Name: _____

CIS 341 Midterm
October 20, 2008

1	/10
2	/10
3	/10
4	/10
5	/10
Total	/50

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 7 pages in this exam.
- Make sure your name is on the top of this page.

1. Parsing

Consider the following grammar for OCaml-style types in which S is the only nonterminal and the terminal tokens are taken from the set $\{$`int`, `bool`, `*`, `->`, `(`, `)`$\}$.

$$S \ ::= \ \texttt{int} \mid \texttt{bool} \mid S \texttt{ * } S \mid S \texttt{ -> } S \mid \texttt{(}S\texttt{)}$$

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

```
type ast =
 | Int
 | Bool
 | Pair of ast * ast
 | Arrow of ast * ast
```

a. Demonstrate that this grammar is ambiguous by giving two different abstract syntax trees (OCaml values of type `ast`) that might be generated by parsing the input sequence: `int * int -> bool`.

b. Write down the context-free grammar obtained by disambiguating the language above so that: `*` associates to the left, `->` associates to the right, and `->` has lower precedence than `*`.

2. Intermediate code generation

Recall that our translation for `while` loops to the control-flow IL was:

$[\![$`while (`e_1`)` $e_2$$]\!]$ =
```
  __lpre:
    If([[e1]] != 0) __lbody __lpost
  __lbody:
    [[e2]];
    Jump __lpre
  __lpost:
```

Suppose we wanted to add support for C or Java-style commands `break` and `continue`. Remember that `break` terminates a loop body early (jumping to the exit point) and that `continue` stops the current iteration, but returns to the top of the loop.

How would you modify the compilation function $[\![e]\!]$ to handle these new features? Show the translations for `while`, `continue`, and `break` in the style shown above. Hint: think about how we we compiled "short circuit" boolean expressions.

3. Calling Conventions

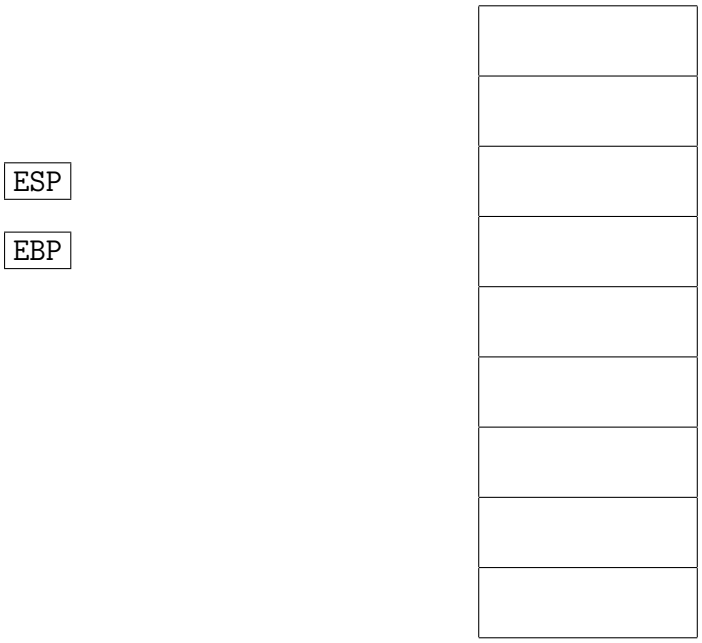
Consider the following C program:

```
int f(int x, int y) {
  int z = x + y;
  /* <--- here */
  return z * z
}

int main() {
  return f(341, 42);
}
```

Recall that on X86, ESP is the "stack pointer" and EBP is the "base pointer". Assume that the C compiler stack-allocates the local variable z and otherwise follows X86 C calling conventions. Fill in the words of memory below (each box is one word) to show the stack as it looks when the program reaches the point marked "here". Use meaningful symbolic labels for any memory addresses that might appear in the stack. Also indicate (using arrows) where in the stack the registers ESP and EBP point. Note: you may not need all of the memory locations provided.

Lower addresses



Higher addresses

4. Closure Conversion

Recall that a closure is a pair where the first element is a data structure representing the environment and the second item is a pointer to code that takes an environment and a function argument and evaluates the closure's body. The following program contains three points marked (1), (2), and (3) where closures will be built at runtime.

```
(1)fun x ->
  let w = x * x in
  (2)fun y -> ((3)fun z -> x + z + y) w)
```

a. Assuming that environments are represented as lists, what environment will be encapsulated by the closure at each point? To help you out, below you will find a set of possible environments.

(1) Environment =

(2) Environment =

(3) Environment =

Possible environment lists:

A. `[]`	B. `[x; z]`	C. `[x; x]`	D. `[x; w]`
E. `[y; z]`	F. `[x; y; z]`	G. `[x; w; y]`	H. `[x; w; y; z]`

b. The code encapsulated in the closure at point (3) has the form `Code(env, z, <body>)`. Which of the following is the correct implementation of `<body>`?

A.
```
x + z + y
```

B.
```
let x = nth 0 env in
let w = nth 1 env in
  ((fun z -> x + z + y) w)
```

C.
```
let x = nth 0 env in
let y = nth 1 env in
let z = nth 2 env in
  x + z + y
```

D.
```
let x = nth 0 env in
let w = nth 1 env in
let y = nth 2 env in
  x + z + y
```

E.
```
let x = nth 0 env in
let w = nth 1 env in
let y = nth 2 env in
  ((fun z -> x + z + y) w)
```

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*
	$\mid$	`x`	*variables*
	$\mid$	`e + e`	*addition*
	$\mid$	`fun (x:T) -> e`	*functions*
	$\mid$	`e e`	*application*

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

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

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

a. Complete the following derivation tree:

$$\dfrac{\dfrac{}{\vdash \texttt{3} : \texttt{int}}\;[Int]}{\vdash \texttt{3 + ((fun (x:int) -> x + 2) 5)} : \texttt{int}}\;[Add]$$

b. Consider extending the language with a simple exception mechanism. There are two new expressions:

`e`	::=	...	*stuff from before*
	|	`failwith e`	*raise an exception carrying integer* `e`
	|	`try e handle(x) => e`	*exception handler*

The expression "`failwith e`" can appear anywhere in a program. Its argument, `e`, is an integer that is carried by the exception to the nearest enclosing exception handler. (If there is no enclosing handler, the program aborts with error code `e`.)

The expression "`try` $\texttt{e}_1$ `handle(x) =>` $\texttt{e}_2$" runs $\texttt{e}_1$. If $\texttt{e}_1$ raises no exception, the result of the the `try` is just the result of $\texttt{e}_1$. Otherwise, when $\texttt{e}_1$ raises an exception carrying integer i, the result of the `try` is the result of evaluating $\texttt{e}_2$ with `x` bound to the value i, i.e. x's scope is the expression $\texttt{e}_2$.

As an example, here are three well-typed programs:

```
3 + (failwith (2 + 5))
```

```
(fun (f:int->int) -> f 3) (failwith 4)
```

```
try (3 + (failwith (2 + 5))) handle(x)=> x + x
```

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

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

$$\frac{}{\texttt{E} \vdash \texttt{try } \texttt{e}_1 \texttt{ handle(x) => } \texttt{e}_2 :}$$