Final

- Please read all instructions (including these) carefully.
- There are 7 questions on the exam, each worth between 10 and 30 points. You have 3 hours to work on the exam.
- You may not discuss the exam with anyone who has not taken the exam until after 4pm Monday.
- The exam is closed book, but you may refer to your six 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 Solution

SID or SS#: _____

Problem	Max points	Points
1	20	
2	20	
3	20	
4	30	
5	30	
6	20	
7	10	
TOTAL	150	

1. **Regular Expressions** (20 points)

(a) Many operating systems require user passwords conform to certain rules to reduce the odds that an attacker can guess a password. Consider the following rules:

- A password consists of upper- and lower-case letters and digits.
- A password has at least four characters.
- At least one character of a password is a digit.

Write a regular expression for passwords satisfying these three rules. Take care that the regular expression notation you use (flex or non-flex) is clear.

```
Letter = [A-Za-z]
Digit = [0-9]
Char = Digit | Letter

Digit Char   Char  Char Char* |
Char  Digit  Char  Char Char* |
Char  Char   Digit Char Char* |
Char  Char   Char  Char* Digit Char*
```

(b) Give a regular expression for all context-free grammars with non-terminals $A, B, \ldots, Z$, terminals $0, 1, \ldots, 9$, and ϵ. A production may not have alternatives (i.e., no productions have "|" on the right-hand side). Productions are terminated by a blank. Your solution may generate grammars with duplicate or useless productions. Don't worry about which non-terminal is the start symbol.

```
NonTerm = [A-Z]
Term =  [0-9]
All  = NonTerm | Term | 'epsilon'

(NonTerm -> All All* ' ')+
```

2. **Garbage Collection** (20 points)
Consider the following simple model for estimating the cost of garbage collection (GC). We divide running time into time spent actually running the user's program (which is useful work) and time spent in GC (which is overhead).

   - The number of cycles required for a single GC is equal to the number of different memory words that GC references. That is, even if a GC touches a particular word of memory more than once, we only charge 1 cycle.
   - The *survival rate* is the size of the live data after a GC, expressed as a fraction of the total heap size. For example, if garbage collecting a 10MB heap leaves 1MB of live data, then the survival rate is 0.1.
   - A user program allocates 1 word of heap for every 10 cycles of time the user program is running (i.e., not in a GC).

   Given a survival rate s and a heap size n, give a formula for the fraction of total execution time spent in garbage collection for Mark and Sweep. Assume the program runs for a long time (i.e., we are interested in the steady state performance).

```
The cycles for a garbage collection is n, since the sweep phase touches
every word of memory.  With a survival rate of s, the user program can
allocate (1 -s)n words after a garbage collection, which means it can run
for 10(1 -s)n cycles.  The time from the start of one garbage collection
to the start of the next is then n + 10(1 - s)n.  Thus, the fraction of
time spent in GC is

                 n                1
MS(s,n) = --------------  =  --------
          n + 10(1 - s)n     11 - 10s
```

   Given a survival rate s and a total heap size n, give a formula for the fraction of total execution time spent in garbage collection for Stop and Copy. Again, give the steady state performance.

```
The argument here is similar to the one above.  The differences are:
First, Stop & Copy systems can only use 1/2 half of the total heap (.5n).
Second, a garbage collection only touches the live data, so only
.5ns data is touched in old space; it is copied to new space, so the total
cost of one garbage collection is 2(.5ns) = sn.  The program can run
for 10(1 - s).5n cycles before another GC is needed.  The fraction of time
spent in GC is

                   sn              s
SC(s,n) = ------------------  =  ------
          sn + 10(1 - s).5n      5 - 4s
```

3. **Grammars** (20 points)

(a) Rewrite the following grammar so that it is left-recursive:

$$\begin{aligned} Decls &\rightarrow Decl;\ Decls \mid EndDecl \\ Decl &\rightarrow Type\ id \\ Type &\rightarrow int \mid char \\ EndDecl &\rightarrow \epsilon \end{aligned}$$

```
Decls -> Decls Decl; | EndDecl

The rest of the grammar is the same.
```

