Name: ____________________________

CIS 341 Midterm
2 March 2011

1	/10
2	/10
2	/18
3	/14
4	/18
Total	/70

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 10 pages in this exam.
- Make sure your name is on the top of this page.

1. True or False (10 points)

a. T F It is possible to write a regular expression that describes exactly the language consisting of all strings of balanced parentheses.

b. T F A left-recursive grammar cannot be implemented by an LL(k) parser for any k.

c. T F One big advantage of using an intermediate representation is that it makes the compiler easier to port to different target architectures.

d. T F OCaml's representation of the type `(int array) array` is more likely to incur performance penalties (due to caching and other hardware implementation techniques) than C's representation of an array declared by `int x[][]`.

e. T F Control may enter a basic block at more than one location.

f. T F When control leaves a basic block, there is at most one possible next instruction to be executed.

g. T F To generate efficient representations of structured data, the compiler must take into account the machine word size and alignment constraints of the target platform.

h. T F The closure produced for the function returned by the following OCaml program would necessarily contain an environment that maps `x` to 3:

```
let x = 3 in
let y = 4 + x in
  fun (z:int) -> y + z
```

i. T F Hoisting is the process used when compiling a functional programming language to bring closed code to the top level.

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

2. Parsing (10 points)

Consider the following grammar for the untyped lambda calculus, in which E is the only nonterminal and the terminal tokens are taken from the set $\{\texttt{var}_x, \texttt{fun}, \texttt{->}, \texttt{(}, \texttt{)}\}$. Here $\texttt{var}_x$ stands for a collection of string-carrying variable identifier tokens, where the string is x. In the concrete syntax, the programmer would just write a string like *foo*, which is represented by the token $\texttt{var}_{foo}$

$$E \ ::= \ \texttt{var}_x \mid E\ E \mid \texttt{fun}\ \texttt{var}_x\texttt{->}\ E \mid \texttt{(}E\texttt{)}$$

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

```
type exp =
 | Var of string
 | App of exp * exp
 | Fun of string * exp
```

a. (4 points) Demonstrate that this grammar is ambiguous by giving two different abstract syntax trees (OCaml values of type `exp`) that might be generated by parsing the input sequence:
`fun` x `->` x x.

b. (6 points) Write down the context-free grammar obtained by disambiguating the language above so that function application associates to the left and has higher precedence than "$\texttt{fun var}_x\texttt{->}$", which you can think of as a unary operator on expressions. For example, the following two inputs should yield identical parse trees:

`fun` x `->` x x x and `fun` x `->((`x x`)` x`)`

3. Intermediate code generation

Consider the following statement language, which simplifies the one used in Project 3 (by eliminating `for` loops and unmatched `if` statements).

stmt	::=		
	|	*lhs* `=` *exp* `;`	
	|	`if (`*exp*`)` *stmt* `else` *stmt*	
	|	`while (`*exp*`)` *stmt*	
	|	`{` *block* `}`	
block	::=	*vdecls stmts*	*stmts* is a list of zero or more *stmt* elements

Suppose we want to add a primitive form of *local exception handling*, similar to (but much simpler than) the kinds of exceptions found in ML or Java. The idea is to extend statements with two new constructs:

stmt	::=	...
	|	`fail;`
	|	`try` *stmt* `with` *stmt*

The intended semantics of `fail` is that it terminates the current (possibly nested) block of code and immediately transfers control to the `with` branch of the nearest lexically enclosing `try` statement. The statement "`try` s_1 `with` s_2" evaluates s_1, and, if no `fail` exception is raised, s_2 is skipped and the program continues as usual. Such `try` statements may nest (`fail` jumps to the *nearest* enclosing `try`'s `with`), and the `with` branch might itself `fail` (assuming there is an outer enclosing `try`).

For the purposes of this problem, the grammars of expressions (given by *exp*) and variable declarations (given by *vdecls*) are not relevant, because neither one includes any form of statement (and hence cannot invoke `fail`).

a. (6 points) Given the operational semantics described above, what value is returned by each of the following programs?

```
/* program A */
int x = 0;
try {
  int y = 1;
  fail;
  x = x + y;
} with
  x = x + 2;
return x;
```

```
/* program B */
int x = 0;
try {
  int y = 1;
  try {
    x = x + y;
  } with {
    x = x + 2;
    fail;
  }
} with
  x = x + 4;
return x;
```

```
/* program C */
int x = 0;
try {
  int y = 1;
  try {
    x = x + y;
    fail;
  } with {
    x = x + 2;
    fail;
  }
} with
  x = x + 4;
return x;
```

- Program A returns:

- Program B returns:

- Program C returns:

b. (12 points) Recall that one way of translating statements to the control-flow IL used in Project 3 is to implement a function `compile_stmt` that takes a context and a statement and returns a pair containing the modified context and a suitable IL-level instruction stream with labeled jump targets. Using OCaml-like pseudo code, the case for compiling `while` loops might look like this:

```
compile_stmt ctxt (s:stmt) : ctxt * stream =
  begin match s with
  | While(exp_guard, stmt_body) ->
    let (ret_exp, code_exp, ctxt_exp) = compile_exp ctxt exp_guard in
    let (ctxt_out, code_body) = compile_stmt ctxt_exp stmt_body in
    (ctxt_out,
     [ __lpre:
          code_exp
          If (ret_exp != 0) __lbody __lpost
       __lbody:
          code_body
          Jump __lpre
       __lpost:  ])
  | ...
  end
```

Briefly describe the changes you would need to make to the `compile_stmt` function to correctly translate `fail` and `try` statements to the IL. Write down the cases (at the level of OCaml pseudo code as in the example for `while` above) for `fail` and `try`. Your translation should raise an error if it encounters a `fail` statement that is not contained within at least one `try`. (Use the back of this page for more space, if necessary.)

4. X86 Assembly Programming

Consider the following C function:

```
int foo(int x, int w) {
  int y = x;
  return y + w;
}
```

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
    subl  $24, %esp
    movl  8(%ebp), %eax
    movl  %eax, -12(%ebp)
    movl  12(%ebp), %eax
    addl  -12(%ebp), %eax
    movl  %ebp, %esp
    popl  %ebp
    ret
```

a. (2 points) The local variable y resides at which (indirect offset) memory location?

a. 8(%ebp)   b. -12(%ebp)   c. 12(%esp)   d. 12(%ebp)

b. (2 points) The function argument w resides at which (indirect offset) memory location?

a. 8(%ebp)   b. -12(%ebp)   c. 12(%esp)   d. 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 needed by function arguments.

a. 16 bytes   b. 24 bytes   c. 32 bytes   d. 40 bytes

d. (6 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:`

```
movl   12(%ebp), %eax
addl   8(%ebp), %eax
movl   %ebp, %esp
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. **Type Checking**

Recall the simply-typed functional language we studied in class:

Abstract syntax of types:

$$\texttt{T} \ ::= \ \texttt{int} \mid \texttt{T -> T}$$

Abstract syntax of expressions:

e	::=	i	*integer constants*
	|	x	*variables*
	|	e + e	*addition*
	|	fun (x:T) -> e	*functions*
	|	e e	*application*

As a reminder, here are the typing rules for this language (the rule names are written $[Rule]$):

$$\frac{}{\texttt{E} \vdash i : \texttt{int}}\ [Int] \qquad \frac{\texttt{x}:\texttt{T} \in \texttt{E}}{\texttt{E} \vdash \texttt{x} : \texttt{T}}\ [Var] \qquad \frac{\texttt{E} \vdash \texttt{e}_1 : \texttt{int} \quad \texttt{E} \vdash \texttt{e}_2 : \texttt{int}}{\texttt{E} \vdash \texttt{e}_1 \texttt{ + } \texttt{e}_2 : \texttt{int}}\ [Add]$$

$$\frac{\texttt{E}, \texttt{x}:\texttt{T}_1 \vdash \texttt{e} : \texttt{T}_2}{\texttt{E} \vdash \texttt{fun (x:T}_1\texttt{) -> e} : \texttt{T}_1 \texttt{ -> } \texttt{T}_2}\ [Fun] \qquad \frac{\texttt{E} \vdash \texttt{e}_1 : \texttt{T}_1 \texttt{ -> } \texttt{T}_2 \quad \texttt{E} \vdash \texttt{e}_2 : \texttt{T}_1}{\texttt{E} \vdash \texttt{e}_1\ \texttt{e}_2 : \texttt{T}_2}\ [App]$$

a. (8 points) Complete the following derivation tree:

$$\frac{}{\vdash \texttt{fun (x:int) -> x + ((fun (y:int) -> x + y) 3)} : \texttt{int -> int}}\ [Fun]$$

b. (10 points) Consider extending the language with a ML-style option types (a specific instance of ML's more general datatypes). There are three new expression forms:

`e`	::=	`...`	*stuff from before*
	|	`None`	*Empty option*
	|	`Some e`	*Non-empty option*
	|	`match e with None -> e	Some x -> e`	*Case analysis*

To typecheck options, we add a new form of types:

$$\texttt{T} ::= \ldots \mid \texttt{T option}$$

Operationally, these options behave just as those in ML—you can "tag" any value with `Some` to indicate the presence of the optional value and use the "tag" `None` to indicate its absence. The `match` expression checks the tag and branches to the appropriate case, binding the tagged value to the variable `x` if needed. Note that in the `Some x -> e` branch of the pattern match, the variable `x` is in scope inside `e`.

Complete the typing rules for these new constructs (note that the return types in the conclusions are missing—you should fill them in):

$$\frac{}{\texttt{E} \vdash \texttt{None} :}$$

$$\frac{}{\texttt{E} \vdash \texttt{Some e} :}$$

$$\frac{}{\texttt{E} \vdash \texttt{match } \texttt{e}_1 \texttt{ with None -> } \texttt{e}_2 \texttt{ | Some x -> } \texttt{e}_3 :}$$