

Written Assignment 10

Due April 27, 2006

This assignment asks you to prepare written answers to questions on register allocation and instruction scheduling. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work. *Please write the name of the account you are using for CS164 and your section time on your homework.* Remember that written assignments are to be turned in either in class or in the CS164 homework box in 283 Soda by 12:30 PM on the due date.

1. Suppose we have a machine with three registers, two integer units and one load/store unit with the following cycle counts (and no out-of-order execution):

Addition	1 cycle
Multiplication	2 cycles
Branch	2 cycles
Load/store	3 cycles

Consider the following basic block:

```
a := load 0($fp)
b := load 4($fp)
c := a + b
d := b * 42
e := d + 2
```

Assume only `a`, `c`, and `e` are live on exit to this block.

- (a) Draw the instruction dependence graph for the basic block shown above. Label each instruction with the priority $\langle L, D \rangle$ as defined in lecture. Perform instruction scheduling for the above basic block using the heuristics discussed in lecture. Note that the heuristics is optimal for this basic block.
 - (b) Given an example of a basic block for which this heuristics does not yield the most optimal scheduling. Explain your answer.
 - (c) Give an example of a control flow graph for which the best instruction scheduling for each basic block does not yield the best instruction schedule for the entire control flow graph.
2. In class we discussed how to perform register allocation and instruction scheduling. Unfortunately, these two important back-end components do not always work well together. This problem asks you to show how this can happen. Assume that the underlying machine is as described in Question 2.
 - (a) If register allocation is done before instruction scheduling, it may assign multiple temporaries to the same register, which can introduce “false” dependencies between variables that can interfere with scheduling. Explain why it is better to perform instruction scheduling on the basic block in Question 2 before register allocation.
 - (b) On the other hand, it’s not always a good idea to do instruction scheduling first, because instruction scheduling can increase the lifetimes of temporaries. Give an example program that fits into three registers if register allocation is done before instruction scheduling, but requires more than three registers (and hence a spill on our machine) if register allocation is done after instruction scheduling.

3. Suppose that we have already updated Cool to support `try/catch`, `try/finally`, and `throw` as in PA6. Now we wish to add a new construct, `protect e`, to the language. At run time, `protect e` simply returns the value of `e`; however, at compile time, the type checker must verify that no exceptions can be thrown out of `e`. (Exceptions may be thrown in the process of evaluating `e`, but they must be caught before `e` finishes evaluating.)

In order to check `protect` expressions at compile time, we need to extend our typing judgment to track not only the type T of an expression, but also a boolean E indicating whether an exception can be thrown out of the expression.

$$O, M, C \vdash e : T, E$$

For example, the extended rule for `+` would be:

$$\frac{O, M, C \vdash e_1 : \text{Int}, E_1 \quad O, M, C \vdash e_2 : \text{Int}, E_2}{O, M, C \vdash e_1 + e_2 : \text{Int}, E_1 \vee E_2} \quad [\text{Plus}]$$

Give the extended type rules for `protect`, `try/catch`, `try/finally`, and `throw`.

4. Consider the following Cool code for a linked list of integers:

```
class List {
  value : Int;
  next : List;
  getValue() : Int { value };
  isSorted() : Bool {
    (isVoid next) ||
    (value <= next.getValue() && next.isSorted())
  };
};
```

Note that the short-circuit logical operators `||` and `&&` are not part of the original Cool language, but we have used them here for the sake of clarity.

- The `isSorted` method only checks that the integers are in ascending order, but the user may want to check that the integers are in descending order instead. Use *higher-order functions* (as described in lecture) to write a new `isSorted` method that allows the caller of your method to specify how to compare two integers.
- Imagine that the user really wanted to store a list of strings instead of a list of integers. Use *polymorphism* (as described in lecture) to update your solution from part (a) so that the user of your class can store *any* type in the list. The user should still be able to specify how to compare these values as in part (a).