Midterm II
Sample Solution

- Please read all instructions (including these) carefully.
- There are 4 questions on the exam, each worth 20 or 25 points. You have 3 hours to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: SAMPLE

SID or SS#: _____

Problem	Max points	Points
1	20	
2	25	
3	20	
4	25	
TOTAL		

1. **Miscellaneous Short Answer** (20 points)

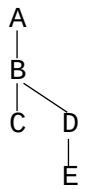
(a) Circle each occurrence of a free identifier in the following expression. (Type declarations are deliberately omitted and are unnecessary for this problem.)

```
let x = (let y = z * x in y + 2) in let y = x * z in let x = x in x
                  X   X                            X
```

Each free identifier is marked with 'X'.

(b) To speed up inherits-from tests in a compiler a (single) inheritance hierarchy can be specified by a matrix M of booleans, with $\texttt{M}[\texttt{Class}_1, \texttt{Class}_2] = \texttt{1}$ if and only if $\texttt{Class}_1 \leq \texttt{Class}_2$. Consider the following matrix (indexed first by rows and then by columns). Note that this matrix is incomplete (it is missing some entries):

$\leq$	A	B	C	D	E
A	1*				
B	1	1*			
C	1*	1	1*		
D	1*	1		1*	
E	1*	1*		1	1*



Each new entry is marked with '*'.

Draw the inheritance tree specified by this matrix (use the space to the right of the matrix). In addition, fill in missing "1" entries of the matrix. Your solution should add as few new "1" entries as possible.

(c) Assume that Cool is modified so that inherited methods cannot be redefined. How can the implementation of dynamic dispatch described in lecture be simplified?

If methods cannot be redefined, then all method dispatches can be statically determined; dispatch would consist of a simple jump to a fixed location, rather than a table lookup followed by a register jump.

2. **Polymorphic Types** (25 points)

For all parts of this question, use the following programming language

$$\texttt{E} ::= \texttt{id} \mid \texttt{(E E)} \mid \texttt{(E + E)} \mid \texttt{(fun x E)}$$

and the following type language

$$\texttt{t} ::= \texttt{t} \rightarrow \texttt{t} \mid \alpha \mid \beta \mid \ldots \mid \texttt{int} \mid \texttt{char} \mid \texttt{float}$$

where $\alpha, \beta, \ldots$ are type variables.

(a) Give a most general parametric polymorphic type for the following expression:

$$\texttt{((fun x x)(fun y (fun z z)))}$$

```
A -> B -> B where A and B are type variables.
```

(b) Give an expression that has the following type.

$$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

```
(fun x x) has the type as one of its instances.
Another example is (fun x (fun y (x y))).
```

(c) An instance of a polymorphic type is *ground* if it has no type variables. Give a grammar for the ground instances of the type

$$\alpha \rightarrow \beta$$

```
S ::= T -> T
T ::= int | char | float | T -> T
```

(d) This question explores the combination of parametric polymorphism and overloading. Let + have the types `int` $\times$ `int` $\rightarrow$ `int` and `float` $\times$ `float` $\rightarrow$ `float`. Outline an algorithm for computing the set of possible polymorphic types for an expression when + is overloaded. You need only give a clear sketch of the algorithm, you do not need to write out type rules for each construct. Do not worry about efficiency—keep your algorithm as simple as possible.

Recall that function overloading means that a function has a finite set of types. Every use of the function must be assigned one of those types in such a way that the entire program type checks. This suggests the following simple and inefficient algorithm.

```
possible_types(e) =
  let S = { t |  |- e : t  with some choice of type
                           int x int -> int  or  float x float -> float
                           for each use of + in e }
  return S
```

where the typechecking of e is carried out using the usual polymorphic inference algorithm.

3. **Subtyping and Overloading** (20 points)

This question explores the combination of subtyping and overloading. Java allows method declarations to be overloaded within a class. If `e : Z` and class `Z` has more than one `f` method, then $\texttt{e.f}(\texttt{e}_1, \ldots, \texttt{e}_n)$ is type checked as follows. Let $T_1, \ldots, T_n$ be the types for expressions $\texttt{e}_1, \ldots, \texttt{e}_n$. Now choose a method `f` in class `Z` such that

(a) the definition of `f` has n formal parameters with types $S_1, \ldots, S_n$,

(b) $T_i \leq S_i$ for $1 \leq i \leq n$,

(c) for any other `f` method in class `Z` with formal parameter types $U_1, \ldots, U_n$ such that $T_i \leq U_i$, it is the case that $S_i \leq U_i$ for $1 \leq i \leq n$.

If there is not a unique method satisfying (a)-(c), then the dispatch expression does not type check. Consider the following fragment of Cool code with overloaded methods:

```
class A {
 f(x: A) :  B   { f() };

 f(x: B) :  C   { new C };

 f(x: C) :  B*  { f(self) };

 f() :  B   { new B };
};

class B  {
 f(x : A) :  C*  { x.f(x.f(x)) };

 f(x : B) :  C*  { (x.f()).f(x) };

 f() :  C*  { self.f(new A) };
}

class C {
  f() :  C*  { f(new B) };

  f(x : B) :  C*  { f(self).f(self) };

  f(x : C) :  C   { x };
}
```

```
The types above marked with '*' can optionally be ERROR because of the
mistake in the original problem statement (they must all be ERROR together).
```

(a) Fill in the return types of each of the methods in the space provided.

(b) Which of the three conditions for overloading resolution listed above are required for type soundness and why? (In other words, could any condition(s) be dropped and still have a sound type system?)

