

Name (printed): _____

Pennkey (login id): _____

My signature below certifies that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this examination.

Signature: _____ Date: _____

1	/14
2	/24
3	/21
4	/17
5	/14
Total	/90

- Do not begin the exam until you are told to do so.
- You have 80 minutes to complete the exam.
- There are 90 total points.
- There are 12 pages in this exam and a separate 4 page appendix.
- Make sure your name and Pennkey (a.k.a. Eniac username) is on the top of this page.
- Be sure to allow enough time for all the problems—skim the entire exam first to get a sense of what there is to do.

1. True or False (14 points)

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 It is possible for a compiler optimization to asymptotically improve the running time of a program (e.g. from $O(n^2)$ to $O(n)$.)
- c.** T F The regular expression $(a^*|b^*)^*$ recognizes the language of strings consisting of some number of a 's followed by the same number of b 's.
- d.** 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.
- e.** 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.
- f.** T F There are some X86lite programs that cannot be represented by well-formed LLVMlite control-flow graphs.
- g.** T F The linker and loader (as we saw in HW2) together determine the memory addresses represented by each label of the X86 assembly code.

2. Compilation and Intermediate Representations (24 points)

In this problem, we consider different ways to compile boolean expressions to a very simplified LLVM IR target. An OCaml 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:

```
type bexp =  
  | Var of string  
  | True  
  | False  
  | Not of bexp  
  | And of bexp * bexp  
  | Or of bexp * bexp
```

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 `compile_bexp_block`, given in the appendix does exactly that; it relies on the helper `compile_bexp` to do most of the hard work of “flattening” nested expressions into LL code. (Note that a source variable `Var x` is compiled to a LL uid `Id x`, which would be pretty-printed as `%x`.)

(a) (Multiple Choice, 3pts.) Which of the following is the (pretty-printed) LL code that is obtained by compiling the following example via `compile_bexp_block`?

```
let example : bexp = Not (Or (Var "a", And (Var "b", True)))
```

```
(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 `compile_bexp` as a subroutine for compiling conditional statements:

`if (b) { stmt1 } else { stmt2 }` represented via the abstract syntax: `If(b, stmt1, stmt2)`.

Suppose `b` is `And(Var "x", huge)`, where `huge` is a long, complicated boolean expression. A compiler that uses the `compile_bexp` from the appendix would generate code for the `if` statement that looks like:

```
compile_stmt (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
```

Note that LL code above “materializes” the value of `b` into the operand `%tmp1` and then does a conditional branch to either the block labeled `%then` or the block labeled `%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 `bexp` is the guard of an `if-then-else` statement. In the example above, it is a waste of time to compute the whole boolean answer if `x` happens to be false—in that case, we know just by looking at `x` that we should jump immediately to the `%else` branch; there’s no need to compute `huge`.

This observation suggests that we can profit by creating a variant of `compile_bexp` called `compile_bexp_br` 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 “huge” example above, where we supply the `then` and `else` labels:

```
let s : stream = compile_bexp_br (And(Var "x", huge)) "then" "else"
```

When we pretty-print `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.

