COMP 141: Midterm Exam Sample Questions

1. What is von Neumann architecture? *It is a computational architecture with general purpose operations, and I/O channels communicated in binary*
2. Enumerate the four main programming paradigms and differentiate them.
   1. *imperative: sequential execution of instructions that modify the state (memory) through variable assignments*
   2. *functional: based on the abstract notion of function, with no program state (memory) in the execution model*
   3. *logic programming: declaring computation as a set of logical statements (rules and facts)*
   4. *object-oriented: computation is modeled by decomposing it into multiple objects that interact with each otherusing their own methods*
3. Define what the syntax and semantics of a PL are.

*Syntax is the study of language constructs from a grammatical point of view.*

*Semantics is the study of the language constructs from the executional point of view, i.e., how the constructs are executed.*

1. Explain the functionality of each of the following.
   * 1. Compiler *translates source code to low-level code (machine code, assembly, byte code)*
     2. Assembler *translates assembly to machine code*
     3. Linker *brings multiple machine codes into a single executable machine code*
     4. Loader *loads the executable machine code to the memory*
     5. Interpreter *receives source-code (in byte code usually) and evaluates it*
     6. Scanner *Receives stream of characters (program text) and identifies tokens with their types*
     7. Parser *receives stream of tokens and builds syntax trees*
     8. Evaluator *receives syntax trees and evaluates them*
2. Explain what the following criteria for PL design are.
   1. Readability *easiness to read the program*
   2. Writability *easiness to code*
   3. Expressiveness *capability to specify a concept*
   4. Reliability *robustness against error-proneness*
3. In Pascal, functions can return scalar and pointer types but cannot return structured types like sets, files, and arrays. Lack of which criterion this refers to? *orthogonality, regularity*
4. Consider the following CFG, called *G*1.

*expr* ∶∶= *expr* @*expr* ∣ *expr* #*expr* ∣ (*expr*) ∣ SYMBOL

SYMBOL = A ∣ B ∣ C

* 1. Consider the following derivations for the expression A#B@C.

*expr* ⇒ *expr* #*expr* ⇒ SYMBOL#*expr* ⇒ A#*expr* ⇒ A#*expr* @*expr* ⇒ A#SYMBOL@*expr*

⇒ A#B@*expr* ⇒ A#B@SYMBOL ⇒ A#B@C

*expr* ⇒ *expr* @*expr* ⇒ *expr* #*expr* @*expr* ⇒ SYMBOL#*expr* @*expr* ⇒ A#*expr* @*expr*

⇒ A#SYMBOL@*expr* ⇒ A#B@*expr* ⇒ A#B@SYMBOL ⇒ A#B@C

Give the parse tree that corresponds to each of the these derivations.

*expr expr*

*expr*

***SYMBOL***

***A***

#

*expr*

*expr*

***SYMBOL***

*@*

*expr*

***SYMBOL***

*expr*

*expr*

***SYMBOL***

#

*expr*

***SYMBOL***

*@*

*expr*

***SYMBOL***

***C***

***B C A B***

1. What are the ASTs for the parse trees given in the previous question?

# *@*

***A***

*@*

#

***C***

***B C A B***

1. Give the disambiguated version of *G*1, called *G*2, considering the following precedence cascade among the operators:

• The highest precedence is for parentheses, • the second highest precedence is for # , • the least precedence is for @ .

*expr* → *expr* @ *term* ∣ *term term* → *term* # *factor* ∣ *factor*

*factor* → (*expr*) ∣ ***SYMBOL***

***SYMBOL*** = ***A*** ∣ ***B*** ∣ ***C***

1. Give the single parse tree that can be generated using *G*2.

*expr*

*expr*

*term*

*term*

*factor*

***SYMBOL***

***A***

#

*factor*

***SYMBOL***

***B***

*@*

*term*

*factor*

***SYMBOL***

***C***

1. What would be the AST for the parse tree in the previous question? *@*

#

***C***

***A B***

1. Redefine *G*2 using EBNF.

*expr* → *term* { @ *term*} *term* → *factor* { # *factor*} *factor* → (*expr*) ∣ ***SYMBOL***

***SYMBOL*** = ***A*** ∣ ***B*** ∣ ***C***

1. Conisder the following CFG rule.

*e* ∶∶= *e* ▲ A ∣ A

* 1. Is ▲ defined as left-recursive or right-recursive? *left-recursive*
  2. Redefine the CFG to make ▲ be left-recursive if it is already right-recursive. Redefine the CFG to make ▲ be right-recursive if it is already left-recursive.

*e* → ***A*** ▲ *e* ∣ ***A***

* 1. For each of the left-recursive and right-recursive definitions in the two previous questions, give the EBNF definition. *left-recursive:*

*e* → ***A***{▲***A***}

*right-recursive:*

*e* → ***A***[▲*e*]

1. Multiplying numbers in a list:
   1. Define a recursive function in Haskell that receives a list of integers and multiplies the elements together. (You may assume that if the list is empty, the result is 1).

***mullist [] = 1***

***mullist (x:xs) = x \* (mullist xs)***

* 1. What would be the inferred type of this function? ***mullist :: Num a => [a] -> a***

1. Define a function cprod in Haskell that receives three lists in input and returns the Cartesian product of them. For example, cprod [1,2] ["a","c"] [True] must return [(1,"a", True), (1,"b",True), (2,"a",True), (2,"b",True)]. *Hint*: Use list comprehensions.

cprod xs ys zs = [(x,y,z) | x <- xs, y <- ys, z <- zs]

1. Give an example where strict and lazy evaluations end in different results.

Consider the following example (in Haskell syntax):

f x = f (x\*2) g x = 1

With strict evaluation (which is Haskell does not have), g (f 1) overflows. With lazy evaluation (which Haskell has), g (f 1) returns 1.