CS143 Final
Spring 2024

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 180 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason other than to access the class webpage.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

SUNET ID: _____

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	10	
2	25	
3	25	
4	20	
5	20	
TOTAL	100	

1. Operational Semantics

In Cool, it is a fatal runtime error to dispatch on a void value. We will add a void-propagating operator `?.` that acts like a normal method dispatch on non-void values, but produces a void value when applied to a void target.

Examples of new expressions are as follows:

```
e0?.f(e1, ..., en)
```

The void-propagating operator applied to a function call is semantically equivalent to the following code:

```
let v <- e0 in
    if isvoid v then
        void -- a void value of same type as the other branch
    else
        v.f(e1, ..., en)
    fi
```

(a) Below is the operational semantics for regular dispatch. The line $v_0 = X(\ldots)$ implies that v_0 is not `void`. How does this rule need to be changed for the void-propagating operator applied to a function call, for the case where e_0 does not evaluate to `void`?

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 \mapsto v_1, S_2 \\ so, S_2, E \vdash e_2 \mapsto v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n \mapsto v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 \mapsto v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(X, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m},\ x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} \mapsto v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0.f(e_1, \ldots, e_n) \mapsto v_{n+1}, S_{n+4}}$$

Answer:

Yes, we must evaluate e_0 first and use the resulting store for the evaluation of the method's actual arguments. We must also change $e_0.f(e_1, \ldots, e_n)$ to $e_0?.f(e_1, \ldots, e_n)$ in the conclusion (below the horizontal bar).

(b) Give the operational semantics rule for the void-propagating operator for the case where e_0 evaluates to `void`.

Answer:

$$\frac{so, S_1, E \vdash e_0 \mapsto void, S_2}{so, S_1, E \vdash e_0?.f(e_1, \ldots, e_n) \mapsto void, S_2}$$

2. Code Generation

We can remove the use of the frame pointer (FP) from the stack machine design we gave in class. Since the offset of the stack pointer (SP) from the activation record is statically known at every point in the code generator, we can reference temporary values using offsets from the stack pointer.

Consider the Cool code and corresponding assembly on the next page. The assembly is generated via a simple stack machine where the expression `add(r1,r2)` adds the `Int` objects referenced by `r1` and `r2` and stores a reference to the result in `a0`.

Your task is to fill in the blank targets of the load and store instructions on to reference the correct temporaries. You must follow the stack machine rules, so if the same value is stored in multiple places you must choose the one that the stack machine would use.

```
class Main {
    f(x: Object): Int {
        1 + case x of
                i: Int => let y: Int <- 1 in y + i;
                s: String => {
                    s <- "hello";
                    1 + s.length()
                };
            esac
    };
};
```

```
Main.f:
    addiu $sp$ $sp$ -8
    sw $s0 8($sp)
    sw $ra 4($sp)
    lw $a0$ int_const1
    sw $a0 ($sp)
    addiu $sp$ $sp$ -4
    lw $a0 16($sp
    lw $t0 ($a0) # get type tag
    sw $a0 ($sp)
    addiu $sp $sp -4
    # branch if not Int type
    bne $t0 INT_TAG label0
    lw $a0$ int_const1
    sw $a0 ($sp)
    addiu $sp $sp -4
    lw $a0 4($sp
    sw $a0 ($sp)
    addiu $sp $sp -4
    lw $a0 12($s
    lw $t0 4($sp)
    add($a0, $t0)
    addiu $sp$ $sp 4
    addiu $sp$ $sp 4
    b label2
label0:
  # branch if not String type
  bne $t0 STR_TAG label1
  lw $a0 str_const0
  sw $a0 4($sp
  lw $a0 int_const1
  sw $a0 ($sp)
  addiu $sp$ $sp$ -4
  lw $a0 8($sp
  jal String.length
  lw $t0 4($sp)
  add($a0, $t0)
  addiu $sp $sp 4
  b label2
label1:
  jal case_abort
label2:
  addiu $sp $sp 4
  lw $t0 4($sp
  add($a0, $t0)
  addiu $sp $sp 4
  lw $ra 4($sp)
  lw $s0 8($sp)
  addiu $sp $sp 8
```

3. Language Design and Type Checking

We will extend the COOL language with the concept of arrays. Arrays in cool are fixed-length mutable sequences. Arrays are constructed by a comma-separated list of expressions surrounded by square brackets, e.g., `[1, x, 42]`. Arrays can be concatenated using the syntax `array1 + array2`, which produces a new array with the values of `array1` followed by the values of `array2`. Finally, arrays can be indexed using the syntax `array1[3]` which accesses the fourth element of of `array1`.

