CS4610—Programming Languages—Exam 3

Spring 2017

April 27, 2017

- **Write your name and UVa ID on this exam.** Write your UVa ID on every page of this exam in case pages become separated. Pledge the exam before you turn it in.
- This exam has eleven (11) pages (including this one) and six (6) questions. If you get stuck on a question, move on and come back later.
- You have 1 hour and 15 minutes to complete this exam.
- The exam is closed book, but you may refer to your reference sheet. You are allowed two paper-sides of notes (either one page [front and back] or two fronts).
- Use of electronic devices (cell phones, tablets, laptops, etc.) is prohibited for the duration of the exam. Even vaguely looking at a device **is cheating**. Avoid questionable situations by turning all of these devices off and placing them underneath your chair.
- 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 question has a relatively straight-forward answer. Points may be deducted for both difficult-to-read answers and also solutions that are overly complicated.
- Partial credit will be awarded for partial answers. If you leave a non-extra-credit portion of the exam blank, **you will receive one-third of the points for that small portion (rounded down)**. If you wish to make an answer to a sub-question blank, draw an "X" through the portion you wish to be blank. **DO NOT** cross out portions of the exam until you near the end because this mark cannot be undone.

UVa ID: KEY

Name (print): KEY

UVa ID (again): _____

Problem	Max Points	Points
1—Parameter Passing	12	
2—Optimization	18	
3—Garbage Collection	17	
4—Concurrency	15	
5—Operational Semantics	20	
6—Debugging, Opsems, Types	18	
Extra Credit	0	
TOTAL	100	

Honor Pledge:

How well do you think you did? _____

1. Parameter Passing (12 points)

Suppose that the following pieces of code are valid in some language.

```
1 int f(int a, int b, int c) {
2     c = a + b;
3     a = a + c;
4     b = b + c;
5     return c;
6 }
```

```
1 void main() {
2     int a = 2, b = 4, c = 1;
3     c = f(a, b, c);
4     print a, b, c;
5 }
```

(a) (6 points) If all the parameters are passed *by value*, what would this code print, and why? Limit your answer to a maximum of five sentences.

The output should be: 2 4 6. In call by value, arguments are computed before the call, and the results of these computations are passed to the function.. So what is passed could be considered a "copy" of the variables. Therefore, what occurs in the called function does not effect the state f the calling function.

(b) (6 points) If all the parameters are passed *by reference*, what would this code print, and why? Limit your answer to a maximum of five sentences.

The output should be: 8 10 6. When parameters are passed by reference, the actual parameters in the function are aliases of the calling function's variables. When a variable is updated in the callee, the value is also updated in the caller, since both bindings are to the same memory location.

2. Optimization (18 points)

(a) (10 points) Consider the following three-address code. Note that the ** operator indicates exponentiation.

```
a <- x ** 0
b <- a ** 2
c <- x
d <- b * b
e <- a + c
f <- b + c
g <- e + f
return g
```

Use the following local optimizations we discussed in class to improve the code. Use only a single optimization at a time, and indicate which optimization you used. Stop when there are no more optimizations to perform.

- Algebraic simplification
- Constant folding
- Common subexpression elimination
- Copy propagation
- Dead code elimination

Algebraic Simplification

```
a <- 1
b <- a * a
c <- x
d <- b * b
e <- a + c
f <- b + c
g <- e + f
return g
```

Copy Propagation

```
a <- 1
b <- 1 * 1
c <- x
d <- b * b
e <- 1 + x
f <- b + x
g <- e + f
return g
```

Constant Folding

```
a <- 1
b <- 1
c <- x
d <- b * b
e <- 1 + x
f <- b + x
g <- e + f
return g
```

Copy Propagation

```
a <- 1
b <- 1
c <- x
d <- 1 * 1
e <- 1 + x
f <- 1 + x
g <- e + f
return g
```

Constant Folding

```
a <- 1
b <- 1
c <- x
d <- 1
e <- 1 + x
f <- 1 + x
g <- e + f
return g
```

Common Subexpression Elimination

```
a <- 1
b <- 1
c <- x
d <- 1
e <- 1 + x
f <- e
g <- e + f
return g
```

