CIS 341 Midterm March 5, 2015

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: _____

SOLUTIONS

1. Representing Structured Data (14 points)

This problem is about the representation of various source language data structures (from a C or OCaml) as they might be translated to the LLVM-lite IR we are using for the course projects.

a. Consider the following LLVM IR types:

```
(a)        { i64, i64 }

(b)        { i64, i64 }*

(c)        { i64*, i64* }*

(d)        [ 4 x { i64, i64 } ]*

(e)        { i64, [ 0 x i64 ] }*

(f)        [ 14 x i8 ]
```

For each of the following source language constructs, indicate which one of the above LLVM IR types could best be used as its translation type, or write "none" if none of the types will work. Each answer is one of `(a)`–`(f)`.

i. *Answer:* `(f)` The C string global constant data `"hello, world!\00"`.

ii. *Answer:* `(b)` The variable `p` in the C program below

```
struct point {int64_t x; int64_t y}

void foo() {
  struct point p;
}
```

iii. *Answer:* `(d)` The variable `p` in the C program below

```
struct point {int64_t x; int64_t y}

void foo() {
  struct point p[] = {{1,2}, {3,4}, {5,5}, {6,7}};
}
```

iv. *Answer:* `(e)` An OCaml-style length-indexed array `x` declared as:

```
let x : int64 array = mkarray (4)
```

v. *Answer:* `(c)` The argument `x` in the polymorphic (a.k.a. generic) OCaml function:

```
fun (x:'a*'b) -> fst x
```

(*Hint:* Clients of this function might need to use `Bitcast`.)

b. Suppose that the LLVM local `%ptr` has type `%T*` where `%T = { i64, i64*, %T* }`

i. What is the type of `%q` in the following program? (Write "ill typed" if it does not typecheck.)

```
%q = getelementptr %T* %ptr, i32 0, i32 1
```

Answer: `i64**`

ii. What is the type of `%r` in the following program? (Write "ill typed" if it does not typecheck.)

```
%r = getelementptr %T* %ptr, i32 0, i32 2, i32 1
```

Answer: ill typed

2. Lexing, Parsing, and Grammars (18 points total)

Consider the following grammar of expressions for a language that has C-style preincrement `++e`, postincrement `e++`, and Haskell-style infix string concatenation `e1 ++ e2`, as well as identifiers and parentheses. To the right is an OCaml datatype for representing the abstract syntax.

$$
\begin{array}{rcl}
exp & ::= & \\
 & | & id \\
 & | & \texttt{++}exp \\
 & | & exp\texttt{++} \\
 & | & exp \texttt{ ++ } exp \\
 & | & \texttt{(}exp\texttt{)}
\end{array}
$$

```
type exp =
  | Id of string
  | Pre of exp        (* preincrement *)
  | Pst of exp        (* postincrement *)
  | Cat of exp * exp (* concatenation *)
```

a. (3 points) This is a *highly* ambiguous grammar. Demonstrate that this is the case by giving *three* distinct parses of the token sequence `++ a ++ ++ b`. Write your answer using the OCaml notation for the abstract syntax. (You can abbreviate identifiers: write `a` for `Id("a")`, etc.)

Answer: Any three of:

```
Cat(Pst(Pre(a)), b)
Cat(Pre(Pst(a)), b)
Cat(Pre(a), Pre(b))
Pre(Cat(Pst(a), b))
Pre(Cat(a, Pre(b)))
```

b. (4 points) This grammar cannot be disambiguated simply by assigning one associativity and precedence level to the `++` token (because it is used in multiple incompatible ways). One way to disambiguate it by hand is observe that any sequence of `++` tokens and "atomic" expressions (such as identifiers or parenthesized subexpressions) can be written "right associatively":

- Any initial prefix of `++` tokens is considered to be applications of the `Pre` operator.
- The remaining token sequence is broken into a non-zero number of "segments", each of which consists of an atomic expression followed by `++` tokens, which are uses of the `Pst` operator.
- If there are multiple segments, they are separated by exactly one `++` token, which is considered as a right-associative use of `Cat`.

