Midterm II (Solutions)
CS164, Spring 2001

April 10, 2001

- Please read all instructions (including these) carefully.
- There are 8 pages in this exam and 5 questions, each with multiple parts. Some questions span multiple pages.
- 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. Partial solutions will be graded for partial credit.

NAME: _____

SID#: _____

email address: cs164-_____

Circle the time of your section: 9:00 10:00 11:00 12:00 1:00 2:00 3:00 4:00

Problem	Max points	Points
1	15	
2	15	
3	25	
4	25	
5	20	
TOTAL	100	

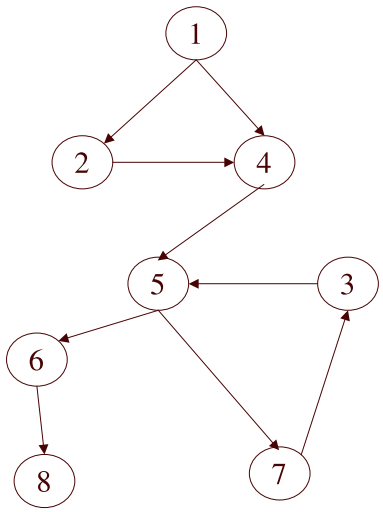
1 Control-Flow Graphs

Consider the following three-address code fragment:

```
        b <- 1
        b <- 2              (1)
        if w <= x goto B
 ---------------------------
        e <- b              (2)
        jump B
 ---------------------------
A:      jump D              (3)
 ---------------------------
B:      c <- 3
        b <- 4              (4)
        c <- 6
----------------------------
D:      if y <= z goto E    (5)
----------------------------
        jump end            (6)
-------------------------
E:      g <- g + 1
        h <- 8              (7)
        jump A
-------------------------
end:    h <- 9              (8)
```

a) Draw horizontal lines to separate basic blocks in the code fragment above. Number each basic block. We suggest that you number starting with #1 at the top of the page.

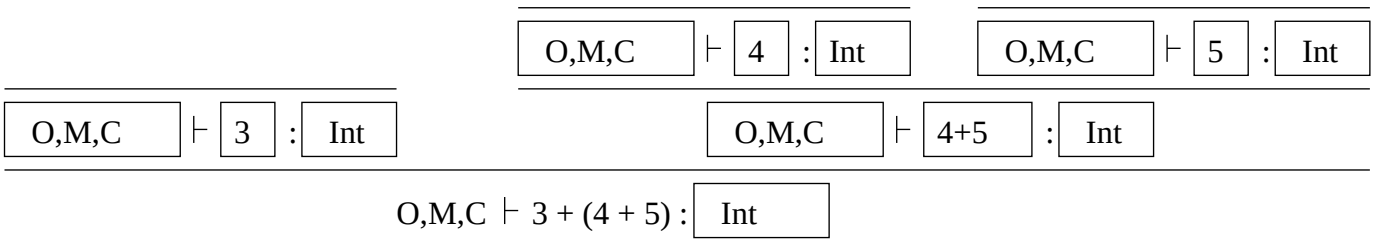
b) Draw a control-flow graph for the fragment above. Use the numbers you assigned in part (a) to label the CFG nodes.



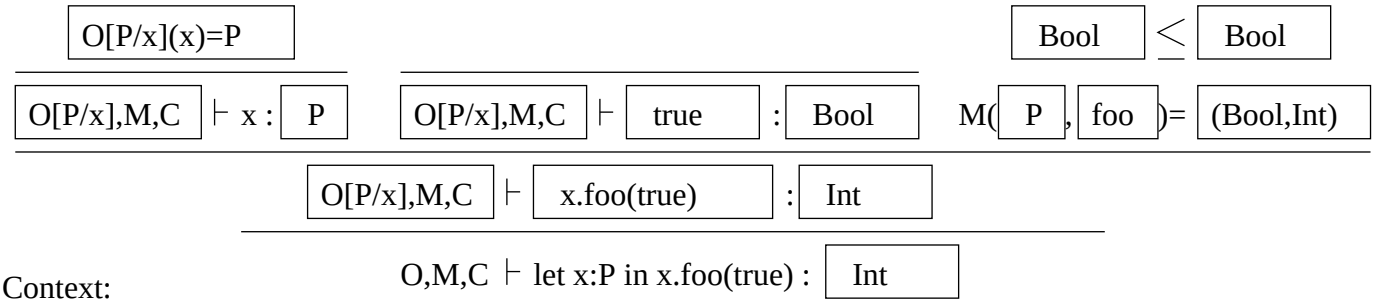
2 Typing and Operational Semantics

Fill in the typing derivation trees below. Every box should be filled in. You do not need to write anything outside the boxes. Where necessary, contextual information appears below the problem.

a) Typecheck `3 + (4 + 5)`.