(b) Consider the following grammar for arithmetic expressions over variables x, y, and z and the digits 0 through 9.

$$\begin{aligned} E &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \mid x \mid y \mid z \\ E &\rightarrow E * E \mid E + E \mid E - E \end{aligned}$$

Write a grammar for the subset of these expressions that can be completely constant folded (i.e., evaluated to a number) at compile time.

```
C -> C * C  |  C + C  |  C - C  |  E * 0  |  0 * E  | 0 | 1 | ... | 9

where E is generated using the grammar above.
```

4. **Code Generation** (30 points)
Consider the following subset of Cool expressions, extended with a `for` loop:

$$
\begin{array}{rcl}
\texttt{E} & \rightarrow & \texttt{for id} = \texttt{E to E do E} \\
& | & \texttt{let id} : \texttt{Type} \leftarrow \texttt{E in E} \\
& | & \texttt{id} \leftarrow \texttt{E} \\
& | & \texttt{int}
\end{array}
$$

In this grammar, `id` is a variable name and `Type` is a Cool class name. The semantics of `for id` $= \texttt{E}_1$ `to` $\texttt{E}_2$ `do` $\texttt{E}_3$ are: $\texttt{E}_1$ is evaluated giving an integer i_1, $\texttt{E}_2$ is evaluated giving an integer i_2, and then $\texttt{E}_3$ is evaluated with `id` $= x$ for each $i_1 \leq x \leq i_2$ in order. If $i_1 > i_2$ then the `for` terminates without evaluating $\texttt{E}_3$. The result of the `for` is 0. The `id` is a new variable that hides any definitions in outer scopes; the scope of `id` is $\texttt{E}_3$.

(a) Give a type rule for `for`. Your rule should be as accurate as possible.

```
O, M, C |- E1 : Int
O, M, C |- E2 : Int
O[ id <- Int], M, C |- E3 : T
---------------------------------
O, M, C |- for id = E1 to E2 do E3 : Int
```

(b) The identifier introduced by a `for` is an *iteration* variable. A program should not explicitly update an iteration variable. Write attribute equations that check programs for assignments to iterations variables (you should only write equations for the productions above).
Your solution should use one inherited attribute *iset* and one synthesized attribute *legal*. *E.iset* is the set of iteration variables in scope at E, and *E.legal* is true if (and only if) E contains no assignments to iteration variables. Your solution should use no other attributes.

```
E -> for id = E1 to E2 do E3          E3.iset = E.iset + { id.name }
                                      E1.iset = E2.iset = E.iset
                                      E.legal = E1.legal & E2.legal & E3.legal

E -> let id : Type <- E1 in E2        E1.iset = E.iset
                                      E2.iset = E.iset - { id }
                                      E.legal = E1.legal & E2.legal

E -> id <- E1                         E1.iset = E.iset
                                      E.legal = E1.legal & (id not in E.iset)

E -> int                              E.legal = true
```

(c) Write a code generation function for `for`. You should show actual assembly code (pseudo-assembly is fine—don't worry about MIPS syntax). Assume the following about the runtime environment:

- There is a frame pointer `$fp`, stack pointer `$sp`, and accumulator `$a0`.
- The iteration variable `id` is stored at offset `idloc` in the frame.
- Use 32-bit integers (not integer objects). Don't worry about how this affects code for the subexpressions (assume the rest of the compiler can handle that).
- Temporaries are pushed on the stack, not stored in the frame.
- The net effect of evaluating `E` leaves the result of `E` in `$a0`, leaves `$sp` and `$fp` unchanged, and may overwrite any other register.

```
In this solution "env" stands for the code generation
environment.  The environment manipulation is not the
interesting aspect of this problem, and in fact no
points were deducted for ignoring it completely.

cgen(env, for id = E1 to E2 do E3) =

cgen(env, E1)
sw $a0 idloc($fp)      // set initial value of id
cgen(env, E2)
sw $a0 0($sp)          // save upper bound on stack
addiu $sp $sp -4
mv $t1 $a0             // copy upper bound to temp $t1
lw $a0 idloc($fp)      // put the counter in the accumulator
top:                   // this is a fresh label
blt $t1 $a0 exit       // exit if counter is bigger than upper bound
cgen(add(id.name,env), E3)
lw $a0 idloc($fp)      // load and update counter
addiu $a0 $a0 -1
sw $a0 idloc($fp)
lw $t1 4($sp)          // load the upper bound
j top                  // loop
exit:                  // another fresh label
addiu $sp $sp 4        // pop the stack
lw $a0 0               // load the accumulator with result of for
                       // (could also be the object 0)
```

