Midterm II
CS164, Spring 2000

April 11, 2000

Problem #1:: Typing and Code Generation (15 points)

Assume that we extend the Cool language with a new looping construct "do e1 until e2". The intended execution of this construct is that the expression e1 (the loop body) is evaluated by e2 (the terminal condition). If the value of e2 is false then the execution of the loop body starts again with e1. Otherwise the execution of the loop terminates and the value of the expression is the value of e1 in the last iteration

* Give an appropriate Cool typing rule for do e1 until e2. To remind yourself of the notation we show below the cool typing rule for if.

$$\frac{\text{O, M, C |- e1 : Bool} \quad \text{O, M, C |- e2 : t2} \quad \text{O, M, C |- e2 : T3}}{\text{O, M, C |- if e1 then e2 else e3 fi : T2 |\_| T3}}$$

* Give Cool operational semantics rules for du e1 until e2. To remind yourself of the notation, we show below one of the two operational semantics rules for if:

$$\frac{\text{so, S1, E |- e1 : Bool (true), S2} \qquad \text{so, S2, E |- e2 : v2, S3}}{\text{so, S1, E |- if e1 then e2 else e2 fi : v2, S3}}$$

* Write the code generation function for do e1 until e2 that generates code for a stack machine with an accumulator. You should only use the following stack-machine instructions:
- "push" - pushes the value of the accumulator on the stack
- "goto L" - jumps to label L
- "iftrue L" tests the value of the accumulator and jumps to label L if the value is true.
- "pop" - copies the value on top of the stack into the accumulator and pops the stack
- "L:" - a label

Remember that after the evaluation of an expression the result must be in the accumulator and the stack height must be as before the evaluation.

stackcgen (do e1 until e2)

Problem #2:: Type Checking (20 points)

The typing rules of Cool contain many details whose presence and correctness is important to achieving a couple of goals:

1. We should not be able to type check programs whose operational behavior is undefined, such as programs that try to invoke a method on an object for which the method is not defined.
2. We should be able to type check all the obviously valid programs

Here is the Cool typing rule for the let construct:

$$\frac{\text{O, M, C |- e1 : T1} \quad \text{T <= T0} \quad \text{O[T0/x], M, C |- e2 : T2}}{\text{O, M, C |- let x : T0 <- e1 in e2 | T2}}$$

We propose below a number of changes to the typing rule for "let". Some of these proposed changes fail to achieve one or both of these two goals. Here is what you must do:

* If goal 1 is violated write (1) along with a one-line Cool expression that type checks (which the proposed change) even though its behavior is different.
* If goal 1 is not violated but goal 2 is violated, write (2) along with a one-line Cool expression that fails to type check (with the proposed change) even though we intednded it to be well typed.
* If neither of the goals are violated by the change, write "none".

class A { a() : Int {0}; }
class B inherits A { b() : Int {1 }; }

* We replace O[T0/x] in the third hypothesis by O.

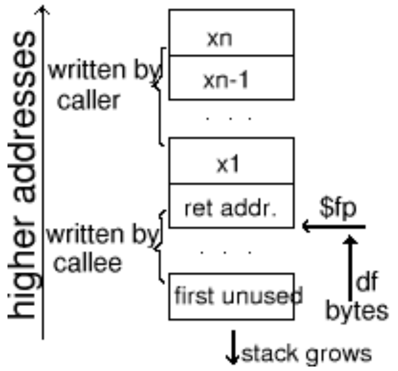
* We replace T2 in the conclusion by T0.

* We remove the second hypothesis T1 <= T0

* We replace the second hypothesis by T0 <=T1

* We replace O[T0/x] in the third hypothesis by O[T1/x]

Problem #3:: Runtime Organization and Code Generation (30 Points)



