DCC888 - Static Analysis of Programs

Name: _____ **ID**: _____

By turning this exam in, I give my word that I have done it alone, on the understanding that I am allowed to consult any material publicly available, except material disclosed by other colleagues who are taking this course, or who have taken it in the past.

This exam concerns a simple functional language, that uses the following grammar:

```
F ::= fun <name> R D = E
E ::= D
    | E op E
    | <name> R E
    | (E)
op ::= + | - | * | @ | div |
         mod | and | or
D ::= <int> | true | false | L
L ::= nil | [D] | D :: L
R ::= Zero | (Succ R)
```

Examples of programs are given below:

Program 1: Factorial

```
fun fact Zero 0 = 1
  | fact (Succ N) n = n * fact N (n-1)
```

Program 2: counts the number of true's in a list

```
fun countTrue Zero nil = 0
  | countTrue (Succ N) (true::L) =
        1 + countTrue N L
  | countTrue (Succ N) (false::L) =
        countTrue N L
```

Program 3: sums up the elements of a list

```
fun sum Zero nil = 0
  | sum (Succ N) (h::t) = h + sum N t
```

Program 4: inverts a list:

```
fun inv Zero nil = nil
  | inv (Succ N) (h::L) = (inv N L) @ [h]
```

The operator @ concatenates two lits, e.g., (1::[2]) @ (3::[4]) = (1::2::3::[4])

Examples of the execution of these programs can be seen below:

```
- fact Zero 0;
val it = 1 : int

- inv (Succ (Succ (Succ (Succ Zero)))) (false::true::false::[true]);
val it = true::false::true::[false] : bool list

- fact (Succ (Succ Zero)) 2;
val it = 2 : int

- countTrue (Succ (Succ (Succ Zero))) (true::false::[true]);
val it = 2 : int

- sum (Succ (Succ (Succ Zero))) (1::2::[3]);
val it = 6 : int

- sum (Succ (Succ (Succ Zero))) (1::2::3::[4]);
  Error!
```

Question 1 (20 Points): In this programming language, it is possible to implement programs that do not terminate. For instance:

Program 5: count to infinite

```
fun countInf Zero n = countInf Zero (n + 1)
```

Program 6: invert forever

```
fun loopInv (Succ Zero) true = loopInv Zero false
  | loopInv Zero false = loopInv (Succ Zero) true
```

Program 7: cycle the list

```
fun eternalCon (Succ N) (h::L) = eternalCon (Succ N) (L @ [h])
```

Design a type system for this programming language that rules out any program that might not terminate. Your type system must be conservative, so, it might rule out programs that will terminate. However, if it allows a program to type check, then that program must necessarily terminate. Additionally, your type system cannot be trivial, that is, it cannot simply rule out every program. In particular, it must rule out the two programs above, and accept all the four programs used as examples in the beginning of this question.

In class, we have used only one environment to implement type checking, e.g., the Sigma table. So, in class we saw rules like $\Sigma \vdash E:T$, meaning that E has type T, given the biddings in Σ. However, for this question, you might want to use more environments (only if you want, of course). Each new environment will hold information of a particular kind.

Extra A (1 Point): implement a grammar for our language in any programming language, and implement yur type checker using this grammar to guide your analysis.