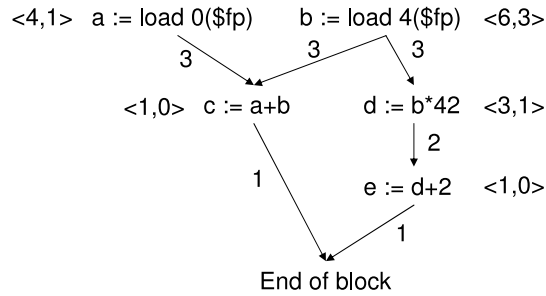


Solutions to Written Assignment 10

1. (a) The instruction scheduling heuristics computes the following instruction dependence graph.

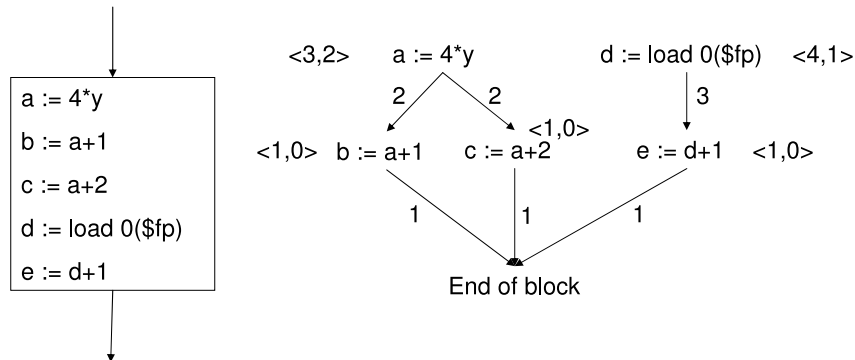


Based on the above graph, the heuristics comes up with the following schedule.

```

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

- (b) Consider the following basic block, along with its instruction dependence graph.



The heuristic comes up with the following schedule, which takes 6 cycles.

```

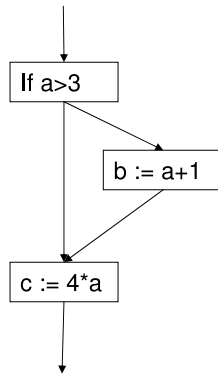
d := load 0($fp)
a := 4*y
nop
b := a+1
c := a+2
e := d+1
    
```

The optimal schedule is as follows, which takes 5 cycles.

```

a := 4*y
d := load 0($fp)
b := a+1
c := a+2
e := d+1
    
```

(c) Consider the following control flow graph.



Optimal basic-block scheduling yields the following schedule, which takes 4 or 5 cycles (depending on whether the branch evaluates to true or false).

```

    if a>3 jump L1
    *nop
    b := a+1
L1: c := 4*a
    nop
  
```

The best schedule is as follows, which takes 3 or 4 cycles (depending on whether the branch evaluates to true or false).

```

    c := 4*a
    if a>3 jump L1
    *nop
    b := a+1
L1:
  
```

2. (a) As-is, this code takes 10 cycles to complete, and if we schedule it we'll move the multiplication up so that the block takes 9 cycles. If, however, we first perform register allocation, then **b** and **d** will be stored in the same register, creating a false dependency and preventing the multiplication instruction from being moved up by the instruction scheduler.
- (b) Here is an example of a program for which register allocation is better done before instruction scheduling, assuming only **g** is live on exit to the block:

```

a := 1
b := 1
c := a + b
d := b + c
e := load 0($fp)
f := d + e
g := a + f
  
```

As-is, we can fit the temporaries into three registers. However, scheduling this code will pull the `load` to the top of the block, requiring a spill, since **a**, **b**, **c**, and **e** will be live at the same time.

3. The simplest rule is the one for `protect`:

$$\frac{O, M, C \vdash e : T_1, false}{O, M, C \vdash \text{protect } e : T_1, false} \quad [\text{Safe}]$$

That is, `protect e` has the same type as `e`, but only type checks if `e` is known not to throw any exceptions.

Next, the rule for `throw`:

$$\frac{O, M, C \vdash e : T_1, E_1}{O, M, C \vdash \text{throw } e : T_2, \text{true}} \quad [\text{Throw}]$$

We know this construct will throw an exception regardless of what `e` does, so we ignore E_1 and return `true`.

Next, the rule for `try/catch`:

$$\frac{O, M, C \vdash e_1 : T_1, E_1 \quad O[T/x], M, C \vdash e_2 : T_2, E_2}{O, M, C \vdash \text{try } e_1 \text{ catch } x : T \Rightarrow e_2 : T_1 \sqcup T_2, E_2} \quad [\text{Try/Catch}]$$

Since any exception thrown by e_1 will be caught, this construct only throws an exception if e_2 throws an exception.

Finally, the rule for `try/finally`:

$$\frac{O, M, C \vdash e_1 : T_1, E_1 \quad O, M, C \vdash e_2 : T_2, E_2}{O, M, C \vdash \text{try } e_1 \text{ finally } e_2 : T_2 \sqcup T_2, E_1 \vee E_2} \quad [\text{Try/Finally}]$$

That is, if either e_1 or e_2 can throw an exception, then the whole thing can throw an exception.

4. (a) We add a `compare` argument to the `isSorted` function. To simulate a two-argument function, we create a function with type `Int -> (Int -> Bool)`.

How do we use this function? First, note that `compare(value)` has type `Int -> Bool`, and it is a function that compares `value` to its argument. Thus, `(compare(value))(next.getValue())` applies this function to the next value, returning a `Bool` that indicates the result of the comparison.

```
class List {
  value : Int;
  next : List;
  getValue() : Int { value };
  isSorted(compare : Int -> (Int -> Bool)) : Bool {
    (isVoid next) ||
    ((compare(value))(next.getValue()) && next.isSorted())
  };
};
```

To implement the original comparison operator, we could create the following function:

```
let lessThan : Int -> (Int -> Bool) <-
  fun (arg1 : Int) : Int -> Bool {
    fun (arg2 : Int) : Bool {
      arg1 <= arg2
    }
  }
```

By the way, this technique for implementing a two-argument function using one-argument functions is called *currying*.

- (b) To make this class polymorphic, we just need to add a type parameter `t` and then change `Int` to `t` wherever necessary.

```
class List<t> {
  value : t;
  next : List<t>;
  getValue() : t { value };
  isSorted(compare : t -> (t -> Bool)) : Bool {
    (isVoid next) ||
    ((compare(value))(next.getValue()) && next.isSorted())
  };
};
```

The `lessThan` function above can be passed to the `isSorted` method of any object of type `List<Int>`.