1. **True or False**

Mark each statement as either true or false.

(a) T F The typical compiler consists of several phases, including: lexing, parsing, transformation to an IR, optimization, and finally, code generation (in that order).

(b) T F The LLVM is a “Static Single Assignment” IR, which means that all local identifiers (e.g. %1, %x, etc.) are allocated on the stack.

(c) T F Other than the entry block, the order in which basic blocks appear within an LLVM function declaration does not affect the meaning of the program.

(d) T F An assembler determines the memory address of every piece of data in the source assembly language program.

(e) T F The HERA memory address space contains $2^{16}$ bytes.
2. Compilation and Intermediate Representations

In this problem, we consider different ways to compile boolean expressions to a very simplified LLVM IR target. A Haskell representation of the LLVM subset we use is given in Appendix B.

The appendix also gives a datatype of boolean-valued expressions (repeated below), which will constitute the source language for the purposes of this problem:

```haskell
data BoolExp
    = VarE String
    | TrueE
    | FalseE
    | NotE BoolExp
    | AndE BoolExp BoolExp
    | OrE BoolExp BoolExp
```

The usual way to compile such a boolean expression to an LLVM-like IR is to generate code that computes an answer, yielding the result in an operand. The function `compileBoolExpBlock`, given in the appendix, does exactly that; it relies on the helper `compileBoolExp` to do most of the hard work of “flattening” nested expressions into LL code. (Note that a source variable `VarE x` is compiled to a LL uid `ld x`, which would be pretty-printed as `%x`.)

(a) (Multiple Choice) Which of the following is the (pretty-printed) LL code that is obtained by compiling the following `example` via `compileBoolExpBlock`?

```
example :: BoolExp
example = NotE (OrE (VarE "a") (AndE (VarE "b") TrueE))
```

(i) %tmp2 = or i1 %a, %tmp1
%tmp1 = and i1 %b, %tmp2
%tmp0 = xor i1 false, %tmp1
ret i1 %tmp0

(ii) %tmp0 = xor i1 false, %a
%tmp1 = or i1 %b, %tmp0
%tmp2 = and i1 %tmp1, true
ret i1 %tmp2

(iii) %tmp2 = and i1 %b, true
%tmp1 = or i1 %a, %tmp2
%tmp0 = xor i1 false, %tmp1
ret i1 %tmp0

(iv) %tmp0 = and i1 %a, true
%tmp1 = or i1 %b, %tmp0
%tmp2 = xor i1 false, %tmp1
ret i1 %tmp2
Conditional Statements  Suppose that we want to use \textit{compileBoolExp} as a subroutine for compiling conditional statements:

\begin{verbatim}
if (b) { stmt1 } else { stmt2 }
\end{verbatim}

represented via the abstract syntax:  \textit{If b stmt1 stmt2}. Suppose \textit{b} is \textit{AndE (VarE "x") huge}, where \textit{huge} is a long, complicated boolean expression. A compiler that uses the \textit{compileBoolExp} from the appendix would generate code for the \textit{if} statement that looks like:

\begin{verbatim}
compileStmt (If b stmt1 stmt2) =

[...huge_code... %tmp0 = ...] ; could be very long (!)
%tmp1 = and i1 %x, %tmp0 ; materialize b into tmp1 (!)
br i1 %tmp1, label %then, label %else ; branch on b’s value (!)

%then:
[... stmt1 ...] ; code for stmt1

%else:
[... stmt2 ...] ; code for stmt2
\end{verbatim}

Note that LL code above “materializes” the value of \textit{b} into the operand %\textit{tmp1} and then does a conditional branch to either the block labeled %\textit{then} or the block labeled %\textit{else}, as appropriate.

Short Circuiting  We will now explore an alternate compilation strategy for boolean expressions that allows for short-circuit evaluation and is tailored for when the \textit{BoolExp} is the guard of an \textit{if} statement. In the example above, it is a waste of time to compute the whole boolean answer if \textit{x} happens to be false—in that case, we know just by looking at \textit{x} that we should jump immediately to the %\textit{else} branch; there’s no need to compute \textit{huge}.

