CS143 Final
Fall 2010

• Please read all instructions (including these) carefully.

• There are 4 questions on the exam, all with multiple parts. You have 120 minutes 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: ____________________________________________

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: ________________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>
1. **Register Allocation** (20 points)
Consider the following fragment of intermediate code after register allocation:

```c
/* r2 and r3 are live on entry */
r1 = r2 + r2
r3 = r3 * r1
r2 = r2 * 2
r1 = r3 + r1
/* r1 and r2 are live on exit */
```

Give the intermediate code *before* register allocation is performed, and in particular give intermediate code whose register interference graph has the maximum number of nodes of all possible programs that yield the code above after register allocation.

- (10 points) Give the intermediate code, and show the live variables at each program point. Use variables `a, b, c, ...` in the intermediate code.

- (10 points) Give the register interference graph for the intermediate code and the graph coloring that results in the final code given above. Each node of the graph should have two labels: the variable the node represents in the intermediate code, and the register (color) it is assigned.
2. Type Checking (30 points)

Tired of writing your own linked list classes in Cool, you decide to modify the language to include a new list type. The type \([C]\) denotes the type of a list of objects of class \(C\). You extend the grammar for Cool as follows:

\[
expr ::= \text{existing Cool expressions} \\
| \text{nil} \\
| \text{cons}(expr, expr) \\
| \text{head} \ expr \\
| \text{tail} \ expr
\]

- \text{nil} evaluates to void and represents the empty list
- \text{cons}(e_1,e_2) allocates a new cons cell in the heap. The cons cell consists of an element \(e_1\) (called the head of the list) and a list \(e_2\) (called the tail of the list)
- \text{head} \(e\) returns the head of the cons cell \(e\) (i.e., \(\text{head}(\text{cons}(e_1,e_2)) = e_1\))
- \text{tail} \(e\) returns the tail of the cons cell \(e\) (i.e., \(\text{tail}(\text{cons}(e_1,e_2)) = e_2\))

Some examples:

- \text{isvoid} \text{nil} evaluates to true;
- \text{cons}(1,\text{cons}(2,\text{nil})) creates a two-element list;
- \text{tail} \text{cons}(1,\text{cons}(2,\text{nil})) returns the one-element list \text{cons}(2,\text{nil});
- \text{cons}(3, \text{tail} \text{cons}(1,\text{cons}(2,\text{nil}))) returns the two-element list \text{cons}(3,\text{cons}(2,\text{nil})),
  but note that this list creates a new cons cell and does not modify the original list \text{cons}(1,\text{cons}(2,\text{nil})).
(a) (6 points) Complete the definition of the method sumList below. The method should take a list of integers as its argument and return the sum of the integers in the list (the sum of an empty list is 0). For example, 

\[
\text{sumList(cons(1,cons(2,cons(3,nil))))} \quad \text{should return 6.}
\]

\[
\begin{align*}
\text{sumList(list : [Int]) : Int} & \{
\end{align*}
\]

\[
\}
\]

(b) (6 points) The type checking rule for cons(e_1,e_2) is given below. Provide type checking rules for head e and tail e that will work with the rule for cons. Ignore SELF_TYPE in your answer.

\[
\begin{align*}
O, M, C \vdash e_1 : T \\
O, M, C \vdash e_2 : [T] \\
O, M, C \vdash \text{cons}(e_1, e_2) : [T]
\end{align*}
\]
(c) (6 points) After adding the above rules to your type checker, you quickly realize that the rules are more restrictive than necessary. Specifically, the rule for \texttt{cons}(e_1,e_2) requires that all objects within the list be of exactly the same type. Write a new typing rule for \texttt{cons}(e_1,e_2) that allows you to prepend arbitrary objects to any list. The rule must be sound but provide as much expressiveness as possible. Ignore \texttt{SELF\_TYPE} in your answer.

(d) (6 points) Your friend who has not taken CS143 is admiring your changes to Cool and proposes the following subtyping relation for lists: if \( A \leq B \) then \([A] \leq [B]\). This change reminds you of the problems with Java arrays and the subtyping relation \texttt{Array}[A] \leq \texttt{Array}[B]. Will following your friend’s suggestion result in the same type safety violation? Explain why or why not.

