Midterm II Solutions
CS164, Spring 2003

April 15, 2003

- Please read all instructions (including these) carefully.
- **Write your name, type your login and circle the section time.**
- There are 8 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.

NAME and LOGIN: _____

SID or SS#: _____

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

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

1 Local Optimizations (20 points)

All operations in this question are on integer values. You may ignore arithmetic overflow issues.

a) Apply common sub-expression elimination to the basic block below and write the final version of the changed instructions in the space provided to the right. **Do not** copy the instructions that remain unchanged. Do not apply other optimizations.

Common Sub-Expression Elimination

Solution:

```
g := a + n

t := b * c

u := a + n          u := g

t := a * a

d := b * c

a := 2 + a

h := b * c          h := d

f := a + n
```

b) Fill in the sets of live variables at all program points in the basic block below. As you compute the live variables, cross out the dead instructions and **do not** consider them in the computation of the remaining live variable sets. The set of live variables at exit of the block is $\{s, e\}$.

Solution:

```
                {x}
b := x * 0
                {}
~~a := a + 1~~
                {b}
n := b
                {}
~~g := a + n~~
                {}
~~u := a + n~~
                {b, n}
a := b ** 2
                {}
~~g := b * b~~
                {a, n}
e := a + n
                {e}
s := 2 + 3
                {s, e}
```

c) Apply copy propagation, constant folding and algebraic simplification to the following code fragment as many times as possible. Write the final version of the changed instructions in the space provided to the right. **Do not** copy the instructions that remain unchanged. Do not apply other optimizations.

Solution:

```
c := 3

o := a

m := b + o          m := b + a

p := c * c          p := 9

i := x * 0          i := 0

n := m * 2          m := m << 1 (optional)

e := c + p          e := 12

r := e + n          r := 12 + n
```

2 Global Optimization (30 points)

Code generation and optimization are usually done using only the static (declared) type of an object. In this question we will explore a simple program analysis that attempts to determine, for each temporary variable at each program point, the dynamic (run-time) type of the object the variable refers to.

We will consider the following subset of an intermediate language for an object-oriented programming language similar to Cool:

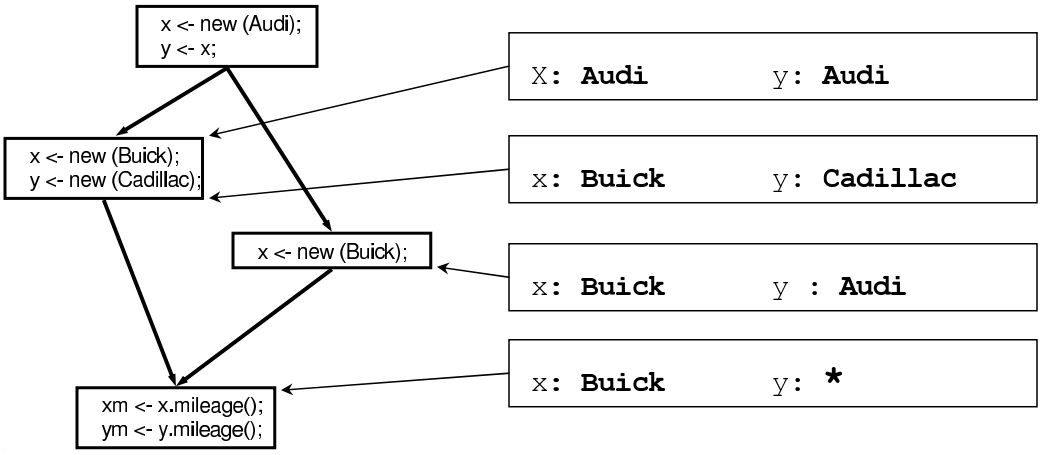
```
x <- y                    # Assignment
x <- new(C)               # Create a new object of class C
x <- y.m(z1,...,zn)       # Method dispatch
if condition jump L       # Conditional jump
jump L                    # Unconditional jump
```

(Here, `x`, `y`, and $\texttt{z}_i$ are temporary variables and `m` is a method name.)

1. (2 points) Write down the relationship that must always exist between the `static_type(e)` of an expression e, and its `dynamic_type(e)`, in a type-safe language like Cool.

   Solution: `dynamic_type(e)` $\leq$ `static_type(e)`

2. (5 points) In the following control-flow graph, fill in the boxes with the exact dynamic type of variables `x` and `y` at the given program points. If it is not possible to predict the dynamic type of a variable, write `*` in that space. (Both `x` and `y` have static type `Car`. `Audi`, `Buick`, and `Cadillac` are subtypes of `Car`.)



We will now specify the global dataflow rules for this analysis.

3. (2 points) Circle one: This is a [*forwards*] *backwards* analysis.

We will define a transfer function T_{in}(x, s) whose value is C if C is the dynamic type of x at the beginning of statement s, and whose value is * if the analysis is not able to determine the dynamic type. T_{out}(x, s) is defined similarly for the end of s. In this problem we will ignore the "unreachable statement" value #.

4. (15 points) You will now define the transfer function for different kinds of statements. Your answers should be in terms of other values of T_{in} and T_{out} and the values C or *.