This observation suggests that we can profit by creating a variant of \textit{compileBoolExp} called \textit{compileBoolExpBr} that, rather than computing a boolean value into an operand, instead takes in two labels (one for the “true” branch and one for the “false” branch) and emits code that jumps directly to the correct one, short circuiting the computation as soon as it’s clear which branch to take.

For example, consider the result of compiling the \textit{huge} example above, where we supply the \textit{then} and \textit{else} labels:

\begin{verbatim}
s ::= Stream
s = runUniqueM (compileBoolExpBr (AndE (VarE "x") huge) "then" "else")
\end{verbatim}

When we pretty-print \textit{s} as a sequence of LL blocks, we get the following code, which can replace the lines marked (!) in the code at the top of this page.

\begin{verbatim}
br i1 %x, label %lbl1, label %else ; if %x is false, jump to %else
%lbl1:
[...stream for huge...]
\end{verbatim}

Note that the compiler generated the intermediate label %\textit{lbl1}, which marks the start of the \textit{huge} code block. The stream for \textit{huge} will contain terminators that branch to %\textit{then} or %\textit{else} as appropriate. Remarkably, this compilation strategy will never need to emit instructions! A boolean expression can be completely compiled into a series of labels and terminators (branches or conditional branches).
(Multiple Choice) It is not too hard to work out what compileBoolExpBr should do by translating some small examples by hand. Mark the code that is a correct (pretty-printed) output for each call to compileBoolExpBr. There is only one correct answer for each case.

(b) \textit{compileBoolExpBr} (\textit{NotE} (\textit{VarE} "a")) "then" "else" =?

\begin{enumerate}[(i)]
\item \texttt{br i1 %a, label \%then, label \%else}
\item \texttt{br i1 %a, label \%else, label \%then}
\end{enumerate}

\begin{enumerate}[(iii)]
\item \texttt{br i1 %a, label \%else, label \%lbl0}
\item \texttt{br i1 %b, label \%else}
\end{enumerate}

\begin{enumerate}[(iv)]
\item \texttt{br i1 %a, label \%then, label \%lbl0}
\item \texttt{br i1 %b, label \%then}
\end{enumerate}

(c) \textit{compileBoolExpBr} (\textit{OrE} (\textit{VarE} "a") (\textit{VarE} "b")) "then" "else" =?

\begin{enumerate}[(i)]
\item \texttt{br i1 %a, label \%lbl0, label \%else}
\item \texttt{br i1 %b, label \%then, label \%else}
\end{enumerate}

\begin{enumerate}[(iii)]
\item \texttt{br i1 %a, label \%else, label \%lbl0}
\item \texttt{br i1 %b, label \%then, label \%else}
\end{enumerate}

\begin{enumerate}[(iv)]
\item \texttt{br i1 %a, label \%else, label \%lbl0}
\item \texttt{br i1 %b, label \%then, label \%else}
\end{enumerate}

(d) \textit{compileBoolExpBr} (\textit{AndE} (\textit{NotE} (\textit{VarE} "a")) (\textit{VarE} "b")) "then" "else" =?

\begin{enumerate}[(i)]
\item \texttt{br i1 %a, label \%lbl0, label \%else}
\item \texttt{br i1 %b, label \%then, label \%else}
\end{enumerate}

\begin{enumerate}[(iii)]
\item \texttt{br i1 %a, label \%else, label \%lbl0}
\item \texttt{br i1 %b, label \%then, label \%else}
\end{enumerate}

