CIS 341 Midterm February 28, 2013

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	/10
2	/20
3	/25
4	/20
5	/25
Total	/100

- Do not begin the exam until you are told to do so.
- You have 80 minutes to complete the exam.
- There are 100 total points.
- There are 12 pages in this exam.
- 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 (10 points)

a. T F In X86 assembly, the instruction `Jmp eax` will cause the program counter regiser (`eip`) to be set to the contents of the `eax` register.

b. T F We could remove the `Call` and `Ret` instructions from the X86lite subset we've been using for class projects without changing the expressiveness of the language.

c. T F It is possible to write a regular expression that matches that set of strings consisting of well-balanced parentheses.

d. T F The regular expression `(`ϵ `+ b + bb)(a + ab + abb)*` accepts all strings of `a`'s and `b`'s with at most two consecutive occurrences of `b`. (Here + indicates alternative choice.)

e. T F LR(k) grammars cannot be right recursive.

f. T F There is no such thing as a shift/shift conflict for a LR parser.

g. T F In an LR(0) parser state, the item $S \longmapsto (.L)$ indicates that nonterminal L is at the top of the stack because L is to the right of the dot.

h. T F Choosing integer tags as a representation for OCaml datatype variants or C and Java-style `enum`s to be sequential and dense (for example just the integers $0 \ldots N$) allows `match` and `switch` compilation to use efficient jump tables.

i. T F In C, the compiler lays out a 2D array with statically known dimensions (like `int M[3][4]`) so that all of the elements are contiguous in memory.

j. T F When compiling C for 32-bit x86 and word aligned fields,

```
sizeof(struct{char a; char b; int c})
```

returns 6 (bytes).

2. Lexing, Parsing, and Grammars (20 points)

a. Consider the following *ambiguous* context-free grammar for the language of regular expressions, where R is the only nonterminal and the terminals are taken from the set $\{\text{`}c\text{'}, *, +, (, ), \epsilon\}$. Here `'`$c$`'` stands for a collection of character-carrying tokens, where the character is c. (We use + rather than | for alternation to avoid confusion with the $|$ symbol used on the grammar.)

$$R ::= \epsilon \mid \text{`}c\text{'} \mid R+R \mid R* \mid RR \mid (R)$$

We might implement the datatype of abstract syntax trees for this grammar using the following OCaml code:

```
type rexp =
 | Eps                    (*  ε     *)
 | Char of char           (*  'c'   *)
 | Alt of rexp * rexp     (*  R+R   *)
 | Star of rexp           (*  R*      *)
 | Seq of rexp * rexp     (*  RR    *)
```

i. (6 points) Demonstrate that this grammar is ambiguous by giving two different abstract syntax trees (OCaml values of type `rexp`) that might be generated by parsing the input sequence:

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

ii. (8 points) Write down the context-free grammar obtained by disambiguating the language above so that all operators associate to the left and $*$ has higher precedence than sequence, which has higher precedence than +.

b. (6 points) Recall that a *palindrome* is a sequence that reads the same forward and backward: `A` and `ABBA` and `BABAB` are all palindromes but `AB` is not. Write down a context free grammar that recognizes all (and only) the palindromes over the two tokens $\{\texttt{A}, \texttt{B}\}$. Your grammar is allowed to be ambiguous. Use the nonterminal S as the start symbol.

$S \quad ::=$

3. LLVM IR & Intermediate code generation (25 points)

In this problem we consider the LLVM intermediate representation. For your reference, the Appendix at the end of the exam defines the grammar for the subset of the LLVM IR we have been using (so far) in the course.

a. (4 points) A collection of LLVM Lite IR blocks with an entry point `entry` must satisfy a number of invariants for it to be well formed (i.e. "make sense") as a program. For example, each `%uid` that is used as an operand in an instruction must be defined before it is used.

What is the *static single assignment* invariant relative to the LLVM IR subset given in the appendix?

b. (6 points) What value will be returned by running the following LLVM code, starting from the label `entry`?

```
entry:
  %t1 = alloca
  store 3, %t1
  %t2 = load %t1
  %t3 = load %t1
  %t4 = mul %t2, %t3
  store %t4, %t1
  %t5 = icmp eq %t2, %t3
  br %t5, label %then, label %else
```

```
then:
  %t6 = alloca
  %t7 = load %t1
  %t8 = mul %t7, %t7
  store %t8, %t6
  %t9 = load %t6
  store %t9, %t1
  br label %else
```

```
else:
  %t10 = load %t1
  ret %t10
```

c. (15 points) Some languages have a `do { `$stmt$` } while(`exp`)` construct that executes $stmt$ once and then repeats it until the guard expression exp becomes false. In this problem, we investigate how to compile this statement form to LLVM IR.