```
br i1 %x, label %lb11, label %else ; if %x is false, jump to %else
%lb11:
  [...stream for huge...]
```

Note that the compiler generated the intermediate label `%lb11`, which marks the start of the `huge` code block. The stream for `huge` will contain terminators that branch to `%then` or `%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).

(There are no questions on this page.)

(Multiple Choice, 3pts. each) It is not too hard to work out what `compile_bexp_br` should do by translating some small examples by hand. Mark the code that is a correct (pretty-printed) output for each call to `compile_bexp_br`. There is only one correct answer for each case.

(c) `compile_bexp_br (Not(Var "a")) "then" "else" = ?`

(i) `br i1 %a, label %then, label %else`

(ii) `br i1 %a, label %else, label %then`

(iii) `br i1 %a, label %else, label %lbl0`
`lbl0:`
`br i1 %b, label %else`

(iv) `br i1 %a, label %then, label %lbl0`
`lbl0:`
`br i1 %b, label %then`

(d) `compile_bexp_br (Or(Var "a", Var "b")) "then" "else" = ?`

(i) `br i1 %a, label %lbl0, label %else`
`lbl0:`
`br i1 %b, label %then, label %else`

(ii) `br i1 %a, label %then, label %lbl0`
`lbl0:`
`br i1 %b, label %then, label %else`

(iii) `br i1 %a, label %else, label %lbl0`
`lbl0:`
`br i1 %b, label %then, label %else`

(iv) `br i1 %a, label %then, label %lbl0`
`lbl0:`
`br i1 %b, label %then, label %else`

(e) `compile_bexp_br (And(Not(Var "a"), Var "b")) "then" "else" = ?`

(i) `br i1 %a, label %lbl0, label %else`
`lbl0:`
`br i1 %b, label %then, label %else`

(ii) `br i1 %a, label %then, label %lbl0`
`lbl0:`
`br i1 %b, label %then, label %else`

(iii) `br i1 %a, label %else, label %lbl0`
`lbl0:`
`br i1 %b, label %then, label %else`

(iv) `br i1 %a, label %then, label %lbl0`
`lbl0:`
`br i1 %b, label %then, label %else`

(c) (12 points) Complete the implementation of `compile_bexp_br` 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 `And` and `Or` cases introduce a new `lmid` label, which labels the start of the second subexpression. We have done the `True` case for you.

```

type elt =
  | T of LL.terminator
  | L of LL.lbl

type stream = elt list

let rec compile_bexp_br (b:bexp) (ltrue:LL.lbl) (lfalse:LL.lbl) : stream =
  begin match b with

    | True  -> LL.[T (Br ltrue)]

    | False -> LL.[T (_____)]

    | Var s -> LL.[T (_____)]

    | Not c -> compile_bexp_br c (_____ ) (_____ )

    | And (b1, b2) ->
      let lmid = mk_tmp "lbl" in

      let p1 = compile_bexp_br b1 (_____ ) (_____ ) in

      let p2 = compile_bexp_br b2 (_____ ) (_____ ) in
      p1
      >@ LL.[L lmid]
      >@ p2

    | Or (b1, b2) ->
      let lmid = mk_tmp "lbl" in

      let p1 = compile_bexp_br b1 (_____ ) (_____ ) in

      let p2 = compile_bexp_br b2 (_____ ) (_____ ) in
      p1
      >@ LL.[L lmid]
      >@ p2
  end

```

3. LLVM IR: Structured Data and Getelementptr (21 points)

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 in-lined RGB value.

```
struct RGB {int64_t r; int64_t g; int64_t b;};
struct Node {struct RGB rgb; struct Node* next;};

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

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

```
%RGB = type { i64, i64, i64 }
%Node = type { %RGB, %Node* }
```

```
@array = global [2 x %RGB] [%RGB {i64 255, i64 0, i64 0}, %RGB {i64 0, i64 255, i64 0}]
@node = global %Node ...
```

(Multiple Choice, 2 pts. each) 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 OCaml-like language into the LLVM IR. One issue is how to represent OCaml-style “algebraic datatypes”. Complete the following questions which prompt you to think about a strategy for representing the following OCaml type at the LLVM IR-level. (For the purposes of this question, ignore the fact that OCaml supports generic datatypes.)

```
type d =  
  | B  
  | C of int64  
  | D of d * d
```

- d. (5 points) An OCaml value of type `d` consists of a “tag” (e.g. `B`, `C`, or `D`) and some associated data values (either an `int64` or a pair of references to `d` values). Write down a (small) number of LLVM datatypes (including at least one for `d`), that can together be used to represent values of type `d`. Briefly justify your representation strategy.

`%d = type -----`

- e. (5 points) An LLVM program that constructs the representation of the OCaml value `D(B, C(341))` as a stack-allocated object would need to use `alloca` (to allocate storage space), `getelementptr` (to compute struct offsets), and `store` (to write values into memory). It would also have to use the `bitcast` instruction (which, if you recall, converts one pointer type to any other pointer type). Briefly explain why.