\begin{enumerate}[(iv)]
\item \texttt{br i1 %a, label \%then, label \%lbl0}
\item \texttt{br i1 %b, label \%then, label \%else}
\end{enumerate}
(e) Complete the implementation of `compileBoolExpBr` below by filling in each blank with the correct `Terminator` (for the first two blanks) or the correct `Label` (for the remaining blanks) to implement the short-circuiting code generator. Note that the `AndE` and `OrE` cases introduce a new `lmid` label, which labels the start of the second subexpression. We have done the `TrueE` case for you.

```
compileBoolExpBr :: BoolExp → Label → Label → UniqueM Stream
compileBoolExpBr b ltrue lfalse = case b of
  TrueE → pure [T (Br ltrue)]
  FalseE → pure [T (_______________________)]
  VarE s → pure [T (_______________________)]
  NotE c → compileBoolExpBr c (_______________________)
             (_______________________)
  AndE b1 b2 → do
    lmid ← newLabel "mid"
    p1 ← compileBoolExpBr b1 (_______________________)
                     (_______________________)
    p2 ← compileBoolExpBr b2 (_______________________)
                     (_______________________)
    pure (concat [p1,[L lmid],p2])
  OrE b1 b2 → do
    lmid ← newLabel "mid"
    p1 ← compileBoolExpBr b1 (_______________________)
                     (_______________________)
    p2 ← compileBoolExpBr b2 (_______________________)
                     (_______________________)
    pure (concat [p1,[L lmid],p2])
```
3. LLVM IR: Structured Data and *getelementptr*

Consider the following C structured datatypes and global variables array and node. RGB is a record with three fields and Node is a linked-list node that contains an inlined RGB value.

```c
struct RGB { int r; int g; int b;};
struct Node { struct RGB rgb; struct Node* next;};

struct RGB array[] = {{255,0,0}, {0,255,0}};
struct Node* node;
```

This program can be represented in LLVM as the following definitions:

```llvm
% RGB = type { i16, i16, i16 }
%Node = type { %RGB, %Node* }

@array = global [2 x %RGB] [%RGB { i16 255, i16 0, i16 0}, %RGB { i16 0, i16 255, i16 0}]
@node = global %Node ...
```

(Multiple Choice) For each LLVM instruction below, mark the corresponding C expression (i.e. such that a subsequent load from %ptr would get the value of the expression).

(a) %ptr = getelementptr %Node, %Node* @node, i32 0, i32 1
   □ node       □ @node->rgb       □ @node->next       □ @node->next->rgb

(b) %ptr = getelementptr [2 x %RGB], [2 x %RGB]* @array, i64 1, i64 0, i32 1
    □ &array[1][0][1] □ array[1].r □ array[1].g □ array[1].b

(c) %ptr = getelementptr %Node, %Node* @node, i32 0, i32 0, i32 2
    □ node.rgb[2]   □ node.rgb.b   □ node[2].rgb    □ node.next.rgb
Suppose that you wanted to compile an Haskell-like language into the LLVM IR. One issue is how to represent Haskell-style “algebraic datatypes”. Complete the following questions which prompt you to think about a strategy for representing the following Haskell type at the LLVM IR-level. (For the purposes of this question, ignore the fact that Haskell supports parameterized datatypes.)

\[
data \textit{Dat} = A \\
| B \textit{Int16} \\
| C \textit{Dat} \textit{Dat}
\]

(d) A Haskell value of type \textit{Dat} consists of a “tag” (e.g. \textit{A}, \textit{B}, or \textit{C}) and some associated data values (either an \textit{Int16} or a pair of references to \textit{Dat} values). Write down a (small) number of LLVM datatypes (including at least one for \textit{Dat}), that can together be used to represent values of type \textit{Dat}. Briefly justify your representation strategy.

%Dat = type 7

This allows for the representation of the \textit{Dat} type with a tag and associated data values using a combination of LLVM datatypes.
4. **HERA calling conventions**

Suppose we had a method of linking a HERA program with a C program. The linkage would depend on having a unified calling convention between the C code and the HERA code.

