COMP 141: Midterm Exam Sample Questions

(1) What is von Neumann architecture? ***It is a computational architecture in which there is a fixed processing unit with general purpose operations, and I/O channels through which programs are communicated in binary format and results are returned.***

(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 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 other using 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 oif the language constructs from the executional point of view, i.e., how the constructs are executed.

(4) Explain the functionality of each of the following.

- (a) Compiler ***translates source code to low-level code (machine code, assembly, byte code)***
- (b) Assembler ***translates assembly to machine code***
- (c) Linker ***brings multiple machine codes into a single executable machine code***
- (d) Loader ***loads the executable machine code to the memory***
- (e) Interpreter ***receives source-code (in byte code usually) and evaluates it***
- (f) Scanner ***Receives stream of characters (program text) and identifies tokens with their types***
- (g) Parser ***receives stream of tokens and builds syntax trees***
- (h) Evaluator ***receives syntax trees and evaluates them***

(5) Explain what the following criteria for PL design are.

- (a) Readability ***easiness to read the program***
- (b) Writability ***easiness to code***
- (c) Expressiveness ***capability to specify a concept***
- (d) Reliability ***robustness against error-proneness***

(6) 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***

(7) Consider the following CFG, called G_1.

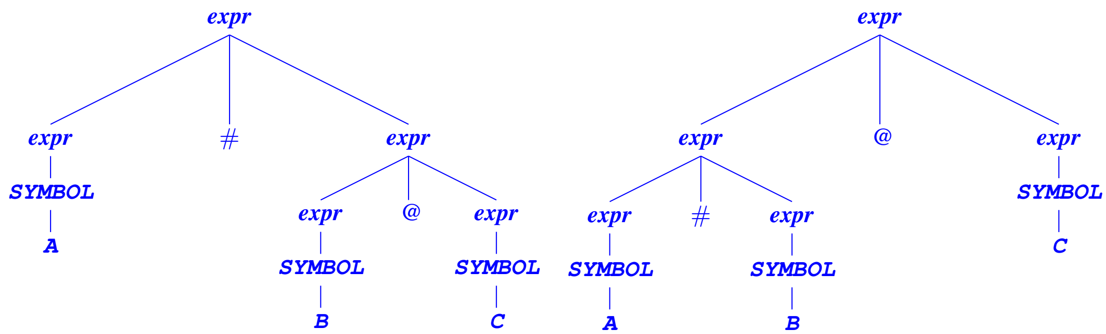
$$\mathit{expr} ::= \mathit{expr} \mathbin{@} \mathit{expr} \mid \mathit{expr} \# \mathit{expr} \mid (\mathit{expr}) \mid \texttt{SYMBOL}$$
$$\texttt{SYMBOL} = \texttt{A} \mid \texttt{B} \mid \texttt{C}$$

(a) Consider the following derivations for the expression $\texttt{A} \# \texttt{B} \mathbin{@} \texttt{C}$.

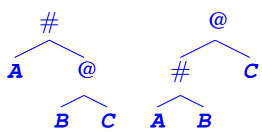
$$\mathit{expr} \Rightarrow \mathit{expr} \# \mathit{expr} \Rightarrow \texttt{SYMBOL} \# \mathit{expr} \Rightarrow \texttt{A} \# \mathit{expr} \Rightarrow \texttt{A} \# \mathit{expr} \mathbin{@} \mathit{expr} \Rightarrow \texttt{A} \# \texttt{SYMBOL} \mathbin{@} \mathit{expr}$$
$$\Rightarrow \texttt{A} \# \texttt{B} \mathbin{@} \mathit{expr} \Rightarrow \texttt{A} \# \texttt{B} \mathbin{@} \texttt{SYMBOL} \Rightarrow \texttt{A} \# \texttt{B} \mathbin{@} \texttt{C}$$

$$\mathit{expr} \Rightarrow \mathit{expr} \mathbin{@} \mathit{expr} \Rightarrow \mathit{expr} \# \mathit{expr} \mathbin{@} \mathit{expr} \Rightarrow \texttt{SYMBOL} \# \mathit{expr} \mathbin{@} \mathit{expr} \Rightarrow \texttt{A} \# \mathit{expr} \mathbin{@} \mathit{expr}$$
$$\Rightarrow \texttt{A} \# \texttt{SYMBOL} \mathbin{@} \mathit{expr} \Rightarrow \texttt{A} \# \texttt{B} \mathbin{@} \mathit{expr} \Rightarrow \texttt{A} \# \texttt{B} \mathbin{@} \texttt{SYMBOL} \Rightarrow \texttt{A} \# \texttt{B} \mathbin{@} \texttt{C}$$

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



(b) What are the ASTs for the parse trees given in the previous question?

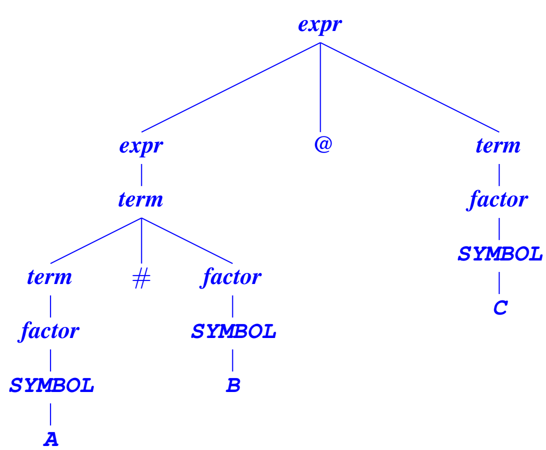


(c) Give the disambiguated version of G_1, called G_2, considering the following precedence cascade among the operators:

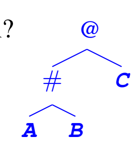
- The highest precedence is for parentheses,
- the second highest precedence is for #,
- the least precedence is for `@`.

$$expr \rightarrow expr \text{ @ } term \mid term$$
$$term \rightarrow term \ \# \ factor \mid factor$$
$$factor \rightarrow (expr) \mid \textbf{SYMBOL}$$
$$\textbf{SYMBOL} = \textbf{A} \mid \textbf{B} \mid \textbf{C}$$

(d) Give the single parse tree that can be generated using G_2.



(e) What would be the AST for the parse tree in the previous question?



(f) Redefine G_2 using EBNF.

$$\mathit{expr} \rightarrow \mathit{term} \; \{ \; @ \; \mathit{term} \}$$
$$\mathit{term} \rightarrow \mathit{factor} \; \{ \; \# \; \mathit{factor} \}$$
$$\mathit{factor} \rightarrow (\mathit{expr}) \mid \textbf{\textit{SYMBOL}}$$
$$\textbf{\textit{SYMBOL}} = \textbf{\textit{A}} \mid \textbf{\textit{B}} \mid \textbf{\textit{C}}$$

(8) Conisder the following CFG rule.

$$e ::= e \blacktriangle \mathtt{A} \mid \mathtt{A}$$

(a) Is ▲ defined as left-recursive or right-recursive? ***left-recursive***

(b) 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.

$$e \rightarrow \textbf{\textit{A}} \blacktriangle e \mid \textbf{\textit{A}}$$

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

$$e \rightarrow \textbf{\textit{A}}\{\blacktriangle \textbf{\textit{A}}\}$$

right-recursive:

$$e \rightarrow \textbf{\textit{A}}[\blacktriangle e]$$

(9) Multiplying numbers in a list:

(a) 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)
```

(b) What would be the inferred type of this function? `mullist :: Num a => [a] -> a`

(10) 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]
```

(11) 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.