COEN 171 Final Study Guide

Chapter 9 Subprograms

9.1 Introduction.
- Data and process abstraction
- Subprograms for process abstraction
- Reuse of program parts - modularity

9.2 Fundamentals of Subprograms

9.2.1 General
- Characteristics
  - Subprograms have a single entry point
  - Calling program suspended - only one runs at a time
  - Returns to caller after execution completes

9.2.2 Basic Definitions

```
def foo(parameters)
```

9.2.3 Parameters
- Formal Parameters in header
- Pass By:
  - Value
    - Copy of formal parameter, not actual
    - Fast
    - But can't modify the caller's variable
  - Reference
    - Copy the address of the variable
    - Can modify the caller's variable
    - Slower, issues with aliasing

9.2.4 Procedures and Functions
- Procedures - produce results in two ways
  - If there are variables that are not formal parameters but are still visible
  - Formal paramters that allow transfer of data
  - Define new statements - single call
- Functions
  - Resemble procedures
  - semantically modeled after mathematical statements
  - some languages allow parameter overloading - multiple definitions adding extra parameters

```
//defined by
float power(float base, float exp){...}
//called by
result = 3.4 * power(10.0, x);
```

9.4 Local Referencing environments

9.4.1 Local Variables
- Scope is the body of the subprogram - typicaly within the corresponding "{}"

9.5 Parameter Passing Methods

9.5.2.1 Pass By Value
- Value of the actual parameter is used to initialize the corresponding formal parameter
- Copy the value
- Fast, but requires more memory
- If you don't want to modify the caller's variable

9.5.2.2 Pass by Result

- No value transmitted to subprogram
- Corresponding formal parameter acts like a local variable
- Value transmitted back to caller just before return

9.5.2.3 Pass by Value result (Pass-by-copy)
- The value of the actual parameter is used to initialize the corresponding formal parameter, which then a
- In fact, pass-by-value-result formal parameters must have local storage associated with the called subpr
- At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter

9.5.2.4 Pass by reference
- Give the address of the variable
    - Provide the access path
- Efficient in both time and space
- Formal paramaters will be slower than pass-by-value
- BE CAREFUL OF WHO IS USING AND HAS ACCESS TO VARIABLES

