1. [20 pts.] Grammars and Order of evaluation

Consider the following concrete grammar for a language A of arithmetic expressions, where $exp$ is the start symbol:

$$
exp ::= \begin{array}{l}
\text{term} \\
\text{let } \text{id} \ '=>' \ exp \ \text{in} \ exp \\
\text{id} \ '=>' \ exp
\end{array}
$$

$$
term ::= \begin{array}{l}
\text{term} \ '+' \ \text{term} \\
\text{term} \ '*' \ \text{term} \\
\text{id} \\
\text{num}
\end{array}
$$

The intended semantics are just as in the lab interpreters: $id$ represents variables; $num$ represents integer literals; $x := e$ evaluates $e$ to a value $v$, sets the value of $x$ to $v$, and yields $v$ as result; $+$ is addition; $*$ is multiplication; let $x = e_1$ in $e_2$ binds $x$ to the value of $e_1$ inside $e_2$ and yields the value of the latter.

(a) [10 pts.] Is this grammar ambiguous? Why or why not?

(b) [10 pts.] Suppose we have two interpreters for this language that both build the same AST as the result of parsing any given concrete input. Interpreter #1 evaluates the operands of $+$ and $*$ from left-to-right; interpreter #2 evaluates them from right-to-left. Is there an A program that behaves differently under the two interpreters? Either show such a program or explain why it can’t exist.
2. [20 pts.] **Scope and Binding**

Consider the following Scala interpreter code for a simple statically-scoped language L of top-level functions and expressions. The s-expression forms for each language construct are shown in comments on the AST case class definitions. Since there is no assignment operator, we do not bother to model the store; instead, the environment binds variables directly to (integer) values.

```scala
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[String, Int]()

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(env + (x -> v), b)
}
case Apply(f, e) => {
  val v = interpE(env, e)
  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)
}
```

(a) [6 pts.] In the interpreter code above, consider the three identifiers p, e, and env. Draw a box around each binding of these identifiers, and a circle around each use of these identifiers.

(b) [3 pts.] Draw an arrow from each use of identifier e (only) to the corresponding binding for e to which it refers.

(c) [3 pts.] What is the result of interpreting the following L program?

```scala
((g z (+ z y))
  (let y 21
    (@ g y)))
```

(d) [3 pts.] Suppose L used dynamic scope instead of static scope. What would be the result of the program from part (c)?

(e) [5 pts.] Change the interpreter code above to implement dynamic scope instead of static scope. (Hint: This requires changing just one variable use!) Mark your change clearly in the code listing.
3. Storage

Consider the following C program fragment:

```c
void f() {
    int a = 42;
}
int g() {
    int b;
    return b;
}
int h() {
    f();
    return g();
}
```

(a) [5 pts.] According to the C language definition, it is undefined what result value is returned by \( h \). Why?

(b) [5 pts.] Suppose that when this code is given to the \texttt{gcc} C compiler (with default settings), it produces machine code that, when run, returns the result value 42 from \( h \). Give a plausible explanation for why this might happen.

(c) [5 pts.] What would happen if this code were given to a Java compiler? (To make this a syntactically legal Java fragment, assume these are member functions of some class.)

(d) [5 pts.] What would we need to do to convert the body of \( g \) into legal Scala?
4. [20 pts. *] Reference Equality

[* This problem is frankly too difficult for an exam, but it is interesting to look at anyway!]

Recall the E4 toy language from week 4’s lab, including `let`, `setFst`, and `setSnd` expressions (as you added them in the lab assignment) and extended with an equality operator `eq`. For reference, the s-expression EBNF syntax for this language is at the bottom of the page.

A value is either an integer or a (boxed) pair. As usual, the first and second elements of a pair can each be an integer or a further pair.

Assume `eq` is a structural equality operator: it returns 1 if its arguments are identical integers or pairs with structurally equal contents, and 0 otherwise. (This is the same as the meaning of the `eq` operator you defined in the lab assignment, but in this case it is built into the language.)

Show that a reference equality operator on pairs can be defined inside language E4. In other words, write an E4 function `eqpair` such that $(\@ \text{eqpair} \ e_1 \ e_2)$ returns 1 if $e_1$ and $e_2$ are the same pair (have the same location), and 0 otherwise. You may assume that `eqpair` is only called on pairs. Use the s-expression syntax to write your function.

