Final Examination

- Please read all instructions (including these) carefully.
- You may not discuss the contents of the exam with anyone who has not taken the exam until after 8:30 p.m. Wednesday.
- There are 8 questions on the exam, each worth between 15 and 30 points. You have 3 hours to work on the exam, so you should plan to spend approximately 22 minutes on each question.
- 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. There are no "tricky" problems on the exam—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.

NAME: Sample solution

SID or SS#: _____

Problem	Max points	Points
1	15	
2	20	
3	20	
4	30	
5	20	
6	20	
7	30	
8	30	
TOTAL	185	

1. **Regular Expressions** (15 points)

   Suppose we have a language with persistent objects that can be located anywhere on a network. The name of an object is similar to names used on the WWW, with the form

   ```
   [<class>:] [//<address>/] <path>
   ```

   The notation above is interpreted as follows:

   - square brackets `[]` delimit optional parts of a name;
   - angle brackets `< >` delimit patterns, described below;
   - an identifier is a sequence of one or more lower case letters, capital letters, digits, and the characters '_', '$', and '-', not beginning with a digit or '-';
   - `<class>` is an identifier;
   - `<address>` is a list of one or more identifiers, separated by '.';
   - `<path>` is a list of one or more identifiers, separated by '/'.

   Write a regular expression describing object names. You may use any of the regular expression operations given in lecture and flex abbreviations.

   Using flexish notation:

   ```
   (({id}:) + epsilon) ((//{address}/) + epsilon) {path}
   ```

   where

   ```
   path = {id} (/{id})*

   address = {id} (.{id})*

   id = [a-zA-Z_$][-a-zA-Z_$0-9]*
   ```

2. **Code Generation** (20 points)

Consider the following fragment of assembly code produced by `coolc`:

```
Main.f:
        addiu   $sp $sp -16
        sw      $fp 16($sp)
        sw      $s0 12($sp)
        sw      $ra 8($sp)
        addiu   $fp $sp 4
        move    $s0 $a0
        lw      $a0 20($fp)
        sw      $a0 0($fp)
        lw      $a0 24($fp)
        jal     Object.copy
        lw      $t1 0($fp)
        lw      $t2 12($a0)
        lw      $t1 12($t1)
        add     $t1 $t1 $t2
        sw      $t1 12($a0)
        lw      $fp 16($sp)
        lw      $s0 12($sp)
        lw      $ra 8($sp)
        addiu   $sp $sp 28
        jr      $ra
```

Give Cool source that generates this code (you need not give a complete Cool program, just a fragment that produces this assembly code).

```
class Main is
  f(x: Int, y: Int, z: Object) is y + x end;
end;

(the type of z is arbitrary)
```

3. **Parsing and Attribute Grammars** (20 points)

Consider the following proposal to add arrays to Cool. The following examples declare `a` to be an array of 5 elements of type `Int` and `b` to be a 10×10 array of `Bool`s.