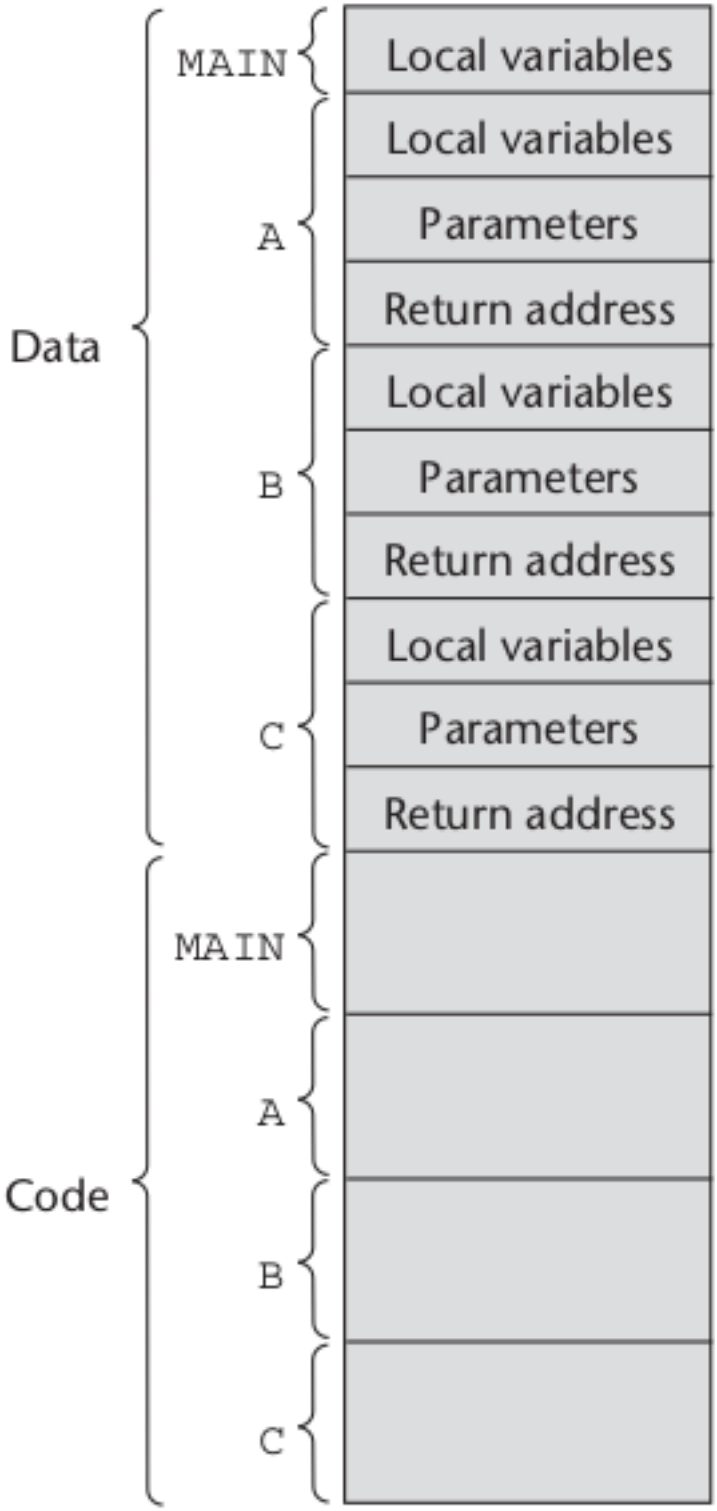
9.5.2.5 Pass by name

- parameter passing method that waits to evaluate the parameter value until it is used
- Simple semantic model as textual substitution

Chapter 10 Implementing Subprograms

10.2 Implementing Simple Subprograms
- Steps to call a program
    1. Save the execution status of the current program unit
    2. Compute and pass the parameters
    3. Pass the return address to the called
    4. Transfer control to the called
- Steps to return from a subprogram

    1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are mo
    2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
    3. The execution status of the caller is restored
    4. Control is transferred back to the caller
- Storage is required for the following
    - Status information about the caller
    - return address
    - Return value for the functions
    - Temporaries used by the code of the subprogram



10.3 Implementing Sub Programs with Stack Dynamic Variables

- The compiler must generate code to cause the implicit allocation and deallocation of local variables.
- Recursion allows multiple instances and calls to a subprogram
- Caller Actions
    1. Create an activation record instance
    2. Save the execution status of the current program unit
    3. Compute and pass the parameters
    4. Pass the return address to the called
    5. Transfer control to the called
- The prologue actions of the called are as follows:
    1. Save the old EP in the stack as the dynamic link and create the new value
    2. Allocate local variables

- Epilogue Actions
    1. Values of parameters moved to the actual parameters
    2. (If a function) move functional value to place accessible to the caller
    3. Restore stack pointer
    4. Restore execution status of caller
    5. Transfer control back to caller

Including Implementing non recursion func 10.3.1/10.3.2, recursion10.3.3 and tail recursion- class notes)
- Recursion
    - Dynamic chain
    - Use stack - recursive steps
    - Easier to write some advanced algorithms
    - Can be slower - require more memory, don't know when it will end (if it does)
    - Types
        - Non-tail
        - tail
            - The amount of stack consumed is fixed - like a loop
            - The last statement executed is the recursive call

```
//tail
int fact(int n, int acc) {
   if(n<1) return acc;
   else return fact(n-1,acc*n);
}
```

Chapter 11 Abstract Data Types and encapsulation constructs

11.1 The concept of abstraction
- remove the implementation
- Call a function to do the work, don't care about how it does it
    - only need it to perform the operation as expected