Hints: (1) Make use of the fact that pairs are mutable. (2) You may find it easier to develop the code in two stages. First, write a version of `eqpair` that works on the assumption that pairs being compared never contain the integer 42 as an element. Then, modify your code to work properly even on pairs that might contain 42.

```
program ::= (gdecs fdecs exp)
gdecs ::= {{(id exp)}}
fdecs ::= {{(id {(id) exp})}}
exp ::= num | id | (:= id exp) | (while exp exp) | (if exp exp exp) | (block {exp}) |
       {0 id {exp}} | (let id exp exp) | (write exp) | 
       (+ exp exp) | (- exp exp) | (* exp exp) | (/ exp exp) | (\ exp exp) | (eq exp exp) |
       (pair exp exp) | (fst exp) | (snd exp) | (isPair exp) | (setFst exp exp) | (setSnd exp exp)
```
5. [20 pts.] Axiomatic Semantics

Recall the very simple imperative language and set of rules used to illustrate axiomatic semantics in Lecture 2b, which are repeated for reference at the bottom of this page. Suppose we want to add a new repeat-until statement. The form of this statement is

\[ \text{repeat } S \text{ until } E \]

The semantics of this statement follow from the fact that it compiles to the following low-level code sequence (in the style of Lecture 2a):

```
top:
  S
  evaluate E into t
  cmp t, true
  brneq top
```

Here is a valid triple describing the behavior of a particular program fragment involving repeat-until:

\[
\{ x = -1 \land y > 0 \}
\]
\[
\text{repeat}
\]
\[
x := x + y;
\]
\[
y := y - 1
\]
\[
\text{until } (y \leq 0)
\]
\[
\{ x \geq 0 \land y \leq 0 \}
\]

(a) [5 pts.] Give a combination of while and sequential composition (\(;\)) statements that is equivalent to repeat \( S \) until \( E \).

(b) [15 pts.] Give the strongest proof rule you can for repeat-until statements. In particular, your rule should be strong enough to justify the above triple. It is not necessary to write down a proof of this triple as part of your answer, but you may find that doing so is helpful to check that your rule works as expected.

\[
\{ P[E/x] \} \ x := E \ { P \} \quad \text{(ASSIGN)}
\]
\[
\{ P \land E \} \ S_1 \ { Q \} \quad \{ P \land \neg E \} \ S_2 \ { Q \} \quad \text{(COND)}
\]
\[
\{ P \} \ \text{if } e \ \text{then } S_1 \ \text{else } S_2 \ { Q \} \quad \{ P \} \ S_1 ; S_2 \ { R \} \quad \text{(COMP)}
\]
\[
\{ P \} \ \text{while } E \ \text{do } S \ \text{end} \ { P \land \neg E \} \quad \text{(WHILE)}
\]
\[
\{ P \} \skip \ { P \} \quad \text{(SKIP)}
\]
\[
\{ P \Rightarrow P' \} \ S \ { Q' \} \quad Q' \Rightarrow Q \quad \text{(CONSEQ)}
\]
6. [20 pts.] Operational Semantics

Consider a simple expression language similar to the one used to illustrate formal operational semantics in Lecture 3b, where we gave rules for evaluation judgments of the form \((e, E, S) \Downarrow (v, S')\). Suppose we want to add a non-deterministic guarded-if expression to this language. For simplicity, we limit the expression to two arms; its syntax is \((\text{gifnz } e \ c \ e \ b \ c \ b)\). Informally, this expression is evaluated as follows: evaluate the guard conditions \(e \ c \ 1\) and \(e \ c \ 2\), in that order, to values \(v \ c \ 1\) and \(v \ c \ 2\); non-deterministically choose some \(i \in \{1, 2\}\) such that \(v \ c \ i\) is non-zero; evaluate the corresponding body \(e \ b \ i\) and yield its value as the result of the entire \(\text{gifnz}\) expression. If it is impossible to choose such an \(i\), the expression cannot be evaluated successfully.

Write down one or more formal operational evaluation rules that capture the semantics of \(\text{gifnz}\).