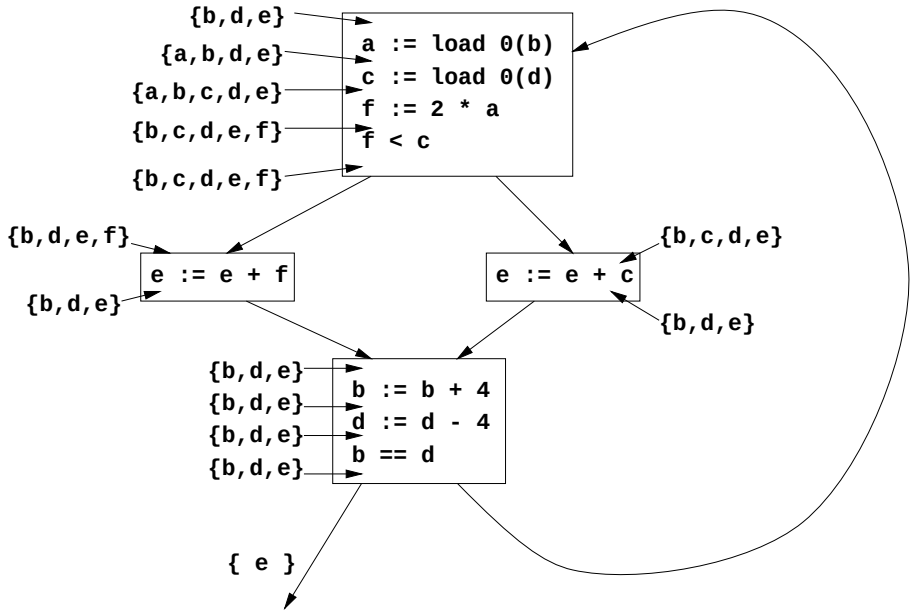
```
a[5]: Int
b[10][10]: Bool
```

In general, an array declaration has an identifier, one or more dimensions, and a type name. All dimensions are declared with explicit integer constants, not arbitrary expressions. Give a grammar for Cool array declarations. In addition, define an attribute `size` and give attribute equations that compute the number of elements of the array. In the examples above, your solution should assign `a.size = 5` and `b.size = 100`. You may assume that that terminal symbols have an attribute `val` that is the lexeme of symbol. Do not use ellipses (...) in your grammar.

```
ArrayDef -> id Dims : type { id.size = Dims.size }
Dims0 -> Dims1 Dim { Dims0.size = Dims1.size * Dim.size }
Dim -> [ int ] { Dim.size = int.val }

(note: Dims0 and Dims1 are actually the Dims non-terminal. The different
name is used to distinguish them in the attribute grammar equation).
```

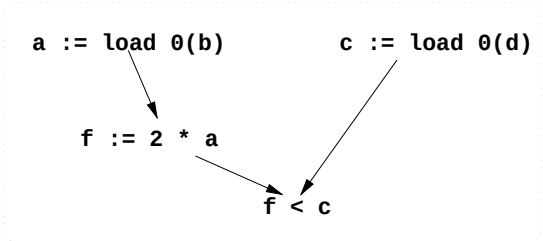
4. **Instruction Scheduling and Register Allocation** (30 points)

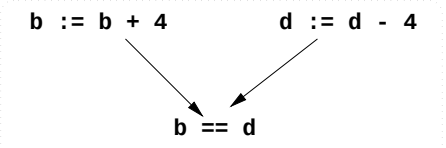
Consider the following intermediate code control-flow graph:

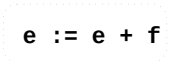


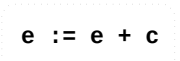
(a) Assuming that `e` is live on exit from the loop, show the set of live variables before and after each instruction. Annotate the control-flow graph above with your answer.

(b) Show the dependence graph for each basic block of this control-flow graph.









(c) In class we discussed both how to perform register assignment and instruction scheduling. Unfortunately, these two important backend components do not always work well together. Assume that register allocation is done before instruction scheduling. Explain why this may be undesirable—i.e., show that instruction scheduling can be made worse by first performing register allocation.

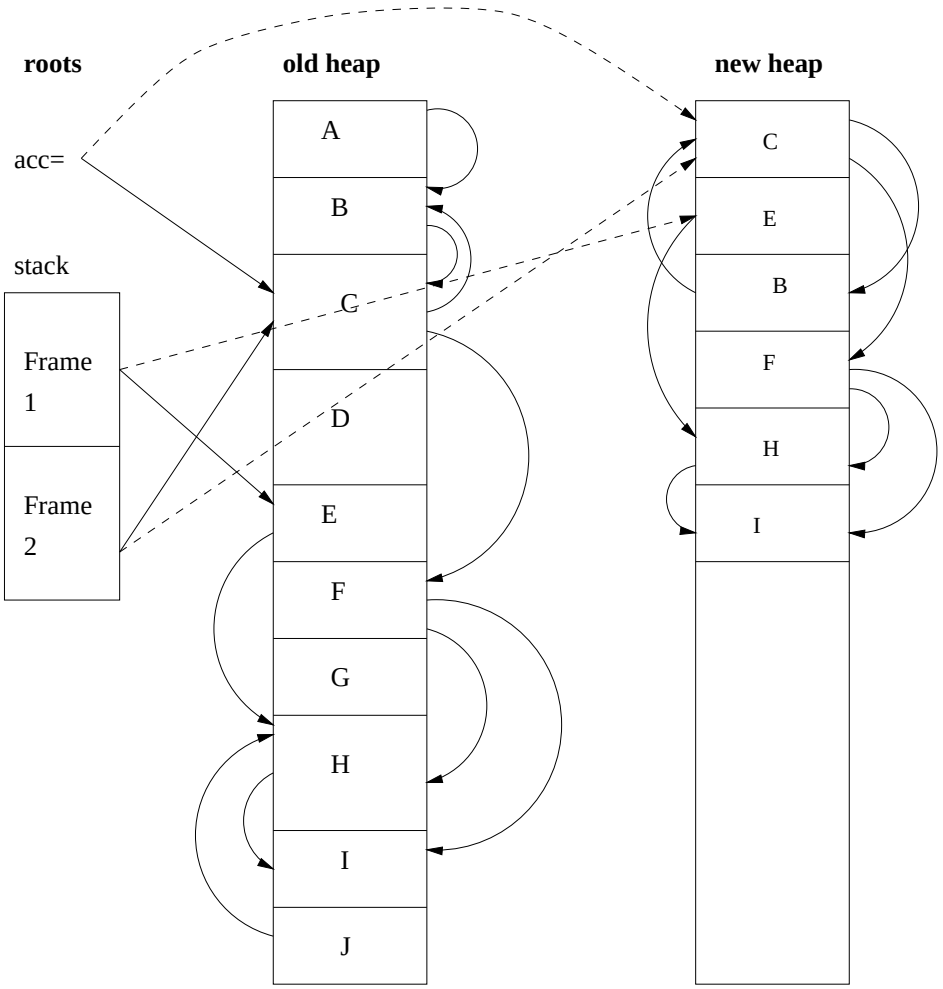
```
By reusing registers, the code that exists after register allocation
may introduce "false" dependencies between instructions. These
dependencies will prevent reordering, resulting in worse code.

For example, before register allocation you could have:

  t1 <- t2 + t3
  t4 <- t5 + t6

These two instructions can be swapped if it will make the scheduling
better.

After register allocation, they could be:

  r0 <- r2 + r3
  r0 <- r5 + r6

These two instructions cannot be swapped (of course, in this simple
example, the first one is useless, but...)
```