11.2 Introduction to Data Abstraction
- Objects, inheritance

11.4 Language Examples
- C++, Java

11.4.1 Abstract Data Types in C++
- Encapsulation, data hiding -> hide in the body
- Classes, inheritance

Chapter 12 Support for Object Oriented Programming

12.1 Introduction

12.2 Object OP

12.2.1 Introduction
- OOP Support requires:
    - abstract data types
    - inheritance
    - dynamic binding of method calls to methods

12.2.2 Inheritance
- Parent/Superclass -> child/derived class
- Override inherited methods/variables
- Instance methods and variables - specific to the object
- Class objects and variables - belongs to the class, only one

12.2.3 Dynamic binding

        - Also dynamic dispatch
        - Need to determine type on execution - could have changed classes
            - for polymorphic references
        -

12.3 Design Issues for OOP

    12.3.1 The exclusivity of objects
        - Types, wrapper classes, etc
        - Nonobject types

    12.3.2 Are subclasses subtypes
        - Determination of a subclass as a subtype
        - Overriding methods must have same return type, parameters

    12.3.3 Single and multiple inheritance. The diamond problem
        - Do you allow a class to inherit from multiple?
        - May have trouble determining which parent class to use -> diamond problem
            - Which version to inherit? Both? One?
        - How to solve? Disambiguation
            - Interfaces instead?

    12.3.4 Allocation and deallocation of objects
        - new/delete
            - Explicit
        - When to allocate? On call? Whenever?
        - Object slicing
            - Truncate class when not enough space on stack
                - using parent reference to child class

    12.3.5 Dynamic and static bindings
        - Static is faster
        - Specify if you want static or dynamic - pay the price when needed

    12.3.6 Initialization of objects
        - How do you initialize values when an object is created?
        - Manual or implicit?
        - On subclass created

From class notes. Aliasing in Java
    - Refering to the same object with different variables/names
    - BE CAREFUL DOING THIS
    - Modify one, modify the rest
    - Make sure modifying the correct actual object, names can be deceiving

4.2 Inheritance in C++
    - Multiple inheritence through interfaces, need to specify
        - VIRTUAL functions/classes
            - Define the implementation later

12.4.3 Dynamic binding C++
    - Type checking
    - Check up the chain of classes

12.5 Implementation of OO constructs
    - C++ classes, inheritance, and dynamic binding
    - C++ is OO

Chapter 15 Functional Programming

15.1 Intro
    - Everything is a function

15.2 Mathematical Functions
- Everything done by evaluation of expressions
- Lambda functions
  - Nameless -> anonymous
- Higher order functions
  - Multiple parameters or functions as yields
- Function composition

```
lambda(x)x * x * x
```

15.3 Fundamentals
- Referential transparency

15.4/5 Lisp/Scheme up to 15.5.13 (included)

15.4.1
- Two data types: lists and atoms
  - Lists
    - Like an array
    - Can be made of nested lists
    - Holds atoms
  - Atoms
    - Single elements
- QUOTE (')
  - DON'T EVALUATE AS A FUNCTION
  - For when you want to examine the literal data

```
//addition
(+ 3 5)
( function_name (LAMBDA ( arg 1 ... arg n ) expression ))
//anonymous functions
(LAMBDA (x) (* x x))
((LAMBDA (x) (* x x)) 7)
//function definition
(DEFINE tau (* 2 pi))
//conditionals
(COND
  (cond1 exp1)
  (cond2 exp2)
  [(ELSE expn)]
)
(COND
  ((< x y) "X smaller than y")
  ((> x y) "X is bigger than y")
  (ELSE :X is equal to Y")
)
//list functions
    //CAR gives first element
    //CDR gives rest
    //give second element of list
    (DEFINE (secondList) (CAR ( CDR list)))
//Equality
    //EQ?
    //LIST?
    //NULL?
// Function checks if an atom is a member of a list
(member 'A '(B C D)) //-> F
RECURSION!
(DEFINE (member atm list1)
  (COND
    (EQ? atm (car list1) #T)
```

```
        (ELSE (member atm (CDR list1)))
      )
    )
```