In this problem you will have to write the code generation fuynctions for generating MIPS assembly using a calling convention different from the one used in the lecture. Here is the sequence of operations that a caller and the called function must perform (as shown in the figure):
* The caller writes the values of the formal arguments x1,...,xn (with x1 at the lowest address). The caller also sets the $fp register to point to the word on the stack right after the arguments, and performs a jal f_entry instruction to transfer control to the called function.
After the call, the caller is responsible for restoring the value of the $fp register
* the called function immediately saves the return address register $ra at address $fp and then proceeds with the valuation of its body. The called function returns by loading the $ra register and performing a jar $ra instruction.
The stack frame layout must be as shown in the figure above. The stack grows towards lower addresses. The major change in strategy is that we are not going to use the stack pointer register anymore. Instead the code generator will keep track at any point in the generated code of the difference between the value of the framne pointer $fp (always pointing to the word where the return address was saved) and the address of the next unused word ont eh stack. The value of this difference is passed to the code generator function as a second argument called df. Thus to write the register $a0 into the next unused word on the stack you can say "sw $a0 -df($fp)".
You will need to use onlky the following kinds MIPS instructions (as described in lecture):
* sw $a0 n($fp) - store the value of the accumulator at address n + $fp
* lw $ra n($fp) - load the return address register withh the value stored at address n + $fp.
* addiu $fp $fp n - adjust the value of the frame pointer by *n*
* jal f_entry - save the return address in register $%ra and jump to the start of the code of function f.
* jr $ra - jump to the address specified in register $ra

Remember that the code that computes an expression should preserve the register $fp and should return the value of the expression in register $a0.

* For example here is the code generation rule for e1 + e2

```
cgen (e1 + e2, df) =
  cgen (e1, df)
  sw $a0 -df($fp)
  cgen(e_2,df+4)
  lw $t1 -df($fp)
  add $a0 $t1 $t2
```

* Write the code generation for function call:

cgen (f(e1,...,en) df) =

*Write the code generation for accessing a parameter:

cgen (x1, df)

*Write the code generation for a function body:

cgen(def f(x1,...,xn) = e,0)=

Problem #4:: Global Analysis And Optimization (35 Points)

In this problem we will work with programs in intermediate-code form using the following address instructions

```
x := y op z
x := y
if x = y goto n
goto n
```

where := denotes assignment, op is some binary arithmetic operator, and *n*is an instruction address. The instructions are considered arranged in a sequence and we will use the index in the sequence as an instruction identifier.

Basic Blocks and Control Flow Graphs

Consider the following intermediate code program:

```
1  b:= 2*a
2  d:=b + c
3  a:=a + 1
4  e:= a*b
5  f:=d
6  if e = c goto 9
7  c:= a * b
8  f := b + c
9  a := 1
```

* Draw horizontal lines across the program to separate it into basic blocks

* Draw the control-flow graph of the above program. Depict each basic block as a box and write it in the number of the instructions contained in that basic block.

4.1 Global Analysis: Availabe expressions

We say that an expression y op z is available in variable x at some point in the program if no matter how the execution reaches that point the value of the variable is the same as that of the expression y oi z. We write "x = y op z" to denote such an available expression.
For each instruction we define two sets,e ach containing elements of the form "x = y op z" :
* In (i) - The available expressions right before insturction i is executed
*Out(i) - the availabe expressions right after instruction i is executed
Here are a few examples of such sets for our program:
* In (1) = { }
* In(2) = {b = 2 * a}
* In(3) = {b = 2 * a, d = b + c}
* Out(3) = {d = b + c} ("b = 2 * a" does not necessarily hold anymore and neither does "a = a+1" because of the assignment to a.)
* Out(5) = {d = b+c, e = a*b, f = b+ c} (the last item is due to the assignment "f := d"; f = d is not availabe because it is not of the form x = y op z)
* In (9) = {f = b + c, e = a * b} ("d = b + c" does not hold because it does not hold after instruction 7.)
Your firs6t task is to write down the rules for computing the In and Out sets. You can use the set union, interasection and difference notations. You can also write "N o(x,S)" to denote the subset of elements in S that do not mention the variable x.
* Write the definition of In(i) as a function of the sets Out (j1),...,Out(jn) where the instructions j1...jn arre the predecessors of i on somme path through the program.
In(i) =
*Write the definition of Out(i) as a function of In(i) if the instruction i is x := y op z, and x is distinct from both y and z
Out(i)=

*Write the definition of Out(i) as a function of In(i) if the instruction i is x := x op y.
Out(i)=

*Write the definition of Out(i) as a function of In(i) if the instruction is x:=y
Out(i)=

*Write the definition of Out(i) as a function of In(i) if the instruction i is goto n
Out(i)=

Optimization
The sets In(i) can be used to detect both common subexpressions and equality of variables at all points in the program. Describe how you can use the set In(i) to optimize the instruction i in each of the following cases.
*The instruction i is x := y op z

* The instruction i is x := y

*The instruction i is if x = y goto n