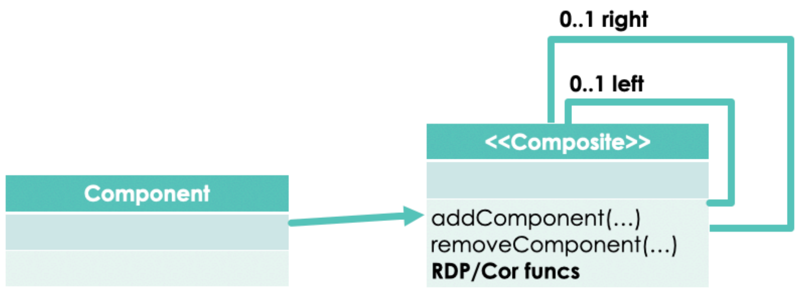
- when should these be used?
    - you have a ‘deeply’ nested structure
        - deeply ⇒ 3+ or a list of children
    - a function at the top level also affects its subcomponents (or vice versa) ⇒ RDP
    - OR you are trying to find something to handle a task, and you don’t care who ⇒ COR
- structure
    - root class that has an associated link to itself. the needed functions are also placed in the root class. then the concrete classes are derived and you make a ‘graph’ out of them

    1. need a root class that links to itself (abstract or interface)
    2. need to be able to add & remove components
    3. need a concrete class
    4. add a function we use to travel down the collection
        a. if it is RDP, there is normally NOT a return
        b. if it is COP, there normally is a return (bool is common as it indicates ‘was handled’)

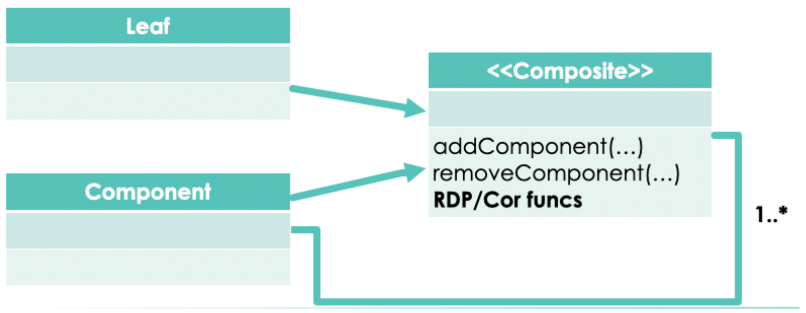




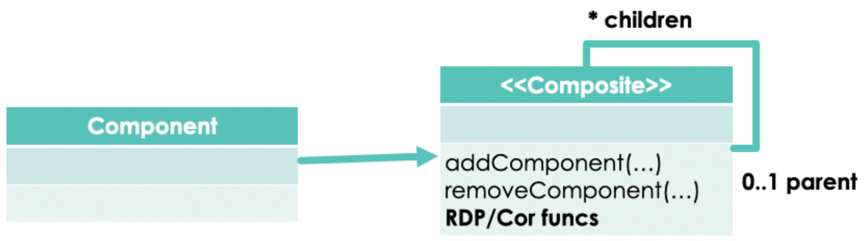
- variations on structure
    - tree



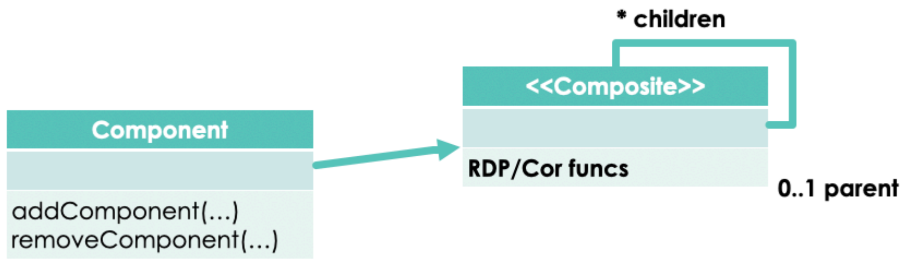
    - composite tree with special leaf node to guarantee there is no link to another composite component



    - chain that allows traversals in both directions



    - type safety



- …

Recursive Descent Parsing

- accessing the entire group object requires a technique called "recursive descent parsing"
- why?
    - we naturally regularly have something that owns another item of the same type
        - graphs :)
    - guarantees the same functions throughout
        - very common in simulators, graphics, monitors, models, animations, web design, etc.
- recursive descent parser (RDP)
    - most common composite 'behavior'
    - recursively forwards information to all components that may care about it
    - just a walkthrough of a generic tree, just adds a task beyond print lol
- examples
    - XML, JSON, HTML loading, updateAll, moveAll

Chain of Responsibility

- closely related to RDP
- purpose is to forward a command until someone handles it
- classical format: linked list & stop when someone says they handled it
- chain can instead by a tree which is common in GUIs
  - makes an early breaking search algorithm work on data with varying types
- examples
  - first internet connection option found, first exception handler that handles my current exception, first component that accepts the 'k' key

Questions:

1. **When is inheritance only okay, rather than RDP?**
   RDP is specifically for treating an object with subcomponents as though it was a single massive class. if the tasks are solely for only **one** element rather than as a **group**, basic inheritance is fine
2.