Copy Propagation

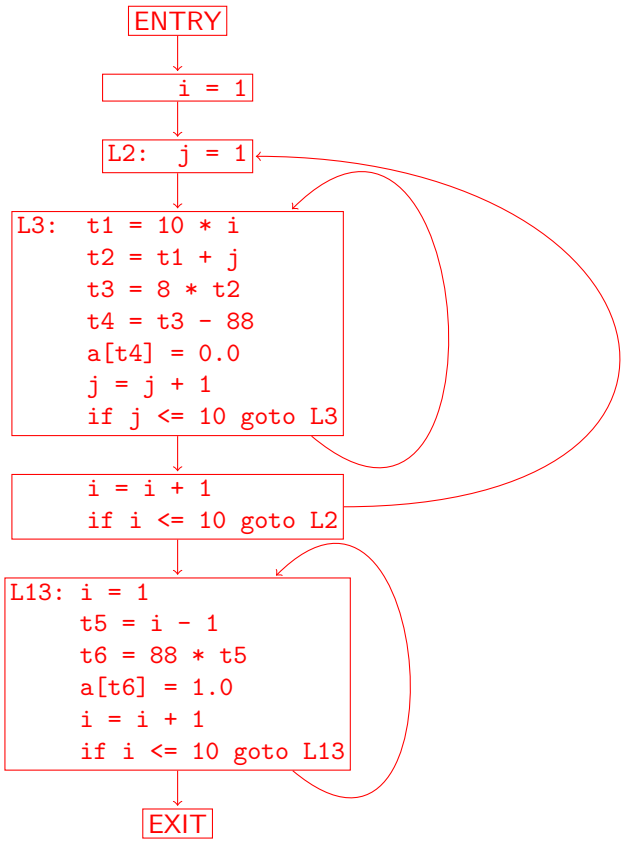
```
a <- 1
b <- 1
c <- x
d <- 1
e <- 1 + x
f <- e
g <- e + e
return g
```

Dead Code Elimination

```
e <- 1 + x
g <- e + e
return g
```

(b) (8 points) Draw a control-flow graph for the following code. Each node in your control-flow graph should be a *basic block*. Do not worry about static single assignment (SSA) form. *Every* statement should appear somewhere in your control-flow drawing.

```
     i = 1
L2:  j = 1
L3:  t1 = 10 * i
     t2 = t1 + j
     t3 = 8 * t2
     t4 = t3 - 88
     a[t4] = 0.0
     j = j + 1
     if j <= 10 goto L3
     i = i + 1
     if i <= 10 goto L2
L13: i = 1
     t5 = i - 1
     t6 = 88 * t5
     a[t6] = 1.0
     i = i + 1
     if i <= 10 goto L13
```



3. Garbage Collection (17 points)

(a) (7 points) George and Julia are discussing the C programming language while eating lunch on the Downtown Mall. "When we get back from lunch," says George, "I'm going to successfully implement stop-and-copy garbage collection for C." Julia scoffs. "There's *no way* that's going to work," she says while rolling her eyes. Who is right and why? Limit your answer to at most 5 sentences.

Julia is correct. Stop-and-copy garbage collection requires the algorithm to rewrite pointer values. In C, it is not necessarily possible to know which values will be used as pointers (i.e. integer values can be treated as pointers).

(b) (10 points) Consider a version of the Cool language that implements mark and sweep garbage collection. Recall that Cool objects may contain pointers to other Cool objects (i.e. attributes). Use the following heap memory layout containing Cool objects (each object takes up one "memory location") to answer the sub-questions. Objects of type `Foo` have two attributes: `Val` and `Next`.

0	1	2	3	4	5	6	7	8	9	10
Type: Foo *Val:* 1 *Next:* 4	*Type:* Int `Int(2)`	*Type:* Foo *Val:* 3 *Next:* 4	*Type:* Int `Int(3)`	*Type:* Foo *Val:* 5 *Next:* 6	*Type:* Int `Int(4)`	*Type:* Foo *Val:* 7 *Next:* 8	*Type:* Int `Int(5)`	*Type:* Foo *Val:* 9 *Next:* 10	*Type:* Int `Int(6)`	*Type:* Void `Void`
×	×			×	×	×	×	×	×	×

i. (5 points) Suppose there is only one activation record, and it contains just a single Cool object (with a single attribute) as defined below. Draw an × in the empty space below each cool object that would be marked by the garbage collector in the diagram above (i.e. mark each reachable memory location).