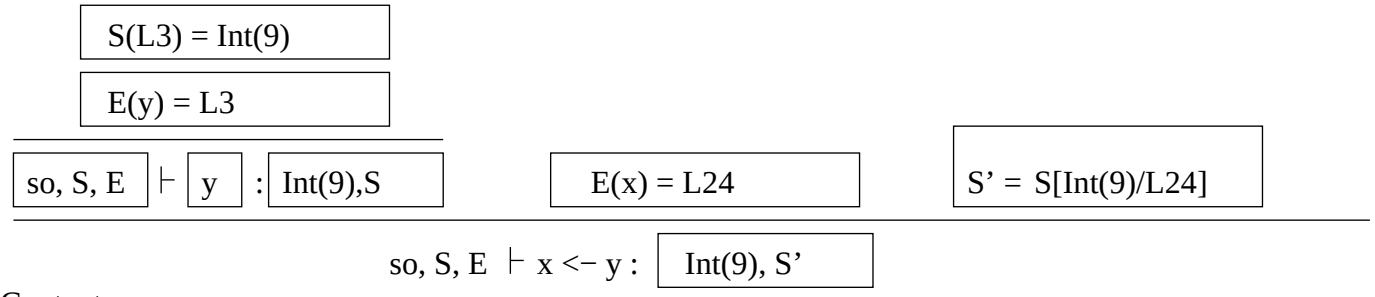


b) Typecheck `let x:P in x.foo(true)` in the given context.



Context:
M(P,foo) = (Bool, Int)

c) Evaluate `x <- y` in the given context using Cool's **operational semantics**.



Context:
E(x) = L24 E(y) = L3 S(L3) = Int(9) S(L24) = Int(6)

3 Typing Rules and Operational Semantics

Consider the following **modified** operational semantics rules for `while` in Cool.

$$\frac{so, S_1, E \vdash e_1 \ : \ Bool(false), \ S_2}{so, S_1, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} \ : \ Int(0), \ S_2} \quad \text{[Loop-False]}$$

$$\frac{\begin{array}{c} so, S_1, E \vdash e_1 \ : \ Bool(true), \ S_2 \\ so, S_2, E \vdash e_2 \ : \ v_2, \ S_3 \\ so, S_3, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} \ : \ v_3, \ S_4 \end{array}}{so, S_1, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} \ : \ v_2, \ S_4} \quad \text{[Loop-True]}$$

a) In one or two sentences, explain the return value of this new `while` expression.

Solution: The return value is the integer 0 if the loop body does not execute, or the value of the loop body in the first iteration if the loop body executes at least once.

b) Give a typing rule for this new `while` expression. The rule must preserve type safety of Cool and must accept as many programs as possible.

Solution

$$\frac{O, M, C \vdash e_1 : Bool \quad O, M, C \vdash e_2 : T}{O, M, C \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int \sqcup T}$$

c) Now imagine that we want a `while` loop that counts its number of iterations. The value of such a `while` loop should be an integer object that holds the number of times the body has been evaluated. Give the typing rule for this new `while` expresion.

Solution:

$$\frac{O, M, C \vdash e_1 : Bool \quad O, M, C \vdash e_2 : T}{O, M, C \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int}$$

d) Give the operational semantics rules for the "counting" `while` described in part (c).

$$\frac{so, S_1, E \vdash e_1 : false, S_2}{so, S_1, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int(0), S_2}$$

$$\frac{so, S_1, E \vdash e_1 : true, S_2 \quad so, S_2, E \vdash e_2 : v, S_3 \quad so, S_3, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int(n), S_4}{so, S_1, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int(n+1), S_4}$$

4 Runtime Organization

Recall the following MIPS instructions:

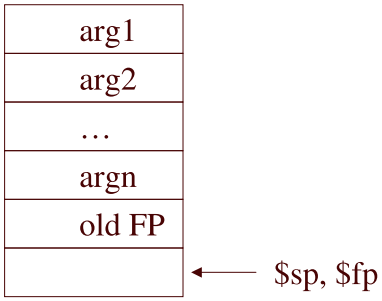
- `sw $a0 n($sp)` - store the value of the accumulator at address `n` + `$sp`
- `lw $ra n($sp)` - load the return address register with the value stored at address `n` + `$sp`.
- `addiu $sp $sp n` - adjust the value of the stack pointer by `n`
- `move $a0 $t1` - move the contents of `$t1` into `$a0`
- `jal f` - jump to addres `f` and store the address of the next instruction in register `$ra`
- `jr $ra` - jump to the address stored in register `$ra`