5. Garbage Collection (20 points)

Show the result of running the stop & copy garbage collection algorithm on the heap shown below. Fill in the second heap with the results of the garbage collection. You need not represent the forwarding pointers from the old to the new space.

The following points are important:

- The roots are processed in the order: accumulator (acc), then the stack (first frame1, then frame2).
- When an object on the heap has several pointers leaving it, you should process these pointers in top to bottom order.



6. Type Checking, Runtime Organization (20 points)

We add a new kind of expression to extend Cool with concurrency:

$$\texttt{Expression} \rightarrow \ldots \mid \texttt{fork(Expression,Expression)} \mid \ldots$$

The semantics of `fork` is that the program may evaluate $\texttt{e}_1$ and $\texttt{e}_2$ in any order, including simultaneously. On a parallel machine $\texttt{e}_1$ and $\texttt{e}_2$ could actually execute at the same time; on a sequential machine the two computations may be interleaved by partially evaluating $\texttt{e}_1$, partially evaluating $\texttt{e}_2$, switching back to $\texttt{e}_1$, and so on.

(a) (5 points) Assume the semantics of $\texttt{fork}(\texttt{e}_1, \texttt{e}_2)$ is that both expressions are evaluated and the result is the value of $\texttt{e}_2$. Write a Cool type rule for `fork`; make your rule as accurate as possible.

```
O, M, C |- e1 : t1
O, M, C |- e2 : t2
---------------------------
O, M, C |- fork(e1, e2) : t2
```

(b) (5 points) Now assume the semantics of $\texttt{fork}(\texttt{e}_1, \texttt{e}_2)$ is that both expressions are evaluated and the result is the value of the first expression that completes. Write a Cool type rule for `fork`; make your rule as accurate as possible.

```
O, M, C |- e1 : t1
O, M, C |- e2 : t2
---------------------------
O, M, C |- fork(e1, e2) : lub(t1,t2)
```

(c) (10 points) Parallel evaluation of $\texttt{fork}(\texttt{e}_1, \texttt{e}_2)$ implies that activation records cannot be stack allocated. Explain why this is the case. (Note: You are not asked to solve the problem, merely to explain the problem.)

```
If the two computations do not share a common stack pointer, they will
simply overwrite each others activation records. If they do share it,
the following sitation can arise
- procedure a in computation 1 is called
- procedure b in computation 2 is called
- procedure a terminates. How can the stack pointer be reset ? If it
is reset to where the stack was when procedure a was called, this will
also clear procedure b's activation from the stack.
```

7. Code Generation (30 points)
This problem explores the design of a code generator for a small language. The grammar of the language is:

$$\begin{aligned} \texttt{Program} &\rightarrow \texttt{f(x)} = \texttt{E} \\ \texttt{E} &\rightarrow \texttt{integer} \mid \texttt{x} \mid \texttt{f(E)} \mid \texttt{E}_1\texttt{?E}_2 : \texttt{E}_3 \end{aligned}$$

The only data type in this language is integer. A `Program` is a single, possibly recursive, function definition `f` of one argument `x`. The formal parameter `x` has lexical scope. The meaning of expressions is:

- `integer` is the integer constant.
- `x` is the value of the formal parameter.
- `f(E)` is a recursive call to `f`.
- $\texttt{E}_1\texttt{?E}_2 : \texttt{E}_3$ is the value of $\texttt{E}_2$ if $\texttt{E}_1$ is non-zero and the value of $\texttt{E}_3$ if $\texttt{E}_1$ is zero.