For example, the top sequence below would be uniquely parsed as shown by the parenthesized bottom version, where `a`, `b`, `c`, and `d` are atomic:

```
++ ++ ++ a ++ b ++ ++ ++ c ++ ++ d ++ ++
++ ++ ++ ( a ++ ((b ++ ++) ++ ((c ++) ++ (d ++ ++ ))))
```

It turns out that sequences of the form described above can be recognized using a *regular expression* over the alphabet $\{\texttt{++}, \texttt{E}\}$ (where `E` stands for an "atomic" expression). For your reference, Appendix A includes a description of the form of regular expressions for use in this question.

Write down a regular expression that matches *exactly* these sequences.

Answer:

$$\texttt{++}^*(\texttt{E++}^*)\texttt{++})^*\texttt{E++}^*$$

c. (3 points) According to the description in part (b) above, what are the relative precedence levels of the three different operators in the disambiguated grammar? Write one of `High`, `Medium`, or `Low` next to each as appropriate:

preincrement `++e`: `Low`

postincrement `e++`: `High`

concatenation `e ++ e`: `Medium`

d. (8 points) Because regular languages are included in LR(1) languages, we know that it is possible to write down an unambiguous context-free grammar that parses the *exp* grammar. Write down such a grammar below, where the initial symbol is `e0`. Use the four terminal tokens `++`, `(`, `)`, `ID` and feel free to add as many nonterminals as necessary.

```
e0:
 | ++ e=e0
 | e1

e1:
 | e2 ++ e1
 | e2

e2:
 | e2 ++
 | e3

e3:
 | ID
 | ( e )
```

3. LLVM IR (15 points)

This problem refers to the C source function `m` and its corresponding (unoptimized) LLVM code, which are found in Appendix B. You may *carefully* tear off that page to use as a reference.

a. (3 points) LLVM IR semantics. Consider the call to `m` in the following instruction:

```
%ans = call i64 @m(i64 -3, i64 5)
```

i. Circle the value of `%6` during this execution of `@m`

[-3] 1 3 5 a pointer to 3 a pointer to -3 a pointer to 5

ii. Circle the value of `%9` during this execution of `@m`

-3 1 [3] 5 a pointer to 3 a pointer to -3 a pointer to 5

iii. Circle the value of `%2` at line 18 during this execution of `@m`

-3 1 3 5 a pointer to 3 a pointer to -3 [a pointer to 5]

b. (5 points) Observe that in the source definition of `m`, the argument `b` is unmodified, and its address is never taken. These facts suggest that we can optimize the LLVM code to avoid using `alloca` for `b` and instead work with the temporary `%b` directly.

Perform that optimization by hand, and complete the template below to show the resulting code, which should be semantically equivalent to the original program. There are no changes necessary in the block with label `5`, and we have given you the last parts of each of the remaining blocks:

Answer:

```
define i64 @m(i64 %a, i64 %b) {
  %1 = alloca i64
  store i64 %a, i64* %1
  %3 = load i64* %1
  %4 = icmp slt i64 %3, 0
  br i1 %4, label %5, label %8

5:
  %6 = load i64* %1
  %7 = sub i64 0, %6
  store i64 %7, i64* %1
  br label %8

8:
  %9 = load i64* %1
  %11 = mul i64 %9, %b
  ret i64 %11
```

c. (3 points) If we naïvely try to perform the optimization described in part (b) to the other argument `a` of `m` (leaving `b` alone) , we might end up with the following *incorrect* LLVM IR program:

```
define i64 @m(i64 %a, i64 %b) {
  %2 = alloca i64
  store i64 %b, i64* %2
  %4 = icmp slt i64 %a, 0
  br i1 %4, label %5, label %8

5:
  %a = sub i64 0, %a        ; <-- flaw
  br label %8

8:
  %10 = load i64* %2
  %11 = mul i64 %a, %10
  ret i64 %11
}
```

In one sentence, describe the flaw indicated in the program above.

Answer: It breaks the SSA invariant that each local is defined exactly once.

d. (4 points) Suppose we extend the LLVM IR with a *conditional move* command that sets `%id` to one of `op2`, if `op1` is `true`, or `op3`, if `op1` is `false`:

```
%id = select i1 op1, i64 op2, i64 op3
```

Use this instruction to implement an optimized version of `@m` that needs only four instructions (other than the terminating `ret`):

Answer:

```
define i64 @m(i64 %a, i64 %b) {
  %1 = icmp slt i64 %a, 0
  %2 = sub  i64 0, %a
  %3 = select i1 %1, i64 %2, i64 %a
  %4 = mul  i64 %3, %b
  ret i64 %4
}
```

4. Compilation and Calling Conventions (15 points)

Recall that according to the x86-64 calling conventions that we have been using, the first two quad-sized arguments to a function are passed in registers `%rdi` and `%rsi`, respectively, and the return value is returned in `%rax`. `%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`.