Imagine a compiler that uses a variation of the calling convention that we used in class. We give below the code that this compiler generates to setup the activation record of a function and also the code that is used to access the i^{th} ($i = 1, \ldots, n$) formal argument of a function with n arguments. The caller pops the arguments off the stack in this calling convention. On entry to a function the return address is in `$ra` and on exit the result value is in `$a0`.

```
cgen(def f(x1, ..., xn) = e) =
    sw      $fp 0($sp)
    addiu   $sp $sp -4    # Push the old FP on the stack
    move    $fp $sp       # Set the new FP
    cgen(e)               # Evaluate the body of the function
    ...                   # Code to return

cgen(xi) =                # get the ith argument
    lw      $a0 z($fp)    # z = 8 + 4 * (n - i)
```

a) Draw the stack contents at a point where the function `f` has been invoked and its execution is just before the code that evaluates the function body (the `cgen(e)` above). Put larger addresses at the top. Mark on the stack the location of the frame pointer, the stack pointer, and the positions where the arguments and the old frame pointer are stored.



(the excitement continues on the next page)

b) Write the code for return (the ... in the above code). To return you must use the instruction `jr $ra` (jump to the address in register `$ra`). Recall that the result value must be in $a0 and the calling function pops the arguments.

```
lw $fp 4($sp)
addiu $sp $sp 4
jr   $ra
```

c) Write the code for function call. To perform the actual call use the instruction `jal f` (jump to the address of function `f` and save the return address in register `$ra`).

```
cgen( f(e1, e2, ..., en) ) =

  sw  $ra 0($sp)      # Save the return address, since
  addiu $sp $sp -4    # cgen(ei) and jal below can clobber it
  cgen(e1)            # Eval args in order 1 -> n
  sw $a0 0($sp)       # Push them on stack
  addiu $sp $sp -4
  cgen(e2)
  sw $a0 0($sp)
  addiu $sp $sp -4
  ...
  cgen(en)
  sw $a0 0($sp)
  addiu $sp $sp -4
  jal   f             # function call
  addiu $sp $sp 4*n   # Pop the arguments
  lw $ra   4($sp)     # Reload the saved $ra
  addiu $sp $sp 4     # fix $sp
```

5 Global Optimization

In Cool, new objects are always allocated on the heap and their storage is recoved by a garbage collector. Experimental results suggest that many objects are used only during the evaluation of the method body that creates the object. Such objects could be allocated on the current stack activation record and recovered inexpensively when the activation record is destroyed.

In this exercise you will write the transfer functions for the global escape analysis needed to discover such objects.

Consider the following intermediate language:

```
x   <- y                    # Assignment
x   <- y.m(x1,...,xn)       # Method invocation
x   <- new(C)               # Create a new object (of some class C)
x.f <- y                    # Field write
if x = y jump L             # Conditional jump
jump L                      # Unconditional jump
```

(in these instructions `x` and `y` are temporary variables, `f` is a field name and `m` is a method name).

We say that an object **can escape** if any of the following actions are performed on it:

1. It is written into the field of an object, or
2. It is passed as an argument to a method, or
3. It has a method invoked on it (e.g., "`y`" in "`y.m(...)`").

We define E_{out}(`x`,`s`) = "true" if the object refered to by `x` immediately after `s` can escape in the computation that follows `s`. Conversely, we say that E_{out}(`x`,`s`) = "false" if no matter what path is taken after `s` the object refered to by `x` cannot escape.

a) Circle each call to `new()` that creates an object that can escape in the following code fragment:

```
a   <- new(A)
b   <- new(B)
c   <- new(C)
d   <- new(D)
x   <- a
y   <- c
x.f <- d
x   <- y.m(a)
```

Solution

new(A) Escapes at y.m(a)

new(C) Escapes at y.m(a) because of y ← c

new(D) Escapes at x.f ← d

(problem continues on next page)

Next you will have to write the transfer functions for the values E_{in} and E_{out}. Your answers should be in terms of other values of E_{in} and E_{out} and constants like true and false. In these examples, `x`, `y` and `z` are distinct variables.

b) E_{in}(`x, y <- x.m(z)`) =

Solution = true

c) E_{in}(`x, y.f <- x`) =

Solution = true

d) E_{in}(`x, y <- z`) =

Solution = E_{out}(`x, y <- z`)

e) E_{in}(`x, x <- new(C)`) =

Solution = false

f) E_{in}(`x, y <- x`) =

Solution = E_{out}(`x, y <- x`) $\vee$ E_{out}(`y, y <- x`)

g) E_{out}(`x, s`) = (`s` is an arbitrary statement)

Solution = $\bigvee_{j \in successors(s)} E_{in}$(`x, j`)

h) (Circle one) Is this a *forward* or *backward* analysis?

Solution Backward