Type: Start *Ptr:* 0

ii. (5 points) In the space below, draw the heap memory layout after the sweep phase of garbage collection. How many memory locations did you free?

0	1	2	3	4	5	6	7	8	9	10
Type: Foo *Val:* 1 *Next:* 4	*Type:* Int `Int(2)`			*Type:* Foo *Val:* 5 *Next:* 6	*Type:* Int `Int(4)`	*Type:* Foo *Val:* 7 *Next:* 8	*Type:* Int `Int(5)`	*Type:* Foo *Val:* 9 *Next:* 10	*Type:* Int `Int(6)`	*Type:* Void `Void`

Mark and sweep was able to free up 2 memory locations.

4. Concurrency (15 points)

(a) (5 points) Describe the difference between *threads* and *tasks* with respect to representation of concurrency in programming languages. Provide an example of a language or language extension that uses each abstraction.

A thread is an active entity that is running concurrently with other entities in the same program. Java (and OpenMP) use notions of threads.

A task is a well-defined unit of work that must be performed by some thread. Cilk is one example of a language that uses task abstractions.

(b) (5 points) Why are big-step operational semantics (like those used to formalize Cool operational semantics) not good enough to model concurrent execution?

Big-step operational semantics are too course-grained to model intermediate states within a program. For example, the single application of the `while` rule represents the entire execution of the loop. This make representing the interleaving of two concurrent computations challenging, if not impossible.

(c) (5 points) Describe one similarity and one difference between OpenMP and Cilk. Limit your answer to six sentences.

There are many options to choose from here. A few are listed below.

Similarities: Both languages provide notions of serial semantics (i.e., if we remove all language-specific keywords, we still have a valid C program). Both are ways of representing concurrency in programs (this doesn't count for full credit). Additionally (though not discussed in class), both OpenMP and Cilk can automatically generate a default number of threads (unlike, say pthreads where the number of threads is explicit, the runtime can choose a default value).

Differences: Cilk is a programming language while OpenMP is a langauge extension. Cilk adds keywords directly to the C language while OMP relies on compiler pragmas. OMP is used to specify physical parallelism (threads), but Cilk defines logical parallelism (tasks). OMP uses Co-Begin for thread/task creations, and Cilk is built on a fork/join model.

5. Operational Semantics (20 points)

(a) (10 points) Consider the following *incorrect* operational semantics rule for Cool `let` expressions.

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ l_{new} = newloc(S_1) \\ so, E, S_1[v_1/l_{new}] \vdash e_2 : v_2, S_2 \end{array}}{so, E, S \vdash \text{let } id : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2} \ (let - init)$$

i. (5 points) Describe why this rule is incorrect.

An updated environment (containing a mapping from id to l_{new}) is not passed to the evaluation of the enclosed expression.

ii. (5 points) In the space below, write a corrected version of the operational semantics rule.

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ l_{new} = newloc(S_1) \\ so, E[l_{new}/id], S_1[v_1/l_{new}] \vdash e_2 : v_2, S_2 \end{array}}{so, E, S \vdash \text{let } id : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2} \ (let - init)$$

(b) (10 points) Cool only supports less-than ($<$) and less-than-or-equal ($\leqslant$) operations. Suppose we wish to add support to greater-than ($>$) comparisons. Write one or more operational semantics rules to describe the evaluation of greater-than evaluations (i.e. given `so`, `E`, `S`, return appropriate values and stores for $e_1 > e_2$). You **may not** use boolean logic (such as *AND*, *OR*, or *NOT*) or mathematical greater-than (or greater-than-or-equal) in your rules.

