Midterm II
CS164, Spring 2006

April 11, 2006

- Please read all instructions (including these) carefully.
- **Write your name, login, SID, and circle the section time.**
- There are 10 pages in this exam and 4 questions, each with multiple parts. Some questions span multiple pages. All questions have some easy parts and some hard parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two pages of handwritten 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. We might deduct points if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

LOGIN: _____

NAME: _____

SID: _____

Circle the time of your section: Fri 10:00 Fri 11:00 Fri 2:00

Problem	Max points	Points
1	10	
2	30	
3	40	
4	20	
TOTAL	100	

1 Liveness (10 points)

Consider the following program fragment.

```
[d]          a := 1
[a,d]          c := 3
[a,c,d]          if a >= 2 goto L2
[a,c,d]          b := c + 1
[a,b,d]          c := b + 2
[a,d,c]      L1:  b := d
[a,d,c]          if a >= d goto L2
[a]          return a
[c]      L2:  a := c + 2
[a,c,]         d := c + 1
[a,c,d]          goto L1
```

(a) Draw horizontal lines to separate the basic blocks.

(b) In the boxes to the left of each instruction write the set of live variables **on entry** to that instruction.

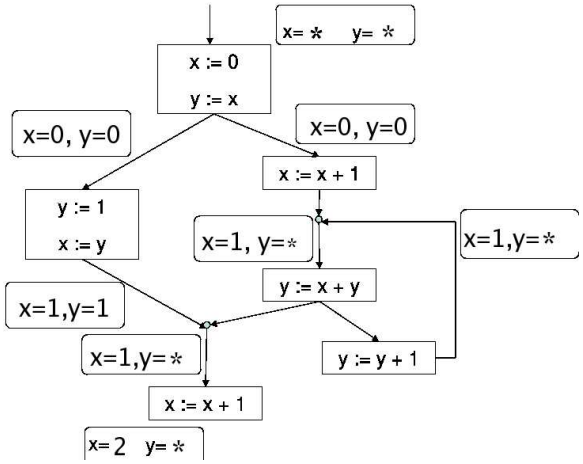
(c) Cross with an X the dead instructions. Explain below how do you recognize a dead assignment.
Solution: Only `b := d` is dead

2 Global Analysis (30 points)

The goal of this exercise is to design a fancier version of global constant propagation discussed in lecture. As in lecture, for each integer variable x we try to find if x is guaranteed to be a constant at a given program point. We write $C_{in}(x, s) = n$ when this is the case right before statement s and the constant is n. The difference from the version discussed in lecture is that now we propagate the information properly across assignments when the right hand side is known to be a constant and across arithmetic operations when both operands are known to be constants.

(a) How can you use the results of this analysis to optimize the instruction "`if` $x \geq 5$ `goto L`"?
Solution: If x is guaranteed to be a constant ≥ 5, then we can replace this instruction with `goto L`. If x is guaranteed to be a constant < 5, then we can eliminate the instruction.

(b) Fill-in the empty boxes with the values computed by the analysis for the variables x and y at each program point in the corresponding CFG. Each box is associated with a point on the nearest edge.



(c) Consider an instruction s with two predecessors p_1 and p_2. Write the formulas for $C_{in}(x, s)$. No need to consider the # value here.

$$\textbf{Solution}: C_{in}(x,s) = \begin{cases} c & \text{if } C_{out}(x,p_1) = C_{out}(x,p_2) = c, \\ * & \text{if } C_{out}(x,p_1) \neq C_{out}(x,p_2), \\ * & \text{if } C_{out}(x,p_1) = * \text{ or } C_{out}(x,p_2) = * \end{cases}$$

(d) Consider an instruction s of the form $x := y + z$. Write the formula for $C_{out}(x, s)$. No need to consider the # value here.

$$\textbf{Solution}: C_{out}(x,s) = \begin{cases} c+d & \text{if } C_{in}(y,s) = c \text{ and } C_{in}(z,s) = d \text{ and } c \neq * \text{ and } d \neq *, \\ * & \text{if } C_{in}(y,s) = * \text{ or } C_{in}(z,s) = * \end{cases}$$

(e) Consider the instruction s of the form `if` $x == n$ `goto L`. Write the formulas for $C_{out}(x, s)$ for the case when the branch is taken.

$$\textbf{Solution}: C_{out}(x,s) = \begin{cases} \# & \text{if } C_{in}(x,s) \text{ is } \#, \\ n & \text{otherwise} \end{cases}$$

(f) Is this analysis a forward or a backward analysis?