Consider the following LLVM IR program, in which `bar` calls `foo`.

```
define i64 @foo(i64 %x, i64 %y) {
  %ans = add i64 %x, %y
  ret i64 %ans
}

define i64 @bar() {
  %1 = call i64 @foo(i64 12, i64 34)
  ret i64 %1
}
```

a. (6 points) For each of the following possible X86lite implementations of `bar`, indicate whether it is "correct" (*i.e.* that it correctly implements the calling conventions), or write "incorrect" and briefly explain (in the margins) why it is wrong.

```
(A)
bar:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $8, %rsp
  movq  $12, %rax
  movq  %rax, %rdi
  movq  $34, %rax
  movq  %rax, %rsi
  callq foo
  movq  %rax, -8(%rbp)
  movq  -8(%rbp), %rax
  movq  %rbp, %rsp
  popq  %rbp
  retq
```

```
(B)
bar:
  pushq %rbp
  movq  %rsp, %rbp
  movq  $12, %rdi
  movq  $34, %rsi
  callq foo
  movq  %rbp, %rsp
  popq  %rbp
  retq
```

```
(C)
bar:
  movq  $12, %rdi
  movq  $34, %rsi
  callq foo
  retq
```

Answer: They are all correct.

b. (6 points) A *tailcall* at the LLVM IR level is a use of a call instruction followed *immediately* by a return of the result of the call (or a void return if the called function has void return type). For example, the function `bar` from part (a) of this problem makes a tailcall to `foo`, since `bar` immediately returns `%1`, which is `foo`'s result.

A tailcall in which all of the function arguments fit in registers can be optimized to just a `jmp`—there is no need to use the x86 `callq` or `retq` instructions at all. Tailcall optimization effectively turns a recursive function into a loop! It reduces the need for $O(n)$ stack space to $O(1)$, where n is the recursion depth. However, some care must still be taken to follow the calling conventions. For each of the following possible x86-lite tailcall-optimized implementations of `bar`, indicate whether it is "correct" (*i.e.* that it correctly implements the calling conventions), or write "incorrect" and briefly explain (in the margins) why it is wrong.

```
(A)
bar:
  pushq  %rbp
  movq   %rsp, %rbp
  movq   $12, %rdi
  movq   $34, %rsi
  jmp    foo
```

```
(B)
bar:
  movq   $12, %rdi
  movq   $34, %rsi
  jmp    foo
```

```
(C)
bar:
  movq   $12, %rdi
  movq   $34, %rsi
  popq   %rbp
  jmp    foo
```

Answer: `(B)` is correct. `(A)` and `(C)` violate callee-save for `%rbp` because when `bar` is called they don't restore it on exit.

c. (3 points) According to the x86 calling conventions we've been following, function arguments seven and higher are pushed to the stack, which is cleaned up by the caller after the callee returns. Briefly explain how this interacts with tailcall optimization.

Answer: Caller-cleanup arguments *prevent* tailcall optimization because the control never returns to the caller, and so the stack pointer isn't restored properly. (Caller-cleanup arguments *do* support functions with a variable number of arguments, like C's `printf`, however. Such var-arg functions are harder to support with callee-cleanup calling conventions.)

5. Scope Checking (18 points)

In this problem we will consider scope checking (a simple subset of) OCaml programs. The grammar for this subset of OCaml is given by the syntactic categories below:

exp	::=		Expressions
	|	$x \mid f$	variables and function names
	|	int	integer constants
	|	exp_1 `+` exp_2	arithmetic
	|	$exp_1\ exp_2$	function application
	|	`let` $x = exp_1$ `in` exp_2	local lets
	|	`(`exp`)`	parentheses (used only for the concrete syntax)

$prog$	::=		Programs
	|	`;;`exp	answer expression
	|	`let` $x\ = exp\ \ prog$	top-level declarations
	|	`let` $f\ x\ = exp\ \ prog$	top-level, one-argument function declarations

