GitHub: https://github.com/poreldeporte/COP_4555_Programming_Languages

COP 4555
Coding 1
ID: 3319524

NOTE: Professor, I spoke with my classmates and we were unsure how the work is to be submitted. I have created this document with the whole assignment. My answers have comments alongside the solution. I have also committed every file to my GitHub (listed above, also sent it through my FIU email). Hope I got it right! Thank you.

Coding 1

1. Which of the following F# expressions is not well typed?

   4 + 5.6
   // this is not valid

2. A curried function has a type of which form?

   t1 -> (t2 -> t3)
   // arrow is right associative (right to left)

3. If an F# function has type *'a -> 'b when 'a : comparison*, which of the following is not a legal type for it?

   (float -> float) -> bool

4. Which of the following statements about F# lists is not true?

   Their built-in functions are polymorphic.

5. Which of the following F# expressions evaluates to [1; 2; 3]?

   1@2@3@[]

6. How does F# interpret the expression List.map List.head foo @ baz?

   ((List.map List.head) foo) @ baz
   // prefix before infix for functions

7. How does F# interpret the type int * bool -> string list?

   (int * bool) -> (string list)

8. Let F# function foo be defined as follows:

```
let rec foo = function
  | (xs, [])    -> xs
  | (xs, y::ys) -> foo (xs@[y], ys)
```

   If foo is supposed to append its two list parameters, which of the following is true?

   foo satisfies all three steps of the Checklist for Programming with Recursion.

9. Which of the following is the type that F# infers for (fun f -> f 17)?

   (int -> 'a) -> ‘a

10. Which of the following has type int -> int list?

    fun x -> x::[5]

11. What type does F# infer for the expression (3, [], true) ?

    int * 'a list * bool

12. What type does F# infer for the expression fun x y -> x+y+"." ?

    string -> string -> string

13. What type does F# infer for the expression fun xs -> List.map (+) xs ?

    int list -> (int -> int) list

14. Which of the following does F# infer to have type string -> string -> string ?

    fun (x, y) -> x + y + "."

15. Which of the following does F# infer to have type (string -> string) -> string ?

    fun f -> f (f "cat")

Problem 16

```
let answer() =
    // defining the gcd function
    let rec gcd = function
        | (a,0) -> a
        | (a,b) -> gcd (b, a % b)

    // defining the addition (.+) operator
```

```
    let (.+)(x, y)(p, q) =
        // performing the fraction addition
        let a = (x*q)+(y*p)
        let b = (y*q)

        // simplifying the result
        let c = gcd(a,b)
        (a/b, b/c);;

    // defining the (.*) operator
    let (.*)(x, y)(p, q) =
        let a = (x*p)
        let b = (y*q)
        let c = gcd(a,b)
        (a/c, b/c);;
```

Problem 17

```
let answer () =
    // this method in List.map reverses the list
    let revlists xs =
        List.map List.rev xs
    revlists [[0;1;1];[3;2];[];[5]]
```

Problem 18

```
// The F# function for interleaving two lists
let rec interleave = function
    |([], ys) -> ys
    |(xs, []) -> xs
    |(x::xs, y::ys) -> x :: y :: interleave (xs,ys)

interleave ([1;2;3],[4;5;6])
```

Problem 19

```
let answer() =
        let cut xs = gencutdata((List.length xs) / 2, xs)

        // cut calls this F# function that cuts a list into two equal parts
        let rec gencutdata(n, list) =
                let rec cut n (val : int list) (list : int list) =
                match (n , list ) with
                | 0, _ -> val, list
                | _, [] -> val, list
                | _, b :: list -> cut (n - 1) (List.rev (b :: val )) list
        cut n [] list

// Call cut with the list value
cut [1;2;3;4;5;6]
```

Problem 20

```
let shuffle list =
   list |> cut |> interleave

shuffle [1;2;3;4;5;6;7;8]
```

Problem 21

```
let rec countaux = function
        | (deck, target) when deck = target -> 0
        | (deck, target) -> 1 + countaux ((shuffle deck), target)

let countshuffle n = countaux((shuffle [1..n]), [1..n]) + 1
```

Problem 22

```
// the cartesian function uses the List.map function to create the list pairs
let rec cartesian (xs, ys) =
        match (xs, ys) with
        | (_, []) -> []
        | ([], _) -> []
        // the list pairs that represents the Cartesian product
        | (x::xs, _) -> (List.map (fun y -> (x,y)) ys)@cartesian (xs, ys)
```

Problem 23

```
// definition of the function
let rec powerset = function

  // if the list is empty then return empty list
  | [] -> [[]]

  // if the list is not empty, break the list into the subsets and call the function
recursively to find all the possible subsets
  | (l::l1) -> let l11 = powerset l1
                                List.map (fun l1' -> l::l1') l11 @ l11

// calling powerset and storing in powset
let powset = powerset [1;2;3]

// print the powerset
printfn "%A" powset;;
```

Problem 24

```
let rec transpose matrix = function
        match matrix with
        // if empty
        | row::rows ->          match row with
        // if not empty
        | col::cols ->  let first = List.map List.head matrix
                        let rest = transpose (List.map List.tail matrix)
                                first :: rest
                                | _ -> []
                                | _ -> [];;
```

Problem 25

1) If the base cases work - which they do

2) If the non base case functions correctly out the input, and produce correct results. It fails case 2 because the function does not sort the list

3) If the recursive call takes a smaller input, which it does pass a smaller input so it passes.

Problem 26

// The mergesort function is missing a case causing the type to be 'a list -> 'b list instead of 'a list -> 'a list. This is because we are not looking to change the type of the list.

// Another problem with the code is that neither merge nor split is tail recursive. This means that it will therefore get stack overflow exceptions on large lists

// Make split and merge functions tail recursive by using the accumulator pattern

Problem 27

E -> E+T | E-T | T

T -> T*Y | T/Y | Y

Y -> F^Y | F

F -> i | (E)

In this way, your grammar works as expected with an expression like :

1) i+i^i*i
2) p+q^r^s^t*u

Problem 28

It has two parse frees for a derivation which confirms that given grammar is ambiguous