Solution: Forward, because we mark an instruction based on something that occurs in the past.

(g) Show a small CFG such that at the end it is true that x has value 0, yet the global analysis is not able to discover that fact. Your program may not contain loops or function calls. Explain your answer.

Solution: Consider a simple CFG with one basic block as follows:

```
x = 0
x = x + y
x = x - y
```

Observe that x is guaranteed to be zero at the end of this block, because x starts out as zero, and then you add y, and then you subtract y. So $x = 0 + y - y = 0$.

However, when global analysis sees this, it doesn't know what value y has, so it assigns $y = *$. Then it sees $x = x + y$, and since $y = *$, that means after this instruction, $x = *$. So in the end, $x = *$, according to global analysis.

Many other solutions are possible.

3 Runtime Organization and Code Generation (40 points)

Suppose we extended the Cool syntax with nested function definitions. We add a new form of *expressions* similar to `let` but defining functions: "**fun** $f(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1}$ $\{e_1\}$ **in** e_2" binds the identifier f to a new function with body e_1 that takes parameters $x_1, \ldots, x_n$ with types $T_1, \ldots, T_n$, returns a value of type T_{n+1}, and can only be called from within e_1 or e_2. Moreover, the body e_1 and the expression e_2 can refer to variables declared in enclosing **let**s, as well as function and method parameters of the enclosing functions, as well as f itself.

Both variable and function names are statically scoped. For example, consider the following method `exp` in class `Math` that computes x^y.

```
class Math {
  f(x : Int) : Int { ... };

  exp(x : Int, y : Int) : Int {
    let a : Int <- 1 in
    fun f(i : Int) : Int {
      fun geta() : Int { a + i } in

      if i = 0 then
          geta()
      else {
          a <- a * x;
          f(i-1);
       }
      fi
    } in
      f(y)
  };
};
```

The `x` in the body of the `f` refers to the parameter `x` of `exp`, and the `a` in the body of `geta` refers to the local variable introduced by the `let` in `exp`. Similarly the `i` in the body of `geta` refers to the argument of `f`. Also, notice that the calls to `f` refer to the `f` declared in `exp`, not the method `f`.

3.1 Type Checking

To type check **fun**, we do not need to modify the typing judgment, nor introduce any additional forms of types. Recall that the Cool typing judgment is of the form $O, M, C \vdash e : T$, where M is a map such that $M(D, f) = (T_1, ..., T_n, T_{n+1})$, whenever class D supports a method named f with n arguments of types T_1,...,T_n, and result type T_{n+1}. You will use the same map M to carry information about the nested functions that are in scope. **We ignore `SELF_TYPE` in this problem.**

(b) Write the remaining hypotheses and complete the conclusion of the typing rule for function invocation:

$M(C, f) = (T'_1, \ldots, T'_n, T_{n+1})$
Solution : $\forall i \in \{1 \ldots n\} \quad O, M, C \vdash e_i : T_i \quad T_i \leq T'_i$

$$\frac{}{O, M, C \vdash f(e_1, \ldots, e_n) : \boxed{\textbf{Solution: } T_{n+1}}}$$

(c) Give a sound typing rule for **fun** (one that allows as many safe programs as possible). If you need to introduce new notation, explain it below the typing rule.

Solution :
$$\begin{array}{l} O' = O[T_1/x_1] \ldots [T_n/x_n] \\ M' = M[(T_1, \ldots, T_n, T_{n+1})/(C, f)] \\ O', M', C \vdash e_1 : T \quad T \leq T_{n+1} \\ O, M', C \vdash e_2 : T' \end{array}$$

$$\frac{}{O, M, C \vdash \textbf{fun } f(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1} \ \{e_1\} \textbf{ in } e_2 : \boxed{\textbf{Solution: } T'}}$$

3.2 Code Generation

The difficulty in code generation is that nested functions may need to use variables declared in enclosing functions. Consider the previous example. The function `f` not only needs access to its own activation record (for variable `i`) but also the activation record for `exp` (for variables `x` and `a`). Similarly `geta` needs access to the activation frame of the enclosing `f` (for `i`) and of the enclosing `exp` (for `a`).