(e) (6 points) Ignoring your friend’s suggestion and the rule you gave in part (c), do the typing rules given in part (b) guarantee that list operations will always succeed, or are there run-time errors that could arise from using list operations, even if they are type correct? Either explain why the rules guarantee list operations always complete successfully, or give an example of an error that might occur.
3. **Operational Semantics and Code Generation** (45 points)

Consider the simple calculator language defined by the following grammar:

\[
S \rightarrow \text{add } c \mid S_1; S_2 \mid \text{undo} \mid \text{repeat } c \ S \mid (S)
\]

In this language, every program evaluates to an integer. The statement \( \text{add } c \) adds the integer constant \( c \) to the current value; \( S_1; S_2 \) executes statement \( S_1 \) and then \( S_2 \), \( \text{undo} \) undoes the last (i.e., the most recently executed and not undone) \( \text{add} \) operation, and finally \( \text{repeat } c \ S \) repeats \( S \ c \) times, where \( c \) is non-negative. Initially (at the start of the program) the current value is 0. If there is no add instruction before an undo operation, then undo has no effect. For example, the program

\[
\text{add 5}; \ \text{add 2}; \ \text{add 7}; \ \text{undo}; \ \text{undo}
\]

evaluates to 5, and the program

\[
\text{add 5}; \ (\text{repeat 3 add 2}) \ ; \ \text{undo}
\]

evaluates to 9 because the undo operation un-does the add operation in the last iteration of the loop.

• (10 points) Is this language ambiguous? If it is ambiguous, draw two distinct parse trees illustrating each source of ambiguity (there may be more than one). If it is not ambiguous, explain why not.

• (5 points) The operational semantics of this language is given as inference rules of the form \( E \vdash S : c, E' \), meaning that under environment \( E \), statement \( S \) evaluates to integer \( c \) and produces a new environment \( E' \). Describe in English what the environment \( E \) must contain.
• (10 points) Give operational semantics for all of the constructs except $(S)$. 
• (5 points) Describe the run-time organization you will use to emit code for this language. For simplicity, assume in this question and the next that repeat statements are not nested (i.e., there is only one loop active at a time).

• (15 points) Give a code generation function $cgen(S)$ for all the constructs except $(S)$. Observe the following:
  - This function should emit pseudo-MIPS assembly code; you may only use the instructions `lw`, `sw`, `li`, `addiu`, `beq`, `beqz`, and `j`.
  - Do not use any run-time library or “starter code” functions from the programming assignments.
  - Do not worry about generating unique labels for statements.
  - Again, assume a repeat statement does not include any repeats within its body.
4. **Short Answer** (25 points)

- (5 points) You are trying to decide which garbage collection algorithm to include in the Cool 2.0 compiler. After careful measurements, you discover that, in the average Cool program, 95% of the heap is garbage when the garbage collector is invoked. Should you use Mark & Sweep or Stop & Copy? Justify your answer.

- (5 points) After profiling some Cool program with different garbage collectors, you discover that some programs run significantly faster right after the garbage collector finishes a collection. Which garbage collection algorithm discussed in class may have this effect and why?

- (5 points) While researching compiler optimization techniques, you come across an algorithm called *escape analysis* that identifies which allocations (e.g., calls to `new`) in a given function become garbage before the function returns and which allocations might escape—that is, still be accessible after the function activation return. If an allocation does not escape a function, how can you use this fact to reduce the number of heap allocations?
(5 points) After taking CS143, you feel confident enough to critique the optimizations performed by the Java compiler. Looking at the byte code files, you see a (stylized) byte-code of the following form:

\[ x.f = 2; y.f = 3; a = x.f; \]

You are surprised that the Java compiler does not replace the last statement by \( a = 2 \), since it claims to perform constant propagation. Do you think the Java compiler misses a constant propagation opportunity or is there a good reason why it does not perform this optimization? Justify your answer.

(5 points) Static Single Assignment form (SSA) is an intermediate representation where each variable is assigned exactly once. Every program can be converted to SSA form by introducing new versions for variables after every assignment. For example \( x=1; x=x+1; \) is converted to SSA form as \( x1 = 1; x2 = x1+1 \). Does conversion to SSA form have a positive or negative impact on register allocation? In particular, can a register interference graph that was not \( k \)-colorable become \( k \)-colorable after conversion to SSA form or vice versa? Justify your answer.