Control Flow

- statement
  - x.foo()
- expression
  - int y = x.foo()
    y = x•f(...)
- 8 types of control:
  - sequencing
    - statements are to be executed (or expressions evaluated) in a certain specified order – usually the order in which they appear in the program text
    - x = 1
    - y = x+2
  - selection
    - depending on some run-time condition, a *choice* is to be made among two or more statements or expressions
    - if
  - iterators
    - a given fragment of code is to be executed repeatedly, either a certain number of times, or until a certain run-time condition is true
    - for, for each, loop
  - methods/functions
    - a potentially complex collection of control constructs (a *subroutine*) is encapsulated in a way that allows it to be treated as a single unit, usually subject to parameterization
  - recursion

        - an expression is defined in terms of (simpler versions of) itself, either directly or indirectly; the computational model requires a stack on which to save information about partially evaluated instances of the expression
    - concurrency
        - two or more program fragments are to be executed/evaluated "at the same time," either in parallel on separate processors, or interleaved on a single processor in a way that achieves the same effect
    - exceptions
        - a program fragment is executed optimistically, on the assumption that some expected condition will be true
            - if that condition turns out to be false, execution branches to a handler that executes in place of the remainder of the protected fragment (in the case of exception han- dling), or in place of the *entire* protected fragment (in the case of speculation)
        - for speculation, the language implementation must be able to undo, or "roll back," any visible effects of the protected code
    - non-determinacy
        - the ordering or choice among statements or expressions is deliberately left unspecified, implying that any alternative will lead to correct results
            - some languages require the choice to be random, or fair, in some formal sense of the word
        - randomness??
    - every single one of these map back to goto behind the scenes
        - goto makes multiple entries & multiple exits easy, but creates spaghetti code…

Functions

- calling sequence → the code executed by the caller immediately before and after a subroutine call
    - Maintenance of the subroutine call stack is the responsibility of the calling sequence
- prologue → code executed at the beginning, setup
    - setx()
      return x
    - formal argument:
      setx(u)
      x = u
    - actual argument:
      setx(2)
- epilogue → code executed at the end, cleanup
- tasks that must be accomplished on the way into a subroutine:
    - passing parameters
    - saving the return address
    - changing the program counter
    - changing the stack pointer to allocate space

    - saving registers (including the frame pointer) that contain values that may be overwritten by the callee but are still *live* (potentially needed) in the caller
    - changing the frame pointer to refer to the new frame
    - executing initialization code for any objects in the new frame that require it
- tasks that must be accomplished on the way out:
    - passing return parameters or function values
    - executing finalization code for any local objects that require it
    - deallocating the stack frame (restoring the stack pointer)
    - restoring other saved registers (including the frame pointer)
    - restoring the program counter
- some of these tasks (e.g., passing parameters) must be performed by the caller, because they differ from call to call
    - most of the tasks, however, can be performed either by the caller or the callee
    - we will save space if the callee does as much work as possible
    -
-
- pass by reference
- pass by value
- stack layout
    - each routine, as it is called, is given a new *stack frame*, or *activation record*, at the top of the stack
    - This frame may contain arguments and/or return values, bookkeeping information (including the return address and saved regis- ters), local variables, and/or temporaries
    - When a subroutine returns, its frame is popped from the stack
    - At any given time, the *stack pointer* register contains the address of either the last used location at the top of the stack or the first unused location, depending on convention
    - The *frame pointer* register contains an address within the fram
- function links
    - dynamic → just go right down the stack
    - static → go to parent

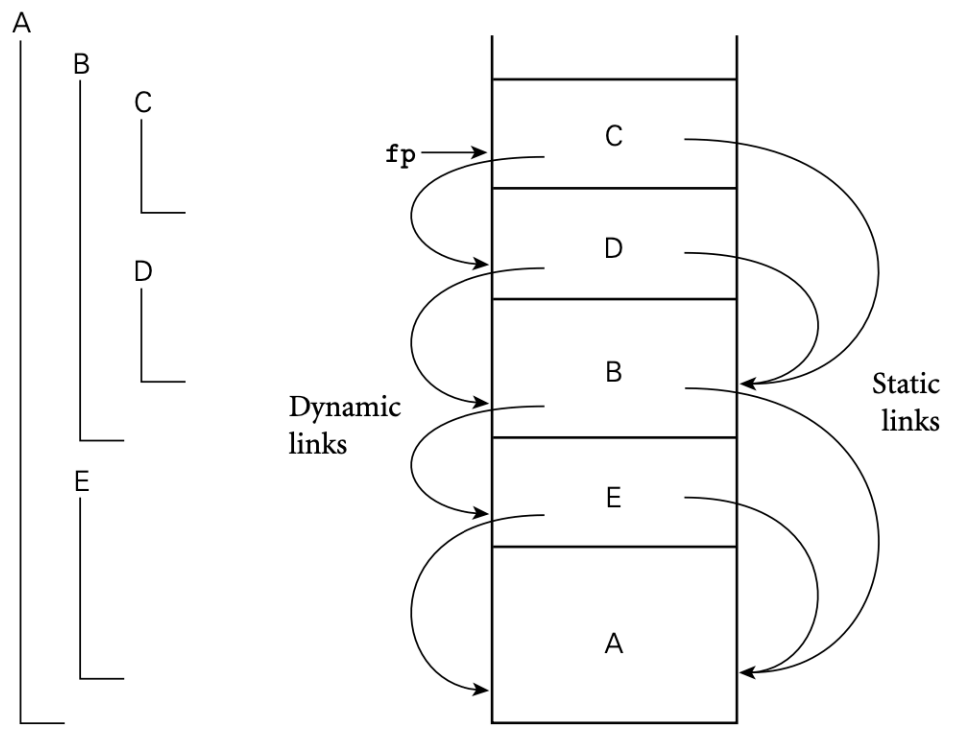


Figure 9.1 **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

•