The following page gives a skeleton of a code generation algorithm for this language; you should fill in the skeleton to generate machine code. In addition, describe your organization of activation records in the space at the bottom of this page.

Use only the following three registers in your solution:

- `$a` is the accumulator
- `$sp` points to the first empty word beyond the top of the stack
- `$ra` holds the return address

You may use `newlabel()` to create new, unique labels. Use any clear pseudo-code notation for your program. You may generate any reasonable three-address code, including MIPS (see problem 2 for example MIPS instructions). Assume that the program is error-free; do not check for any errors.

```
Activation record (stack growing downwards on page):

------------
| argument |
|----------|
| ra       |
------------
             <- sp points to next free space at function entry

(There is no need for a frame pointer...)
```

```
cg_fundef(p)  = /* called with the function definition  */
{

    case p of
       f(x) = e =>
         emit "{f}:"             // label for function
         emit "addiu $sp $sp -4" // reserve space for activation record
         emit "sw $ra 4($sp)"    // save return address
         cg_e(e);
         emit "lw $ra 4($sp)"    // return...
         emit "addiu $sp $sp 8"  // pop full AR
         emit "jr $ra"
}

cg_e(e) =
{

    case e of
         integer =>
           emit "li $a {integer}" // load value of integer into acc

         x => // Only 1 argument, we know where it is...
              // (assuming static checking has been done correctly...)
           emit "lw $a 8($fp)"

         f(e) =>
           // Call f
           cg_e(e);
           emit "addiu $sp $sp -4" // reserve space for argument
           emit "sw $a 4($sp)"     // save it
           emit "jal {f}"          // call function

         e1 ? e2 : e3 =>
           cg_e(e1)
           elselab = newlabel()
           endlab = newlabel()
           emit "beqz $a0 {elselab}" // branch to else if zero
           cg_e(e2)                  // non-zero: evalauet e2
           emit "ba {endlab}"
           emit "{elselab}:"         // zero: evaluate e3
           cg_e(e3)
           emit "{endlab}:"
}
```

8. Short Answers (30 points)
For each of the following questions we are looking for a clear, concise answer as well as the right idea.

(a) (5 points) Give a definition and an example of an ambiguous grammar.

```
A grammar is ambiguous if there is a string that has two parse trees.

Example:
E -> E + E | E * E | int

(eg on input: 2+3*4)
```

(b) (5 points) What does it mean for a type rule to be sound?

```
A type rule is sound if when it assigns type t to an expression, that
expression evaluates to type t (in Cool this is understood to mean
"evaluates to a type that is a subtype of t").
```

(c) (10 points) Debuggers work by keeping track of which group of instructions in the assembly code corresponds to which line of the source code. An assumption of most debuggers is that the assembly instructions for a source line appear consecutively in the assembly program. Name an optimization that makes the job of the debugger more difficult and explain why. You should illustrate your answer with an example.

```
Instruction scheduling may arbitrarily move the instructions of a
basic block around. It may thus intersperse the instructions from one
source line with those from another, thus destroying the assumption on
which the debugger is based.

Example (x, k are global):
x = y * 3
k = a * 2

Initially could compile to:
t1 <- y * 3     -- statement 1
store t1 @ x    -- statement 1
t2 <- a * 2     -- statement 2
store t2 @ k    -- statement 2

But after instructio scheduling it might become:
t1 <- y * 3     -- statement 1
t2 <- a * 2     -- statement 2
store t1 @ x    -- statement 1
store t2 @ k    -- statement 2
```

(d) (10 points) In the following program, what does `g()` print under: (a) call-by-value and lexical scope, (b) call-by-value and dynamic scope, (c) call-by-reference and lexical scope, and (d) call-by-reference and dynamic scope.

```
fun g() =
{
   int a = 2;
   void f(int b) {
       b = b * a;
       a = a - b;
       }
    {
      int a = 10;
      f(a);
      print a;
    }
}
```

```
a: 10
b: -90
c: 20
d: 0
```