Suppose we have (most of the) implementations of compilation judgments:

$$C \vdash [\![exp]\!] = (\text{operand} * \text{stream})$$
$$C \vdash [\![stmt]\!] = \text{stream}$$

Here, a stream is a sequence of labels, LLVM instructions, and terminators that can be broken up into basic blocks. (You need not distinguish these cases with constructors.)

How do you translate the `do–while` construct to LLVM? That is, give (OCaml-like) pseudo code that says how you would implement the rule below. Indicate which labels and LLVM uids should be freshly generated.

$C \vdash [\![$`do { `$stmt$` } while (`exp`)`$]\!] =$

4. X86 Assembly Programming (20 points)

Consider the following C function:

```
void foo(int x, int* y) {
  int a = x + (*y);
  *y = a;
}
```

Recall that `int* y` declares `y` to be an `int` pointer and that `*y` accesses the contents pointed to by `y`. The `gcc` compiler (in 32-bit only mode and without optimizations) produces the following X86 assembly code, which is in our X86lite subset and follows `cdecl` calling conventions:

```
_foo:
        pushl       %ebp
        movl        %esp, %ebp                  ; %ebp set here
        subl        $12, %esp
        movl        12(%ebp), %eax
        movl        8(%ebp), %ecx
        movl        %ecx, -4(%ebp)
        movl        %eax, -8(%ebp)
        movl        -8(%ebp), %eax
        movl        (%eax), %eax
        movl        -4(%ebp), %ecx
        addl        %ecx, %eax
        movl        %eax, -12(%ebp)
        movl        -8(%ebp), %eax
        movl        -12(%ebp), %ecx
        movl        %ecx, (%eax)
        addl        $12, %esp
        popl        %ebp
        ret
```

a. (4 points) The caller of this function places argument `x` at which (indirect offset) memory location? (Relative to the value in `%ebp` after the line marked above.)

a. `-12(%ebp)` b. `-8(%ebp)` c. `-4(%ebp)` d. `8(%ebp)` e. `12(%ebp)`

b. (4 points) The local variable `a` resides at which (indirect offset) memory location? (Relative to the value in `%ebp` after the line marked above.)

a. `-12(%ebp)` b. `-8(%ebp)` c. `-4(%ebp)` d. `8(%ebp)` e. `12(%ebp)`

c. (4 points) How much memory does the stack frame used by `_foo` in this code take up in bytes? Include the saved return address and base pointer, and any stack space allocated for local storage, but *not* the space allocated by the caller for the function arguments.

a. 8 bytes b. 16 bytes c. 20 bytes d. 24 bytes e. 28 bytes f. 32 bytes

d. (8 points) Which of the following optimized versions could replace the body `_foo:` and still be correct with respect to the C program and `cdecl` calling conventions? Mark *all* that are correct—there may be more than one.

i.
```
_foo:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        movl    8(%ebp), %ecx
        addl    %ecx, (%eax)
        popl    %ebp
        ret
```

ii.
```
_foo:
        pushl  %ebp
        movl   %esp, %ebp
        movl   12(%ebp), %eax
        addl   8(%ebp), %eax
        movl   %ebp, %esp
        popl   %ebp
        ret
```

iii.
```
_foo:
        movl 4(%esp), %eax
        addl 8(%esp), %eax
        ret
```

iv.
```
_foo:
        movl 4(%esp), %ebx
        movl 8(%esp), %eax
        addl %ebx, (%eax)
        ret
```

5. Scope Checking (25 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
	|	$exp_2\ exp_2$	function application
	|	`let` $x = exp_1$ `in` exp_2	local lets
	|	`(`exp`)`	

$prog$	::=		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 a judgment in the empty context: $\cdot \vdash prog$

a. Consider the following program:

```
let x = 3
let f y = let z = 3 + y in z + x
;; f x
```