   (a) T_{out}(x, x <- new(C)) =
   Solution: C

   (b) T_{out}(x, x <- y) =
   Solution: T_{in}(y, x <- y)

   (c) T_{out}(x, y <- z) =
   Solution: T_{in}(x, y <- z)

   (d) T_{out}(x, x <- y.m(z_1,...,z_n)) =
   Hint: you can assume nothing about the behavior of y.m() except that it has a certain declared (static) return type.
   Solution: * is a good answer. Better (more precise) answers are possible when the static return type has no subclasses or is SELF_TYPE

   (e) T_{in}(x, s) [for any statement s] =
   Solution: $\begin{cases} \texttt{C} & \text{if } T_{out}(\texttt{x}, t) = \texttt{C} \text{ for all } t \in \text{pred}(s) \\ * & \text{otherwise} \end{cases}$

5. (6 points) Describe how you can use the information computed by this analysis to optimize one of the intermediate language instructions considered in this problem.

   Solution: You can use this information to optimize method dispatches, by turning dynamic dispatches y.m(z_1,...,z_n) into static dispatches whenever you know the dynamic type of y at compile-time.

3 Type Checking (20 points)

For this problem consider that you have the following Cool class declarations. Notice that the body of method `a` is not yet filled in. Do not worry about syntactic errors when writing Cool code in this problem.

```
class A {
    a() :  SELF_TYPE  {

      new A  (* filled in for Solution *)

    }
}
class B inherit A {
    b() : Int { 0 }
}
```

a) (7 points) Consider the following modified typing rule for Cool conditionals.

$$\frac{O, M, C \vdash e_1 : \texttt{Bool} \quad O, M, C \vdash e_2 : T_2 \quad O, M, C \vdash e_3 : T_3 \quad T_2 \leq T_3}{O, M, C \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} : T_2}$$

Write below a very short Cool program fragment that uses the classes defined above and typechecks with the modified rule for conditionals, but attempts to invoke the method `b` on an object of dynamic type `A` at run time.

Solution:

One way to do it is:

```
(if false then new B else new A).b()
```

b) (13 points: 2 points for circling the right rule, 4 points for filling in the function body, 7 points for well-typed code that causes a type error) Exactly one of the following Cool subtyping rules is sound for all classes C. Circle the **unsound** rule:

$$\boxed{C \leq \texttt{SELF\_TYPE}_C} \qquad\qquad \texttt{SELF\_TYPE}_C \leq C$$

To demonstrate that the rule you **circled** is **unsound**, fill in the body of method `a` above and write a short Cool program fragment that typechecks with the **unsound** subtyping rule and generates a type error at run-time.

Solution:

The simplest way to do it is:

```
(new B).a().b()
```

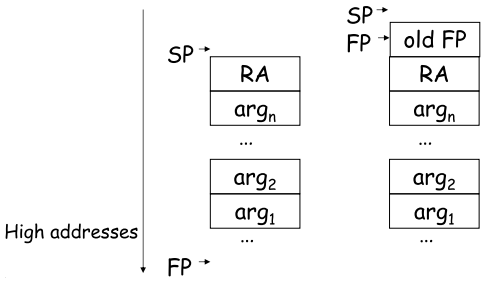
You'll also need to fill in `new A` above (or something else that has dynamic type `A`).

4 Code Generation (30 points)

For this problem you will have to implement code generation for function calls and returns and for formal argument access in a body of a function. You are given the fact that the stack contents at the time when a function with n arguments is called is as shown in figure below. The stack grows to lower addresses and the stack pointer always points to the first unused word on the stack (i.e., the return address to be used by the called function is on top of the stack). Except for the stack pointer, there is no other information passed in registers from the caller to the called function.

You will use the frame pointer (FP) to access the formal arguments while executing the body of the function. The value of the frame pointer during the execution of the called function should be equal to the value of the stack pointer on function entry. Part of your job is to implement all the manipulations of the frame pointer.

As in lecture, the evaluation of an expression must leave the stack unchanged and the result of the expression in register `$a0`. You can use the register `$t0` as a temporary register if you need one.



In your answer you can use any MIPS instructions, but we recommend you use the following abbreviations:

- `push r` for `sw r, 0($sp); subiu $sp, $sp, 4` (push the contents of register r onto the stack),
- `r := pop` for `addiu $sp, $sp, 4; lw r, 0($sp)` (pop the stack and move its top into register r)
- `la r, Label` to load the address of a label into register r. Use this instruction to get the return address for a call site. Do not use the `jal` instruction since it is the responsibility of the caller to save the return address.
- `jr r` to jump to the address contained in register r.

a) (5 points) As lecture, functions are defined using the notation `def f(x1, ..., xn) = ebody` and the body `ebody` of the function can refer only to the formal arguments x_k for $k \in \{1, .., n\}$.

Draw to the right of the above picture the layout of the stack at the moment when we start running the code that evaluates `ebody` for the called function. You need to draw only the new elements on the stack with respect to the layout on function entry. Make sure you show the value of the stack pointer and the frame pointer.

Solution:(see above)

b) (5 points) Write the code generation for accessing the k-th formal argument in a function with n arguments ($k \in \{1, .., n\}$) from within the code that evaluates the function body.

`cgen(xk) =`

Solution:

```
lw z($fp)           # z = 4 * (n - k) + 8
```

c) (8 points) Write the code generation for the body of the function.

`cgen(def f(e1, ..., en) = ebody) =`

Solution:

```
push $fp
addiu $fp $sp 4     # the push makes the $sp off by 4
cgen(ebody)
$fp := pop
lw $t0 4($sp)
jr $t0
```

d) (12 points) Write the code generation for a function call. The layout of the stack on function return must be exactly the same as on function entry.

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

Solution:

```
          cgen(e1)
          push $a0
            ...
          cgen(en)
          push $a0
          la $t0, RetLabel
          push $t0
          j f               # f is the label at the start of f's code
RetLabel: addiu $sp, $sp, z # z = 4 * (n + 1)
```