Scoping contexts for this language consist of comma-separated lists of variable and function names:

`G`	::=		Scoping Contexts
	|	$\cdot$	empty context
	|	x`,G`	add x to the context
	|	f`,G`	add f to the context

Scope checking for expressions is defined by the following inference rules, which use judgments of the form $\boxed{\texttt{G} \vdash exp}$. Recall that the notation $x \in \texttt{G}$ means that x occurs in the context list `G`.

$$\frac{x \in \texttt{G}}{\texttt{G} \vdash x}\ [\textsc{VarX}] \qquad \frac{f \in \texttt{G}}{\texttt{G} \vdash f}\ [\textsc{VarF}] \qquad \frac{}{\texttt{G} \vdash int}\ [\textsc{Int}] \qquad \frac{\texttt{G} \vdash exp_1 \quad \texttt{G} \vdash exp_2}{\texttt{G} \vdash exp_1 + exp_2}\ [\textsc{Add}]$$

$$\frac{\texttt{G} \vdash exp_1 \quad \texttt{G} \vdash exp_2}{\texttt{G} \vdash exp_1\ exp_2}\ [\textsc{App}] \qquad \frac{\texttt{G} \vdash exp_1 \quad x, \texttt{G} \vdash exp_2}{\texttt{G} \vdash \texttt{let}\ x = exp_1\ \texttt{in}\ exp_2}\ [\textsc{Let}]$$

Scope checking for programs is defined by these three inference rules, which use judgments of the form $\boxed{\texttt{G} \vdash prog}$.

$$\frac{\texttt{G} \vdash exp}{\texttt{G} \vdash \texttt{;;}\ exp}\ [\textsc{Exp}] \qquad \frac{\texttt{G} \vdash exp \quad x, \texttt{G} \vdash prog}{\texttt{G} \vdash \texttt{let}\ x = exp\ prog}\ [\textsc{LetX}] \qquad \frac{x, \texttt{G} \vdash exp \quad f, \texttt{G} \vdash prog}{\texttt{G} \vdash \texttt{let}\ f\ x = exp\ prog}\ [\textsc{LetF}]$$

A program $prog$ is considered to be well-scoped exactly when it is possible to derive the judgment in the empty context: $\cdot \vdash prog$

a. Consider the following program:

```
let f x = 341
let g y = (let z = f y in z) + y
;; g 3
```

It is well-scoped, as shown by the following (partial) derivation that has some missing sub-derivations as marked by names D1?, D2?, and D3?.

$$\dfrac{\text{D1?} \qquad \dfrac{\text{D3?} \qquad \dfrac{\text{D2?}}{\texttt{g}, \texttt{f}, \cdot \vdash \texttt{;; g 3}}\,[\textsc{Exp}]}{\texttt{f}, \cdot \vdash \texttt{let g y = (let z = f y in z) + y ;; g 3}}\,[\textsc{LetF}]}{\cdot \vdash \texttt{let f x = 341 let g y = (let z = f y in z) + y ;; g 3}}\,[\textsc{LetF}]$$

i. (3 points) Which of the following derivation trees belongs in the spot marked D1?

a. $\dfrac{\texttt{x} \in \texttt{x}, \cdot}{\texttt{x}, \cdot \vdash \texttt{x}}\,[\textsc{VarX}]$ b. $\dfrac{}{\cdot \vdash \texttt{341}}\,[\textsc{Int}]$ c. $\boxed{\dfrac{}{\texttt{x}, \cdot \vdash \texttt{341}}\,[\textsc{Int}]}$ d. $\dfrac{}{\texttt{f}, \cdot \vdash \texttt{341}}\,[\textsc{Int}]$

ii. (3 points) Which of the following derivation trees belongs in the spot marked D2?

a. *Answer:* THIS ONE:

$$\dfrac{\dfrac{\texttt{g} \in \texttt{g}, \texttt{f}, \cdot}{\texttt{g}, \texttt{f}, \cdot \vdash \texttt{g}}\,[\textsc{VarF}] \qquad \dfrac{}{\texttt{g}, \texttt{f}, \cdot \vdash \texttt{3}}\,[\textsc{Int}]}{\texttt{g}, \texttt{f}, \cdot \vdash \texttt{g 3}}\,[\textsc{App}]$$

b.