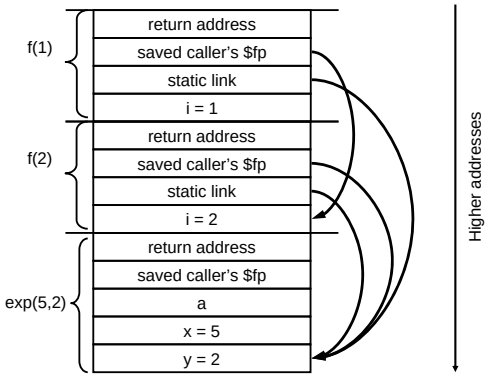
One way to implement this feature is to use a different type of activation record for nested functions (as opposed to the activation record used for methods). This new activation record contains an extra entry known as a static link, stored in between the saved frame pointer and the first argument. The static link is a pointer to the active activation record of the *nearest statically enclosing function/method*.

The first three activation records for a method call to `exp(5,2)` are given below (one for `exp(5,2)`, one for the function call to `f(2)`, and one for `f(1)`). In the diagram, we have noted with an arrow that the static links for `f(2)` and `f(1)` point to the begining of the activation frame for `exp(5,2)`.

(d) Complete the stack of activation records at the time of the call to `geta()` for `exp(5,2)` (i.e., having three calls to `f`). Include the activation record for `geta()`. Then, draw arrows to show where the static links and saved frame pointers point.

Solution: A diagram containing both

- the AR for `f(0)` on top of the AR for `f(1)` with a static link to the bottom of the AR for `exp(5,2)` and
- the AR for `geta()` on top of the AR for `f(0)` with a static link to the bottom of the AR for `f(0)`.



(e) Generate MIPS code to store the result of the assignment `a <- a * x` in function `f`. Assume that the code for the multiplication has already been generated with the result in `$a0`. (Hint: refer to the diagram given on the previous page. You can use temporaries to generate code.)

Solution:

```
Math.exp_f:
    ...# code to compute a * x leaving the result in $a0
    # load the static link in a temporary (say $t1) using $fp
    # store $a0 in a using $t1
    ...
```

(f) Complete the generated MIPS code for geta(). Return the result in register $a0. The function prologue and epilogue has been provided for you. (Hint: use your completion of the above diagram.)

Solution:

```
Math.exp_f_geta:
    addiu $sp $sp -8 # room for the FP and RA
    sw $fp 8($sp)
    sw $ra 4($sp)
    addiu $fp $sp 4

    # load the static link to f(0) in $a0 using $fp
    # load i using $a0
    # load the static link to exp(5,2) in $a0 using $a0
    # load a using $a0
    # add a and i and store the result in $a0

    lw $ra 0($fp)
    lw $fp 4($fp)
    addiu $sp $sp 12 # pop the static link as well
    jr $ra
```

(g) Complete the generated MIPS code for the call to geta() from f.

```
Math.exp_f:
    ...

    # push static link ($fp)
```

Solution:

```
    jal Math.exp_f_geta
    ...
```

(h) Complete the generated MIPS code for the call to f(i - 1) from f. Assume that the code for the subtraction has already been generated with the result in $a0.

```
Math.exp_f:
    ...# code to compute i - 1 in $a0

    # push $a0
    # load static link in $a0 using $fp
    # push $a0
```

Solution:

```
    jal Math.exp_f
    ...
```

(i) How many loads are required for reading a variable in a nested function?

Solution: The number of enclosing functions plus one.

(j) Describe an alternative compilation strategy that reduces the number of loads required for a variable access.

Solution: Store a static link to each enclosing function in the AR.

4 Short Answers (20 points)

(a) Recall that in Cool (and many other languages), objects are stored in the heap. Explain why can't the objects be stored in the activation frame.

Solution: Objects must be able to outlive the current invocation

(b) Give an example of statically typed language and one of a dynamically typed language.

Statically typed: **Solution**: Cool Dynamically typed: **Solution**: Scheme

(c) What is a limitation of a dynamically typed language?

Solution: Slow run-time checks, bugs are found very late.

(d) What is a limitation of a statically typed language?

Solution: Some valid programs are rejected by the type checker.

(e) Consider a data-flow analysis that marks all invocations of `new` that create objects whose field f is eventually referenced in the rest of the execution. Would this be a forward or a backward analysis?

Solution: Backward

(f) What is the most general condition under which $T_2 \sqcup T_3 \leq T_2$, where $\sqcup$ is the least upper bound operator on Cool types ?

Solution: $T_3 \leq T_2$