Condition (a) is required since dispatch is ill-defined when the wrong number of actual paramaters is supplied. Condition (b) is required since actual paramaters need to conform to the types of the formals in order to guarantee the method will not break.

Condition (c) can be dropped. The intent of (c) is to pick the "best" from a possible set of methods that satisfy (a) and (b). If (c) is removed, more programs will fail to type check (since there will be times that (a) and (b) generate more than one method), but type _soundness_ will not be violated, since there is still no possibilty of the type checker approving a program which has a run-time type error.

(c) Java does not have SELF_TYPE. However, there are no problems in combining overloading resolution with the SELF_TYPE rules for Cool. Briefly explain why this is the case.

SELF_TYPE cannot be used as the type of a formal parameter in Cool. Since the overloading discussed in the problem is argument overloading, the inclusion of SELF_TYPE does not affect overloading resolution.

4. **Attribute Grammars and Code Generation** (25 points)

Consider the following programming language:

$$\begin{array}{lcl} \texttt{D} & ::= & \texttt{id(id,...,id)} = \texttt{E; D} \mid \epsilon \\ \texttt{E} & ::= & \texttt{id} \mid \texttt{int} \mid \texttt{E} + \texttt{E} \mid \\ & & \texttt{if E} = \texttt{E then E else E} \mid \texttt{id(E,...,E)} \end{array}$$

All values in this language are 32-bit integers. A function call inside a function body B is *tail recursive* if it is the "last" computation that a function body performs—i.e., the value of B is the value of the call. For example, in the function

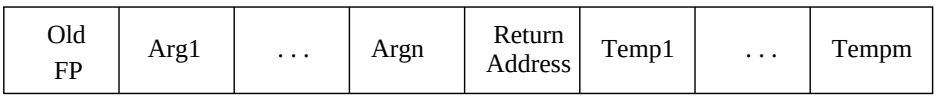
$$\texttt{f(x)} = \texttt{if f(x} - \texttt{1)} = \texttt{0 then f(x} - \texttt{2) else 0}$$

the call `f(x-1)` is not tail recursive and the call `f(x-2)` is tail recursive. In general, the following function calls are *not* tail recusive: calls inside an addition expression, inside the predicate of an if-then-else, and inside expressions for actual arguments of other function calls. A function call in any other position *is* tail recursive.

(a) Give an attribute grammar that assigns boolean attribute `E.tailrec` "true" if `E` is a tail-recursive function call and "false" if `E` is a non-tail-recursive function call.

```
One inherited attribute, tailrec, is needed.

D -> id(id,...,id) = E; D          E.tailrec=true
  |  /* empty */

E -> id
  |  int
  |  E_1 + E_2                     E_1.tailrec=false
                                   E_2.tailrec=false

E -> if E_1 = E_2 then E_3 else E_4  E_1.tailrec=false
                                   E_2.tailrec=false
                                   E_3.tailrec=E.tailrec
                                   E_4.tailrec=E.tailrec

E -> id(E_1,...,E_n)               E_1.tailrec=false
                                    ...
                                   E_n.tailrec=false
```

(b) Assume that activation records are layed out as follows for this language:



Assume that caller/callee responsibilities are divided as discussed in lecture; in particular, the caller stores the frame pointer and actual arguments of the AR, and the callee saves the return address and allocates space for temporaries. The caller-side code for a normal call to $\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_n)$ is:

```
cgen(f(e1,...,en)) =
 sw $fp 0($sp)      | store frame pointer
 addiu $sp $sp -4   |
 cgen(e1)           | evaluate and
 sw    $a0 0($sp)   | store actual parameter #1
 addiu $sp $sp -4   |
 ...
 cgen(en)           | evaluate and
 sw    $a0 0($sp)   | store actual parameter #n
 addiu $sp $sp -4   |
 jal   f_entry      | jump to function entry
```

Special code can be generated for tail-recursive function calls. If `g` makes a tail recursive call to `f`, then instead of pushing a new AR on the stack for `f`, the call to `f` can reuse (overwrite) the AR for `g`. Note that this works because `g` would immediately return after `f` returns anyway.

Show how to generate code for a tail-recursive function call to $\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_n)$ by modifying the code for a standard function call above. Indicate clearly all changes to the code, including which instructions are deleted, modified, or added (and where). Use any compile-time constants you need, but be sure to define what they are (you do not need to show how to compute them).

```
cgen( f(e1, ..., en), argc ) =          | tail-rec call
   | --- do not store our $fp, since we are not pushing a new A.R.
   | --- evaluate arguments as usual, since they may depend on values
   |     in the caller's A.R.
   cgen(e1)                             | evaluate and
   sw    $a0 0($sp)                     | store actual parameter #1
   addiu $sp $sp -4                     |
    ...
   cgen(en)                             | evaluate and
   sw    $a0 0($sp)                     | store actual parameter #n
   addiu $sp $sp -4                     |
   | --- now that arguments are evaluated, copy them down to
   |     overwrite the old A.R.
   | --- reload our return address before it is overwritten!
   lw    $ra ((-4)*(argc + 1))($fp)     | reload our return address
   lw    $a0 (4*(n - 1 + 1))($sp)       | copy actual #1
   sw    $a0 ((-4)*1)($fp)              |
```

```
 ...
lw    $a0 (4*(n - n + 1))($sp)      | copy actual #n
sw    $a0 ((-4)*n)($fp)             |
addiu $sp $fp (-4)*(n + 1)          | set $sp to point beyond frame
j f_entry                           | jump to function entry
```