There are several ways to do this, but the simplest is perhaps:

$$\frac{so, E, S \vdash e_2 < e_1 : v_1, S_1}{so, E, S \vdash e_1 > e_2 : v_1, S_1} \ (greater - than)$$

6. Debugging, Opsems, Types (18 points)

Consider these shown Reason programs that *do not type-check*; the code implicated by the type checker will be highlighted and underlined. Each has English comments explaining what the program *should* do, as well as assertions that *should* type check *and* succeed.

(a)
```
/* "sepConcat sep [s1;s2;s3]" should insert "sep" between "s1", "s2", and "s3", and
    concatentate the result. */
/* Recall that List.fold_left takes a function, an accumulator, and a list as input */
let rec sepConcat = fun sep sl =>
  switch sl {
  | [] => ""
  | [h, ...t] =>
    let f = fun a x => a ^ (sep ^ x);
    let base = [];
    List.fold_left f base sl
  };

assert (sepConcat "," ["foo", "bar", "baz"] == "foo,bar,baz");
```

i. (3 points) Why is `sepConcat` not well-typed?

The base case is of type `list 'a`, but it is being used in conjunction with string concatenation.

ii. (3 points) Describe how you would fix the code so that `sepConcat` works correctly.

`let base = "";`

(b)
```
/* "padZero xs ys" returns a pair "(xs', ys')" where the shorter of "xs" and "ys" has
    been left-padded by zeros until both lists have equal length. */
let rec clone = fun x n =>
  if (n <= 0) {
    []
  } else {
    [x, ...clone x (n - 1)]
  };

let padZero = fun (l1: list int) (l2: list int) => {
  let n = List.length l1 - List.length l2;
  if (n < 0) {
    (clone 0 ((-1) * n) @ l1, l2)
  } else {
    (l1, [clone 0 n, ...l2])
  }
};

assert (padZero [1, 2] [1] == ([1, 2], [0, 1]));
```

i. (3 points) Why is `padZero` not well-typed?

Note that `l2` has type `list int`, which forces `clone` to return an int when using the cons syntax. However, `clone` returns a list.

ii. (3 points) Describe how you would fix the code so that `padZero` works correctly.

Change the highlighted code to use list concatenation (`@`).

(c)
```
/* "mulByDigit d [n1;n2;n3]" should multiply the "big integer" "[n1;n2;n3]"
   by the single digit "d". */
let rec mulByDigit = fun d n =>
  switch (List.rev n) {
  | [] => []
  | [h, ...t] => [ mulByDigit d t , (h * d) mod 10]
  };

assert (mulByDigit 4 [2, 5] == [1, 0, 0]);
```

i. (3 points) Why is `mulByDigit` not well-typed?

The function, `mulByDigit`, returns a list, which is added as the first element of a new list, where the second element is an integer. Lists must be homogeneous in Reason/OCaml.

ii. (3 points) Describe how you would fix the code so that `mulByDigit` works correctly.

This one needs a decent amount of work. Note that you only want to reverse the list once, so we'll need some sort of helper function to do the work once we have the reversed list. Here is one way to redo the program:

```
let mulByDigit = fun d n => {
  let rec helper = fun d n rem => {
    switch(n) {
    | [] => if (rem > 0) { [rem] } else { [] }
    | [hd, ...tl] =>  {
        let v = ((hd * d) + rem) mod 10;
        let new_rem = ((hd*d) + rem) / 10;
        helper d tl new_rem @ [v]
      }
    }
  };
  helper d (List.rev n) 0
};
```

Here is an alternate way that uses list folding:

```
let mulByDigit = fun d n => {
  let (ret, rem) = List.fold_left (fun (acc, rem) e => {
     ([(e*d+rem) mod 10, ...acc], (e*d+rem)/10)
  }) ([], 0) (List.rev n);
  if (rem > 0) { [rem, ...ret] } else { ret }
};
```

Extra Credit (at most 5 points)

(a) (2 points max) Describe how breakpoints can be used to implement "stepping backwards" in a debugger (i.e., After stopping the code, the user should be able to back up to the previous statement and inspect program state.)

(b) (2 points max) What is the difference between Co-Begin and Fork/Join thread/task creation? What is an example of a language/language extension that uses each?

(c) (1 point) What is one piece of advice you wish you were told at the beginning of CS 4610?