The array `[T1, T2, ..., Tn]` has the static type `[T]` where `T` is the least upper bound of the array element types `T1, T2, ..., Tn`.

The following extension to the Cool CFG adds array construction and indexing expressions:

$$
\begin{aligned}
\mathit{expr} ::= & \ldots \\
& \mid \ [\mathit{expr}\ [\![,\ \mathit{expr}\ ]\!]^*] \\
& \mid \ \mathit{expr}\ [\mathit{expr}]
\end{aligned}
$$

(a) Give type checking rules for constructing arrays [Array], indexing into arrays [Array-Index], and concatenating two arrays [Array-Concat]. You do not need to handle invalid array accesses in your type rules.

Answer:

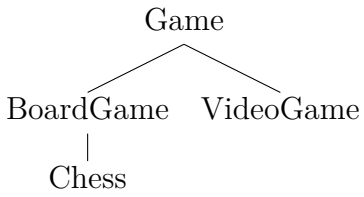
$$
\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T = \bigsqcup_{1 \le i \le n} T_i \end{array}}{O, M, C \vdash [e_1, e_2, \ldots e_n] : [T]} \text{ [Array]}
$$

$$
\frac{\begin{array}{c} O, M, C \vdash e_1 : [T_1] \\ O, M, C \vdash e_2 : Int \end{array}}{O, M, C \vdash e_1[e_2] : T_1} \text{ [Array-Index]}
$$

$$
\frac{\begin{array}{c} O, M, C \vdash e_1 : [T_1] \\ O, M, C \vdash e_2 : [T_2] \\ T = T_1 \sqcup T_2 \end{array}}{O, M, C \vdash e_1 + e_2 : [T]} \text{ [Array-Concat]}
$$

(b) Is array sub-typing sound? That is, if $T_1 \leq T_2$, would it be sound to say the list type $[T_1] \leq [T_2]$? If it is sound, please provide an explanation as to why. If it is not sound, please provide an example where list subtyping would fail.

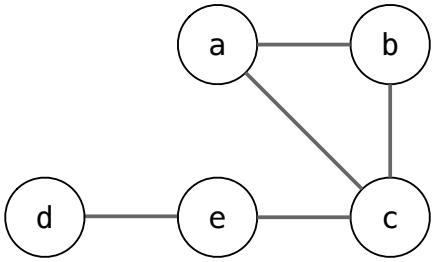
In your explanation, you may use a COOL program with the following class hierarchy to illustrate your point:



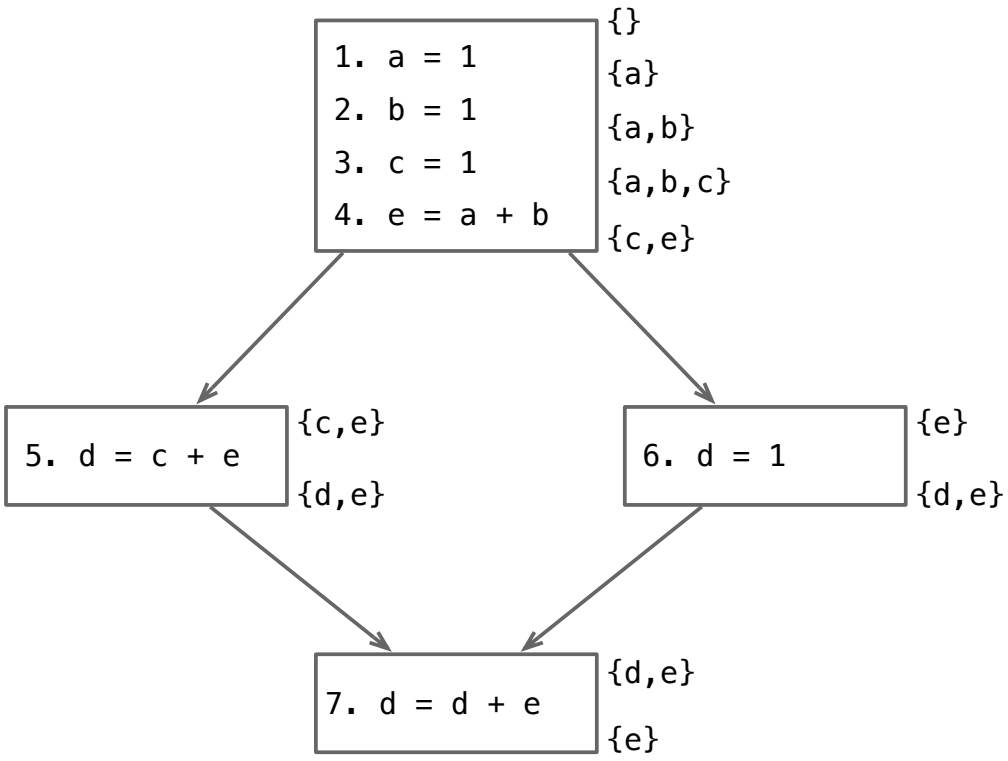
Answer:
No. Suppose you have an array of type [BoardGame] $\leq$ [Game]. Then we can assign a list of type [BoardGame] to a variable of type [Game]. If we then modify the list with an element of type VideoGame, which is legal since VideoGame $\leq$ Game, we have successfully added a VideoGame to an array of dynamic type BoardGame. If we index into the original variable (of type [BoardGame]) we can access the VideoGame as type BoardGame, which would lead to an error at runtime.

4. Register Allocation

Consider the following register inference graph:



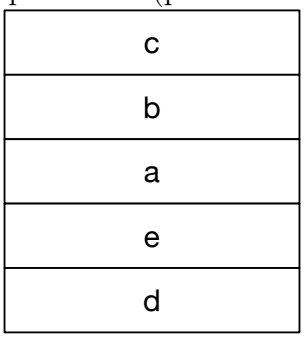
a) Fill in all 7 statements in the below control-flow graph, so that it results in the register interference graph. Use only statements of the form `x = 1` and `x = y + x`, where `x`, `y`, and `z` are variables (i.e., a, b, c, d, e). Assume that no variables are live on entry and that only `e` is live on exit.



b) Using the graph coloring heuristics in lecture 16, give the smallest number of colors k that enable the heuristic to complete without spilling. If there are multiple nodes that could be deleted from the graph, break ties by first selecting a node with the fewest neighbors and second by choosing the node whose label is first in alphabetical order. Using your provided k, give the state of the stack when all nodes have been deleted from the graph.

Value of k: 3

Top of stack (pushed last)



Bottom of stack (pushed first)

c) Provide the register allocation by listing the variables assigned to each register (named `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, and `r7`). Use the minimal number of registers.

Answer:

r1: c
r2: b, d
r3: a, e

5. Optimization

Consider the following three-address IR code for a function `foo`. Assume `a` and `b` are `foo`'s arguments.

```
1  foo:
2      c = 0
3      d = a / 2
4  L1:
5      if c > d goto L2
6      e = b - 1
7      f = e
8      i = e - 1
9      g = e * 4
10     h = i - c
11     k = f * 4
12     j = i - c
13     c = c + 1
14     goto L1
15 L2:
16     ret k
```

(a) Divide the function into basic blocks by giving the line numbers of each basic block. Give the line numbers as inclusive–inclusive ranges and name each block. So, if the second basic block covered lines 10, 11, 12, then you would write: "BB2: 10–12".

Answer:

BB1: 1–3
BB2: 4–5
BB3: 6–14
BB4: 15–16

(b) What are the edges between the basic blocks? Give your answers as (source,sink) pairs. So if there is an edge between BB1 and BB2 from (a), then you would write: "(BB1,BB2)".

Answer:

(BB1, BB2)
(BB2, BB3)
(BB2, BB4)
(BB3, BB2)

(c) What variable(s) (if any), violate the static single assignment (SSA) property?

Answer:

The variable c violates the SSA property, as it is written to twice.

(d) Assume common sub-expression elimination and constant folding have been applied several times (until convergence) to all statements except `c = c+1` on line 13. What variables can be removed by dead code elimination?

Answer:

g, h, i, j

(blank page for extended answers)