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{y},\texttt{x},\cdot \vdash 3}\,[\textsc{Int}] \quad \dfrac{\texttt{y} \in \texttt{y},\texttt{x},\cdot}{\texttt{y},\texttt{x},\cdot \vdash \texttt{y}}\,[\textsc{VarX}]}{\texttt{y},\texttt{x},\cdot \vdash \texttt{3 + y}}\,[\textsc{Add}] \quad \dfrac{\dfrac{\texttt{z} \in \texttt{G?}}{\texttt{G?} \vdash \texttt{z}}\,[\textsc{VarX}] \quad \dfrac{\texttt{x} \in \texttt{G?}}{\texttt{G?} \vdash \texttt{x}}\,[\textsc{VarX}]}{\texttt{G?} \vdash \texttt{z + x}}\,[\textsc{Add}]}{\texttt{y},\texttt{x},\cdot \vdash \texttt{let z = 3 + y in z + x}}\,[\textsc{Let}] \quad \texttt{D2?}}{\texttt{x},\cdot \vdash \texttt{let f y = let z = 3 + y in z + x} \;\; \texttt{;; f x}}\,[\textsc{LetF}]$$

$$\dfrac{\texttt{D1?} \qquad \texttt{x},\cdot \vdash \texttt{let f y = let z = 3 + y in z + x} \;\; \texttt{;; f x}}{\cdot \vdash \texttt{let x = 3} \;\; \texttt{let f y = let z = 3 + y in z + x} \;\; \texttt{;; f x}}\,[\textsc{LetX}]$$

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. $\dfrac{}{\cdot \vdash \texttt{3}}\,[\textsc{Int}]$ c. $\dfrac{}{\texttt{x},\cdot \vdash \texttt{3}}\,[\textsc{Int}]$ d. $\dfrac{\texttt{x} \in \cdot}{\cdot \vdash \texttt{x}}\,[\textsc{VarX}]$

ii. (3 points) What is the missing context G?

a. z, x, · b. y, x, · c. x, y, z, · d. z, y, x, ·

iii. (8 points) In the space below, draw the missing derivation tree marked by D2? . Be sure to include the names of the rules as in the tree above.

b. (5 points) The following program is syntactically well-formed but ill-scoped—one of the $x \in \texttt{G}$ checks fails for some variable x and some context `G`. What are they? (Be careful with the ordering of variables in the context!)

```
let a = 3
let f1 b = b + a
let f2 c = (let d = f1 3 in f1 a) + (f1 d)
;; f2 (f1 a)
```

x = _____

`G` = _____

c. (6 points) The LETF rule above does not allow the function to be recursive. Write down the premises of the rule needed to allow a well-scoped function body to mention the function name itself:

$$\frac{}{\texttt{G} \vdash \texttt{let}\, f\ x = exp\ prog}\ [\textsc{LetRecF}]$$

Appendix

LLVM Lite IR

A subset of the LLVM IR that we have used in Project 3:

$$
\begin{array}{rcl}
op & ::= & \texttt{\%uid} \mid constant \\
\\
bop & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{shl} \mid ... \\
\\
cmpop & ::= & \texttt{eq} \mid \texttt{ne} \mid \texttt{slt} \mid \texttt{sle} \mid ... \\
\\
insn & ::= & \\
 & \mid & \texttt{\%uid} = bop\ op1, op2 \\
 & \mid & \texttt{\%uid} = \texttt{alloca} \\
 & \mid & \texttt{\%uid} = \texttt{load}\ op1 \\
 & \mid & \texttt{store}\ op1, op2 \\
 & \mid & \texttt{\%uid} = \texttt{icmp}\ cmpop\ op1, op2 \\
\\
terminator & ::= & \\
 & \mid & \texttt{ret}\ op \\
 & \mid & \texttt{br}\ op\ \texttt{label \%lbl1}, \texttt{label \%lbl2} \\
 & \mid & \texttt{br label \%lbl} \\
\\
block & ::= & \texttt{lbl:} \\
 & & \quad insn_1 \\
 & & \quad \ldots \\
 & & \quad insn_n \\
 & & \quad terminator
\end{array}
$$

An LLVM Lite IR *program* is a collection of blocks such that all labels mentioned in the terminators of those blocks are included among the labels of the blocks themselves. One label called `entry` is the designated entry point of the program. A `%uid` is an operand identifier and `%lbl` is a label identifier.