COMP 141: Midterm Exam

Instructions: Complete each of the following problems. There are 100 total points with extra 20 points. Each problem has its own point value. Unless explicitly specified, if a problem has multiple parts, they are equi-valued. You can verify your Haskell code in **GHCi** before putting your answers here. You have **90 minutes** for this exam.

Name:

Problem 1 ***(4 points).*** Which **programming paradigm** does each of the following correspond to?

(a) sequential execution of instruction. ***imperative programming***

(b) program as a set of logical rules and facts. ***logic programming***

(c) passing functions as arguments to other functions. ***functional programming***

(d) using assignment to change the values of variables. ***imperative***

Problem 2 ***(4 points).*** For each of the following, specify the tool's name.

1. Receives a list of tokens and creates the syntax tree. ***Parser***
2. Receives multiple binary files and creates a single executable file. ***Linker***
3. Receives the syntax tree of the program and runs it. ***Evaluator***
4. Receives the program text and returns the assembly code of it. ***Compiler***

Problem 3 ***(4 points).*** Each of the following cases represents lack of a specific PL design criterion. Specify that criterion in each case.

(a) C includes a large set of numeric types. It may introduce confusion in programming.

Reliability, Security, Maintainability

(b) In Assembly, a simple program may end up being hundreds of code lines. ***Expressiveness, Readability, Writibility***

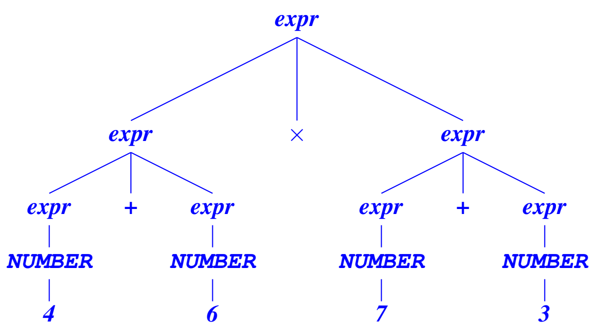
Problem 4 ***(35 points).*** Consider the following **CFG**, called G_1.

$$expr ::= expr + expr \mid expr \times expr \mid (expr) \mid \texttt{NUMBER}$$
$$\texttt{NUMBER} = \texttt{[0-9]+}$$

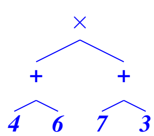
(a) Consider the following derivation for the expression $4 + 6 \times 7 + 3$.

$$
\begin{aligned}
expr &\Rightarrow expr \times expr \\
&\Rightarrow expr + expr \times expr \\
&\Rightarrow \texttt{NUMBER} + expr \times expr \\
&\Rightarrow 4 + expr \times expr \\
&\Rightarrow 4 + \texttt{NUMBER} \times expr \\
&\Rightarrow 4 + 6 \times expr \\
&\Rightarrow 4 + 6 \times expr + expr \Rightarrow 4 + 6 \times \texttt{NUMBER} + expr \\
&\Rightarrow 4 + 6 \times 7 + expr \\
&\Rightarrow 4 + 6 \times 7 + \texttt{NUMBER} \\
&\Rightarrow 4 + 6 \times 7 + 3
\end{aligned}
$$

Give the **parse tree** that corresponds to this derivations ***(5 points)***.



(b) What is the **AST** for the parse tree given in the previous question ***(5 points)***?



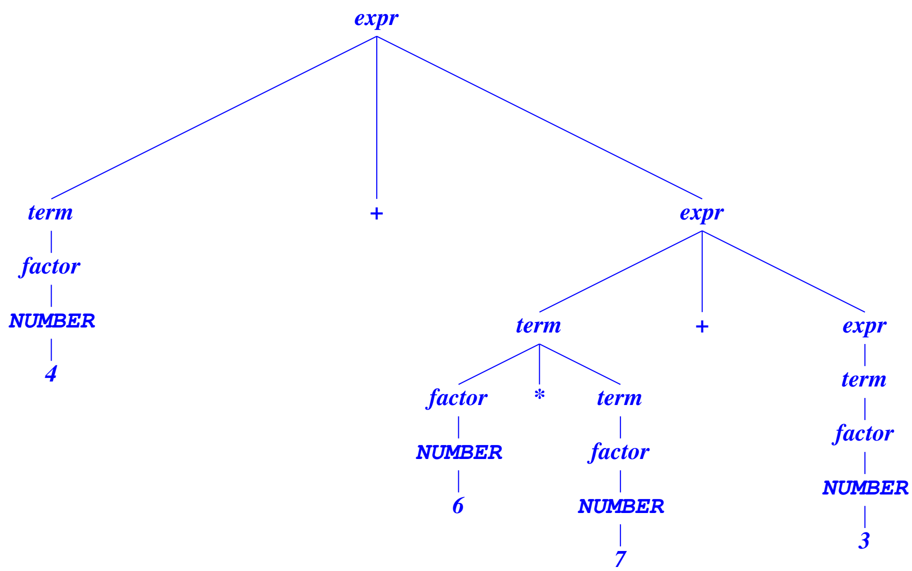
(c) Give the **disambiguated** version of G_1, called G_2, considering the following precedence cascade among the operators ***(10 points)***:

- The highest precedence is for parentheses,
- the second highest precedence is for $\times$,
- the least precedence is for $+$.

In addition, assume that both $+$ and $\times$ are **right-recursive**.

$$
\begin{aligned}
&expr ::= term + expr \mid term \\
&term ::= factor \times term \mid factor \\
&factor ::= (expr) \mid \textit{\textbf{NUMBER}} \\
&\textit{\textbf{NUMBER}} = \texttt{[0-9]+}
\end{aligned}
$$

(d) Since $G2$ is disambiguated, it generates a single syntax tree for each expression. Give the single **parse tree** that can be generated using G_2 for expression above, i.e., $4 + 6 \times 7 + 3$. ***(5 points)***.



(e) Redefine G_2 using **EBNF** ***(10 points)***.

$$expr ::= term\ [+\ expr]$$
$$term ::= factor\ [\times\ term]$$
$$factor ::= (expr) \mid \textbf{NUMBER}$$
$$\textbf{NUMBER} = \texttt{[0-9]+}$$

Problem 5 ***(3 points)*****.** For each of the following items, specify whether functional programming is a good choice or not. (Yes/No)

1. linear computations
2. parallel computations
3. mathematical analysis of program behavior

no, yes, yes

Problem 6 ***(50 points)*****.** Functional programming in Haskell:

(a) Define function `app` in Haskell that receives two lists (of the same type) and returns the appendage of them. For example, `app [5,4,2] [8,6]` would be evaluated to `[5,4,2,8,6]`. Use **pattern matching** on one of input lists to define `app`. You cannot use the library function ++ for this purpose.

```
app [] ys = ys
app (x:xs) ys = x:(app xs ys)
```

(b) Use **list comprehension** to define function `len` that receives a list (of any type) and returns the length of the input list. You cannot use the library function `length` for this purpose.

```
len xs = sum [1 | x <- xs ]
```

(c) Define function `larger` that receives a list of pairs of numbers and returns a list of numbers in which the larger component of each pair is in the output list. For example, if input is `[(1,7), (9,6), (5,2), (2,3)]`, then the output would be `[7, 9, 5, 3]`. Use **list comprehension** for this purpose.

```
larger xs = [if x > y then x else y | (x,y) <- xs]
```

(d) Use **list ranges** to specify all numbers between 4 and 60 that divide 7.

```
[7,14 ..  60]
```

(e) Define function `count` that receives an item x along with a list ys as input and returns the number of times x appears in ys. For example, `count 5 [1,5,2,5,1]` should return `2`, whereas `count 5 [1,2,1]` should return `0`. You can use pattern matching on lists to define this functions.

```
count _ [] = 0
count x (y:ys) = if (x == y) then 1 + (count x ys) else (count x ys)
```

Problem 7 ***(10 points).*** ***(Extra point)*** For each of the following type expressions, define a Haskell function that has the type. Avoid annotating the type in the function definition.

(a) `(Num a, Num b) => (a, b) -> a -> b -> (a, b)` ***f p x y = (fst p + x, snd p + y)***

(b) `[(a, b)] -> [a]` ***f xs = [x | (x,y) <- xs]***

Problem 8 ***(10 points).*** ***(Extra point)*** Define function `secondFifth` that receives a list and returns the second fifth of that list. That is, the input list is divided to five equal size sublists and the second sublist is returned. For example,

```
ghci> secondFifth [1..10]
[3,4]
ghci> secondFifth [1..15]
[4,5,6]
ghci> secondFifth [1..20]
[5,6,7,8]
```

Let's assume that the number of items in the list is divisible to 5. *Hint*: You can use `take`, `drop`, `length` and `div`.

```
    secondFifth xs = drop ((length xs) `div` 5) (take (2 * ((length xs) `div` 5))
xs)
```