- f. (5 points) In English and pseudocode, briefly explain what code you would expect to generate when compiling the following code, assuming that the variable x of type d is compiled according to the representation type you explained above.

```
let rec sum (x:d) =  
  match x with  
  | B -> 0  
  | C y -> y  
  | D (d1, d2) -> (sum d1) + (sum d2)
```

4. X86 Assembly and C calling conventions (17 points)

Recall that according to the x86-64 calling conventions that we have been using, the first argument to a function is passed in register `%rdi` and `%rbp` is a callee-save register and `%rsp` is caller-save. In X86lite syntax, the source operand comes before the destination, as in `movq src, dest`.

For this problem, consider the small C program and the accompanying X86 assembly code shown in Appendix B. The C code provides a `print_i64` function that prints its 64-bit value to the terminal. The main calls the function `foo`, which is implemented purely in assembly code as shown in `foo.s`. When these two files are compiled together, the resulting executable will print six numbers to the terminal.

(Multiple Choice, 3pts. each). There is only one correct answer for each case. Note that we have written small numbers in decimal and large numbers in hexadecimal.

Suppose that the first number printed to the terminal is `0x7fff5dd00990` (written in hexadecimal), which can be interpreted as a machine address. What will be printed by each of the five subsequent calls to `print_i64`?

- a. 17 `0x7fff5dd00990`
 42 `0x7fff5dd00988` (i.e. `0x7fff5dd00990 - 8`)
 341 `0x7fff5dd00980` (i.e. `0x7fff5dd00990 - 16`)

- b. 17 `0x7fff5dd00990`
 42 `0x7fff5dd00988` (i.e. `0x7fff5dd00990 - 8`)
 341 `0x7fff5dd00980` (i.e. `0x7fff5dd00990 - 16`)

- c. 17 `0x7fff5dd00990`
 42 `0x7fff5dd00988` (i.e. `0x7fff5dd00990 - 8`)
 341 `0x7fff5dd00980` (i.e. `0x7fff5dd00990 - 16`)

- d. 17 `0x7fff5dd00990`
 42 `0x7fff5dd00988` (i.e. `0x7fff5dd00990 - 8`)
 341 `0x7fff5dd00980` (i.e. `0x7fff5dd00990 - 16`)

- e. 17 `0x7fff5dd00990`
 42 `0x7fff5dd00988` (i.e. `0x7fff5dd00990 - 8`)
 341 `0x7fff5dd00980` (i.e. `0x7fff5dd00990 - 16`)

f. (2 pts.) Briefly explain what will happen if you delete lines 27 and 28 of `foo.s`, recompile, and then run the resulting executable.

5. Lexing, Parsing, and Grammars (14 points)

- a. Consider the following unambiguous context-free grammar over the symbols a and b that accepts *palindromes*—strings that read the same forward and backward. For example, a , bab , and $aabaa$ are all accepted by this grammar, whereas ab , aab , and $baaaba$ are not.

$$P ::= \begin{array}{l} \epsilon \\ a \\ b \\ aPa \\ bPb \end{array}$$

- i. (Yes or No) Is this grammar deterministically parsable by an LL(k) parser (for some k)? Briefly explain.
- ii. (Yes or No) Is this grammar deterministically parsable by an LR(k) parser (for some k)? Briefly explain.

- b. Consider the following unambiguous context-free grammar over the symbols a and b that accepts *balanced* strings in which every a is followed by a matching b. For example, ab, aabb, and aababb are all accepted, but ba, a, and aab are not.

$$Q ::= \begin{array}{l} \epsilon \\ QaQb \end{array}$$

- i. (Yes or No) Is this grammar deterministically parsable by an LL(k) parser (for some k)? Briefly explain.
- ii. (Yes or No) Is this grammar deterministically parsable by an LR(k) parser (for some k)? Briefly explain.
- c. (6 points) The following ambiguous grammar of mathematical expressions includes infix binary multiplication * and exponentiation operators ^ as well as tokens representing variables (ranged over by x).

$$E ::= \begin{array}{l} x \\ E * E \\ E ^ E \end{array}$$

Write down a disambiguated grammar that accepts the same set of strings, but makes multiplication left associative, exponentiation right associative, and gives exponentiation higher precedence than multiplication. For example, the token sequence $x * y ^ z ^ w * v * u$ would be parsed as though it had parentheses inserted like: $(x * (y ^ (z ^ w)) * v) * u$.