As some of you have noted, HERA lacks a modulus operator. So, we might write this operation in C, as follows:

```c
int modulus (int a, int b) {
    if (a < b)
        return a;
    else
        return modulus (a - b, b);
}
```

We assume that C ints are compiled into HERA words, that the `modulus` function lives at a label named `modulus`, and obeys the HERA calling conventions we have discussed.

Write a HERA function `divisibleBy3` that returns whether or not its one argument is divisible by 3. That is, the function returns 1 if the argument is divisible by 3 and returns 0 if it is not.

The first and last lines of the function are provided for you.

```hera
LABEL (divisibleBy3)
```

```hera
RETURN (fpAlt, pcRet)
```
Appendix

This Haskell module implements an IR that is a drastic simplification of the LLVM subset that you have been working with. In particular, all identifiers in this fragment name `i1` (i.e. boolean) values. It does not support function calls, mutable state (via alloca or pointers), datastructures, etc. However, it does support control flow blocks conditional branches, direct branches, and the `ret` instruction.

```haskell
-- Binary operations on boolean values
data BinaryOp = And | Or | Xor

-- Syntactic Values
data Operand
  = Const Bool -- only boolean constants!
  | Id Local -- a local identifier

-- Instructions
data Instruction
  = Binop BinaryOp Operand Operand

-- Block terminators
data Terminator
  = Ret Operand -- ret i64 %s
  | Br Label -- br label %lbl
  | Cbr Operand Label Label -- br i1 %s, label %l1, label %l2

-- Basic blocks
data Block = Bl { instructions :: [(Local, Instruction)], terminator :: Terminator }

-- Control Flow Graph: a pair of an entry block and a set labeled blocks
data Cfg = Cfg { entry :: Block,
  , blocks :: [(Label, Block)]
}

An abstract datatype of boolean-valued expressions:

```haskell
data BoolExp
  = VarE String
  | TrueE
  | FalseE
  | NotE BoolExp
  | AndE BoolExp BoolExp
  | OrE BoolExp BoolExp
```
Compile boolean expressions into LL code:

The function `compileBoolExp` compiles a boolean expression into a sequence of LL instructions (and the locals they define) and returns an operand that holds the result. Note that for any boolean value \( b \), the result of `false XOR b` is \( \neg b \) (logical negation). Function `compileBoolExpBlock` packages the results of compiling the expression as a basic block.

```haskell
-- Elements of an instruction stream
data Element = T Terminator | L Label
-- Streams:
-- A stream is just a sequence of elements
type Stream = [Element]

-- Create a fresh label:
newLabel :: String → UniqueM String
newLabel = newUniqueString

-- Compile a binary operation:
compileBinaryOp bop c1 c2 = do
  local ← newUniqueString "tmp"
  (c1_code, c1_opnd) ← compileBoolExp c1
  (c2_code, c2_opnd) ← compileBoolExp c2
  pure (c1_code ++ c2_code ++ [(local, Binop bop c1_opnd c2_opnd)], Id local)

-- Compile a BoolExp
compileBoolExp :: BoolExp → UniqueM ((Local, Instruction), Operand)
compileBoolExp TrueE = pure ([], Const True)
compileBoolExp FalseE = pure ([], Const False)
compileBoolExp (VarE s) = pure ([], Id s)
compileBoolExp (NotE c) = do
  local ← newUniqueString "tmp"
  (c_code, c_opnd) ← compileBoolExp c
  pure (c_code ++ [(local, Binop Xor (Const False) c_opnd)], Id local)

compileBoolExp (AndE c1 c2) = compileBinaryOp And c1 c2
compileBoolExp (OrE c1 c2) = compileBinaryOp Or c1 c2

-- Compile a BoolExp into a basic block
compileBoolExpBlock :: BoolExp → UniqueM Block
compileBoolExpBlock b = do
  (insns, opnd) ← compileBoolExp b
  pure (Bl { instructions = insns, terminator = Ret opnd })
```