5. **Miscellaneous Short Answer** (30 points)
For each of the following questions we are looking for clarity and conciseness as well as the right idea.

(a) The scope of a C function is from the point of definition to the end of the file. How does this scope rule simplify compiler implementation?

If functions are defined before they are used, then it is possible to implement a compiler that traverses the file only once (a one-pass compiler).

(b) In Cool, the meaning of a program is `new Main.main()`. Change the language so that the meaning of a program is just `new Main`. How can a program written under the old definition be ported to the new definition with minimal changes?

Given some Cool program P written for the (new Main).main() semantics, we can port it to the (new Main) semantics by embedding the method dispatch in an attribute initialization expression. However, a general solution must take care not to introduce spurious dispatches: the program P might allocate more than one Main object, and we would change the program's behavior if we dispatched the main method every time a Main object were allocated. For the following general solution, let T be a typename not occuring in the input program P.

```
P[T/Main] (* the original program with Main globally renamed to T *)

class Main {
  main : T <- (new T);
  ignore : Object <- main.main();
};
```

(c) Give a definition of a handle (in bottom-up parsing).

If S -*-> AXw -> ABw, where A is a string, X -> B is a production, and w is a string of terminals, then AB is a handle of ABw at B.

More informal definitions were also acceptable.

(d) Give a definition of principal type.

A type T is principal for an expression if all other types for that expression can be derived by substituting for type variables in T.

(e) Give a definition of forwards analysis. Give a definition a backwards analysis.

A forwards analysis pushes information from inputs towards outputs.
A backwards analysis pushes information from outputs towards inputs.

(f) Give an example of a program where activation records cannot be allocated on the stack. Use any reasonable and clear programming notation.

```
(((fun x (fun y (x + 1))) 3) 2)
```

6. **Instruction Scheduling and Register Allocation** (20 points)

(a) 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 instruction scheduling is done before register allocation. Explain why this may be undesirable; i.e., show that register allocation can be made worse by first performing instruction scheduling. *(Note: For those who studied last year's final, this is a different question!)*

```
Instruction scheduling may increase the live range of variables by
increasing the distance between a variable's definition and its last use.
Larger live ranges for variables means that more live ranges may overlap,
leading to more edges in the register interference graph, and therefore
a harder register allocation problem.
```

(b) A *clique* is a fully connected graph (every node has an edge to every other node). Given a register interference graph that is a clique with k nodes, how many register allocations are there with no spilling for a machine with k registers? Justify your answer.

```
All of the nodes will have degree k - 1, which means we can delete them
in any order and assign each variable to any register.  There are k!
such assignments.
```

7. **Global Analysis** (10 points)
One homework asked you to develop a global constant propagation algorithm that simultaneously inferred the "constantness" of all variables in a control-flow graph. Recall that during analysis a variable may be considered to have one of three kinds of abstract values: # (don't care), a constant c, or $*$ (any value). When we simultaneously infer whether all values are constants, the analysis can be extended from propagating simple constants to full constant folding.

Give the most precise transfer function you can for the statement

$$\texttt{x} := \texttt{y} + \texttt{z}$$

The input and output of your function should be a tuple of $\langle x, y, z \rangle$ abstract values.

```
Call the function F.

F(< a, c1, c2>) = < c1 + c2, c1, c2 >
F(< a, #, b>)   = < #, #, b >
F(< a, b, #>)   = < #, b, # >
F(< a, *, b>)   = < *, *, b >   provided b is not #
F(< a, b, *>)   = < *, b, * >   provided b is not #
```