CIS341 Midterm 2017 Appendices

(Do not write answers in the appendices. They will not be graded)

Appendix A: i1 LL

This OCaml module implements an IR that is a drastic simplification of the LLVM subset that you have been working with in the course projects. 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.

```
module LL = struct
  type uid = string
  type lbl = string

  (* Binary operations on boolean values *)
  type bop = And | Or | Xor

  (* Syntactic Values *)
  type operand =
    | Const of bool      (* only boolean constants! *)
    | Id    of uid      (* a local identifier *)

  (* Instructions *)
  type insn =
    | Binop of bop * operand * operand

  (* Block terminators *)
  type terminator =
    | Ret of operand          (* ret i64 %s *)
    | Br  of lbl             (* br label %lbl *)
    | Cbr of operand * lbl * lbl (* br i1 %s, label %l1, label %l2 *)

  (* Basic blocks *)
  type block = { insns: (uid * insn) list; terminator: terminator }

  (* Control Flow Graph: a pair of an entry block and a set labeled blocks *)
  type cfg = block * (lbl * block) list
end
```

An abstract datatype of boolean-valued expressions:

```
type bexp =
| Var of string
| True
| False
| Not of bexp
| And of bexp * bexp
| Or of bexp * bexp
```

Compile boolean expressions into LL code:

The function `compile_bexp` compiles a boolean expression into a sequence of LL instructions (and the UIDs 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 `compile_bexp_block` packages the results of compiling the expression as a basic block.

```
let ( >@ ) x y = y @ x
let ( >:: ) x y = y :: x

let mk_tmp =
  let ctr = ref 0 in
  fun s -> let x = !ctr in ctr := x + 1; s ^ (string_of_int x)

(* compile a bexp *)
let rec compile_bexp (b:bexp) : ((LL.uid * LL.insn) list) * LL.operand =
  let open LL in
  let cmp_bin op c1 c2 =
    let opnd = mk_tmp "tmp" in
    let c1_code, c1_opnd = compile_bexp c1 in
    let c2_code, c2_opnd = compile_bexp c2 in
    c1_code >@ c2_code >@ [opnd, Binop(op, c1_opnd, c2_opnd)] , Id opnd
  in
  begin match b with
  | True -> [], Const true
  | False -> [], Const false
  | Var s -> [], Id s

  | Not c ->
    let opnd = mk_tmp "tmp" in
    let c_code, c_opnd = compile_bexp c in
    c_code >@ [opnd, Binop(Xor, Const false, c_opnd)] , Id opnd

  | And(c1, c2) -> cmp_bin And c1 c2
  | Or (c1, c2) -> cmp_bin Or c1 c2
  end
end

(* compile a bexp into a basic block *)
let compile_bexp_block b =
  let insns, opnd = compile_bexp b in
  LL.{ insns = List.rev insns; terminator = Ret opnd }
```

Append B: C and x86 Assembly Code

main.c:

```
#include <inttypes.h>
#include <stdio.h>

extern void foo();

void print_i64(int64_t x) {
    printf("%llx\n", x);
}

int main(int argc, char** argv) {
    foo();
    return 0;
}
```

foo.s:

```
1      .globl _foo
2  _foo:
3      pushq %rbp
4      movq %rsp, %rbp
5
6      /* print the current stack pointer value: 0x7fff5dd00990 */
7      movq %rsp, %rdi
8      callq _print_i64
9
10     movq $341, %rdi      /* part (a) */
11     callq _print_i64
12
13     pushq $42           /* part (b) */
14     movq (%rsp), %rdi
15     callq _print_i64
16
17     pushq $17          /* part (c) */
18     movq %rsp, %rdi
19     callq _print_i64
20
21     movq 8(%rsp), %rdi  /* part (d) */
22     callq _print_i64
23
24     leaq 8(%rsp), %rdi  /* part (e) */
25     callq _print_i64
26
27     movq %rbp, %rsp
28     popq %rbp
29     retq
```