$$\dfrac{\dfrac{\texttt{g} \in \texttt{g}, \cdot}{\texttt{g}, \cdot \vdash \texttt{g}}\,[\textsc{VarF}] \qquad \dfrac{}{\texttt{g}, \cdot \vdash \texttt{3}}\,[\textsc{Int}]}{\texttt{g}, \cdot \vdash \texttt{g 3}}\,[\textsc{App}]$$

c.

$$\dfrac{\dfrac{\texttt{g} \in \texttt{f}, \texttt{g}, \cdot}{\texttt{f}, \texttt{g}, \cdot \vdash \texttt{g}}\,[\textsc{VarF}] \qquad \dfrac{}{\texttt{f}, \texttt{g}, \cdot \vdash \texttt{3}}\,[\textsc{Int}]}{\texttt{f}, \texttt{g}, \cdot \vdash \texttt{g 3}}\,[\textsc{App}]$$

d.

$$\dfrac{\dfrac{\texttt{g} \in \texttt{y}, \texttt{g}, \texttt{f}, \cdot}{\texttt{y}, \texttt{g}, \texttt{f}, \cdot \vdash \texttt{g}}\,[\textsc{VarF}] \qquad \dfrac{}{\texttt{y}, \texttt{g}, \texttt{f}, \cdot \vdash \texttt{3}}\,[\textsc{Int}]}{\texttt{y}, \texttt{g}, \texttt{f}, \cdot \vdash \texttt{g 3}}\,[\textsc{App}]$$

iii. (12 points) In the space below, draw the missing derivation tree marked by `D3?`. Be sure to label the uses of each inference rule as in the tree above.

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{f} \in \texttt{y},\texttt{f},\cdot}{\texttt{y},\texttt{f},\cdot \vdash \texttt{f}}\,[\textsc{VarF}] \quad \dfrac{\texttt{y} \in \texttt{y},\texttt{f},\cdot}{\texttt{y},\texttt{f},\cdot \vdash \texttt{y}}\,[\textsc{VarX}]}{\texttt{y},\texttt{f},\cdot \vdash \texttt{f y}}\,[\textsc{App}] \quad \dfrac{\texttt{z} \in \texttt{z},\texttt{y},\texttt{f},\cdot}{\texttt{z},\texttt{y},\texttt{f},\cdot \vdash \texttt{z}}\,[\textsc{VarX}]}{\texttt{y},\texttt{f},\cdot \vdash \texttt{let z = f y in z}}\,[\textsc{Let}] \quad \dfrac{\texttt{y} \in \texttt{y},\texttt{f},\cdot}{\texttt{y},\texttt{f},\cdot \vdash \texttt{y}}\,[\textsc{VarX}]}{\texttt{y},\texttt{f},\cdot \vdash \texttt{(let z = f y in z) + y}}\,[\textsc{Add}]$$

1 Appendix A: Regular Expressions

Regular expressions over an alphabet $\mathcal{A}$ are defined by the following grammar:

$rexp$	$::=$		
	$\mid$	ϵ	empty string
	$\mid$	a	$a \in \mathcal{A}$ alphabet symbol
	$\mid$	$rexp\ rexp$	sequential concatenation
	$\mid$	$rexp^*$	Kleene-star (zero or more repetitions)
	$\mid$	$rexp \mid rexp$	alternative choice
	$\mid$	$(rexp)$	

2 Appendix B: LLVM

A C program that computes the `a`'s absolute value times `b`.

```
int64_t m(int64_t a, int64_t b) {
  if (a < 0) {
    a = -a;
  }
  return a * b;
}
```

Corresponding LLVM IR coded, created using `clang` with no optimizations turned on:

```
define i64 @m(i64 %a, i64 %b) {
  %1 = alloca i64
  %2 = alloca i64
  store i64 %a, i64* %1
  store i64 %b, i64* %2
  %3 = load i64* %1
  %4 = icmp slt i64 %3, 0          ; 'slt' is signed less than
  br i1 %4, label %5, label %8

5:
  %6 = load i64* %1
  %7 = sub i64 0, %6
  store i64 %7, i64* %1
  br label %8

8:
  %9 = load i64* %1
  %10 = load i64* %2
  %11 = mul i64 %9, %10
  ret i64 %11
}
```