From Class Notes

Chapter 16 Logic Programming

16.2 A brief introduction to Predicate Calculus
- Propositions -> Logical statement that may or may not be true
  - Can be stated as a question
  - Can be stated as a truth -> build knowledge database
- Predicates
- Symbolic logic

16.4 An overview of Logic Programming
- Declarative Semantics
- State the conditions of the result, not how to get it

16.6 The Basic Elements of Prolog
- Edinburgh Syntax
- Terms
  - Constant
    - atom
    - integer
  - Variable
  - Structure
- Fact statements
  - State information
  - Can state implications
  - Perform breadth-first and depth-first search to find the global
    - DFS because fewer resources
    - BFS can be highly parallel, but lots of memory
    - Backtracking for subgoals

```
//knowledge statements
find_max(List, Max) :-
   select(Max, List, Tail), \+ (member(Element, Tail), Element > Max).
last([List|Rest], Last) :- last_(Rest, List, Last).
    last_([], Last, Last).
    last_([List|Rest], _, Last) :- last_(Rest, List, Last).
//goal statements
find_max([3,4,5,6,7], X). //7
last([3,4,5,8,7], X). //7
```

16.7 Deficiencies of Prolog
- Not pure or perfect
- Always matches in the same order
- (!) -> cut operator
  - Subgoal
  - Succeeds immediately, backtracking cannot solve
  - Don’t backtrack past this point
- Generate and test
- Assumes closed world
  - Based only on the knowledge DB
- Negation is difficult
  - Uninstantiates all variables in a goal that fails
- Intrinsic
  - Don’t specify how it gets there, only the result
  - Can be very slow then
  - Much less optimisation available
- Shouldn’t be able to influence execution time

        - But you can
        - Order statements
        - Solution trees
        - Goals

16.8 Applications of Logic Programming
    - Relational database -> SQL
    - Expert symbols
    - Natural language programming
    - AI

Chapter 13 Concurrency

Explain difference between Parallel, Distributed and concurrent programming
    - Concurrent
        - At the same time
        - Same time execution of instructions
        - Threads (like java thread library)
        - Thread level parallelism
        - Make the program run faster
        - Multicore -> use more
        - Single core -> use cache better
    - Parallel
        - Run faster at scale
    - Distributed
        - Multiple computers interconnected
        - Communication protocol to send/receive
        - Sync assumed by protocol

Explain difference between shared memory and distributed memory
    - Shared memory is shared
        - Everybody can access, race conditions
        - Like shared memory on a multicore processor
        - Break up the work - but beware race conditions
            - Sync to be sure work is done
            - Protect shared memory from being accessed at the same time
    - Distributed memory is for the individual
        - Good for scaling

Explain Different ways of achieving parallelism
    - Instruction
        - Hardware is in parallel, but hides the details
            - Pipeline
            - IPL
    - Tasks done by OS
        - Divide up independent tasks among cores
    - Data
        - One function applied to huge amounts of data
        - FP is good for concurrency

```
parallel(func);
pthread(func); //c
```

    - Threads
        - Java library are part of standard process -> program on executing
          - Loader (O.S).
            - Determines this is the PC
            - Instructions loaded on L1
            - SP
            - Registers
            - PE,TLB
            - ...

- Thread
  - Lightweight process

  - Minimum amount of code that you can execute. Threads share some state with the process and have also s
    - Important: Have their own PC - program counter
    - Share code
  - Contains:
    - Thread -> create/destroy
    - Synchronize threads
    - Avoid race conditions
      - Locks
      - Semaphors
      - Monitors
      - Atomic operations

Define Thread Level Paralellism- Java like concurrency
- Threads in the same process share address space
- Allows for race conditions among shared data
- Scheduler determines what runs and when
- All of java runs in threads