CS 558 Programming Languages Winter 2020 – Practice Mid-term Exam Sample Solutions

1.(a) Yes, the grammar is ambiguous, because there are expressions with two distinct parse trees. For example, the expression \(x + y + z\) has these two parse trees:

\[
\begin{array}{c}
\text{expr} \\
\text{term} \\
\text{id} + \text{term} \\
\text{id} + \text{id} + \text{term} \\
\end{array}
\]

\[
\begin{array}{c}
\text{expr} \\
\text{term} \\
\text{id} + \text{id} + \text{term} \\
\text{id} + \text{id} + \text{id} \\
\end{array}
\]

(b) No such program can exist. The only way evaluation order can affect the result is because of side-effects, and the only side-effect in the language is assignment. The operands of ‘+’ and ‘*’ are always terms; evaluating terms never causes an assignment. So it can’t matter which order the operands are evaluated.

2.(a,b,c) Marked below. For (e), the idea is that to achieve dynamic scope, we simply set the initial environment for evaluating functions to the caller’s environment (rather than the lexical environment of the callee, which for this language is empty).

case class Program(fdefs:List[FunDef], body:Expr) {} // ((fdefs) body)
case class FunDef(name:String, param:String, body:Expr) {} // (name param body)
sealed abstract class Expr {}
case class Num(n:Integer) extends Expr // n
case class Var(x:String) extends Expr // x
case class Add(l:Expr, r:Expr) extends Expr // (+ e e)
case class Apply(f:String, e:Expr) extends Expr // (@ f e)
case class Let(x:String, e:Expr, b:Expr) extends Expr // (let x e b)

type Env = Map[String,Int]
val emptyEnv : Env = Map[]()
def interp(p:Program): Int = {
def interpE(env:Env,e:Expr) : Int = e match {
  case Num(n) => n
  case Var(x) => env.getOrElse(x, throw InterpException("undefined variable:" + x))
  case Add(l,r) => interpE(env,l) + interpE(env,r)
  case Let(x,e,b) =>
    val v = interpE(env,e)
    interpE(v + (x->v),b)
  case Apply(f,Expr) =>
    val v = interpE(env,f)
    for (fdef <- p.fdefs)
      if (fdef.name == f)
        return interpE(EmptyEnv + (fdef.param->v),fdef.body)
    throw InterpException("undefined function:" + f)
  }
  interpE(emptyEnv,p.body)
}

(c) Under static scoping, the use of \(y\) in the body of \(g\) is free, so interpreting the program raises the exception InterpException("undefined variable:y").

(d) Under dynamic scoping, the binding of \(y\) is visible inside \(g\), so the program evaluates to \(21 + 21 = 42\).
3. The key idea is that two pairs are reference-equal iff a change to an element in one pair is visible in the other. So our function should use `setFst` (or `setSnd`) to alter one pair and `fst` (or `snd`) to see if that has also changed the other pair. The rest is just details: we must do the update with a value that is different from the existing element value (in both pairs), and we must reset the original element value after completing the test (no matter how it comes out).

Here’s one concrete implementation:

\[
\text{eqpair (a b)}
\begin{align*}
\text{let q (fst a)}
\text{if (== q (fst b))}
\{ (fst a) = (fst b) = q \}
\text{let p (if (== q 42) 43 42) \{ any p != q will do \}}
\text{block}
\text{(setFst a p)}
\text{let r (== (fst b) p)}
\text{block}
\text{(setFst a q)}
\text{r))}
\{ (fst a) != (fst b), so pairs cannot be the same \}
\end{align*}
\]

4. (a)

\[
w_1 = L(1,L(2,L(3,null)))
a_1 = L(3,null)
w_2 = L(1,L(2,L(3,null)))
u_1 = L(3,L(2,L(1,null)))
w_3 = L(1,null)
a_2 = L(3,L(2,L(1,null)))
\]

The `last` function returns the last (boxed) cell of the list, i.e. the single-element list consisting of the last element of the argument list. (Note that it returns a cell, not the value in the cell.) It does not change any pointers in its list argument. The `rev` function reverses the list in place, returning the (new) head of the reversed list; it is an imperative version of the list reversal problem posed in lab. The only tricky point is that reversing the list in place changes the cell containing 3, so `a_2` does not print the same as `a_1`.

(b)

\[
w_1 = L(1,L(2,L(3,null)))
a_1 = L(3,null)
w_2 = L(3,null)
u_1 = L(3,null)
w_3 = null
a_2 = L(3,null)
\]

Under call-by-reference, any call `last(z)` updates `z` to be the same as the return value, i.e. the last (boxed) cell in the list; hence `w_2 = a_2`. Calling `rev` on this just returns the same list (reversing a single element list doesn’t do anything), but under call-by-reference, any call `rev(z)` sets `z` to null.

5. (a)

\[
S; \text{while } (\neg E) \text{ do } S
\]

(b)

\[
\frac{(P)S(Q) \quad (Q \land \neg E)S(Q)}{(P) \text{ repeat } S \text{ until } E \quad (Q \land E) (\text{REPEAT-UNTIL})}
\]

6.

\[
\text{(Gif-1)}
\]

\[
\text{(Gif-2)}
\]