CS 558 Programming Languages Winter 2013 – Sample Mid-term Questions with Solutions

The questions on this sample exam are drawn from the mid-terms of previous offerings of this course. Use this sample as a guide to the **style** of questions to expect, but **not** the precise topics to be emphasized. (Also, some of these questions assume fairly detailed knowledge of Java, since that language was formerly used heavily in the homeworks, rather than Python.) The actual exam will have about seven questions.

The exam is open-book, open-notes; you can use any paper reference materials you wish. Computers and other electronic aids are *not* permitted. You must work independently, and you may not share reference materials with other students.

1. Consider the following program, written in Ocaml syntax.

```
let h (z) =
  let a = z in
  let f (x) = x + a in
  let g (y) =
          let a = 10 in
          f (y + a) in
  g (z - 10) in
h(0)
```

(a) What is the value of this expression assuming lexical scoping?

(b) What is the value of this expression assuming dynamic scoping?

Answer: (a) 0 (b) 10

2. Consider the following C program fragment:

```
void f() {
  int a = 2;
}
int g() {
  int b;
  return b;
}
int h() {
  f();
  return g();
}
```

(a) According to the C language definition, it is undefined what result value is returned by `h`. Why?

(b) The code generated by `gcc -O0` actually returns the result value `2` from `h`. Why might this happen?

(c) What would happen if this code were given to a Java compiler? (Assume these are member functions of some class.)

Answer:

(a) The result of `h` is the result of `g`, which is the value of `b`, which was never initialized and is therefore undefined.

(b) In the generated code, the stack location of `b` is the same as the location of local variable `a` in `f`. The call to `f` sets this location to `2`; the code in `g` reads this location and uses it as the value for `b`, which is then returned. (Note: Without knowing what assembly code `gcc` generates, you can't be sure this is why the program behaves as it does, but it's the most probable explanation.)

(c) A Java compiler would reject the program (during semantic analysis), because the use of `b` before it has been defined violates the "definite assignment" property.

3. Consider the following grammar, where the non-terminals are {INT,+,*} and the start symbol is expr :

```
expr := '+' expr expr
      | '*' expr expr
      | INT
```

Is this grammar ambiguous or not? Give brief but convincing evidence for your answer.

Answer: This grammar is not ambiguous. To be ambiguous, a grammar must allow two different parse trees, and hence two different left-most derivations, for the same input string. Therefore, at some point in a left-most derivation, there must be choice of which rule to use to expand the left-most non-terminal. But in this grammar, all the rules match a different initial symbol, so at most one rule can ever be successfully applied at any given point in the input string. Hence the grammar must not be ambiguous.

(Note: In general, it is quite hard to prove non-ambiguity except for very simple grammars like this one; proving ambiguity (by exhibiting two different parse trees for a single input) is much easier.)

4. Consider the following OCaml program:

```
let f(x:int ref, y:int ref) : int ref =
  if !x > !y then x else y
let g(x:int ref) : unit =
  x := !x * 2
let a = ref 1
let b = ref 2
let main() : unit =
   g(f(a,b));
   print_int (!a + !b)
```

(a) What will be printed when `main()` is executed?

(b) Translate this program as closely as possible into *either* C or C++. Treat `a` and `b` as global variables. (Note: If you choose C, you'll need to use pointers. If you choose C++, you may use either pointers or references, whichever you prefer.)

Answer: (a) 5 (b) Here's a C solution (on the left) and a C++ solution using references (on the right):

```
#include <stdio.h>
int *f (int *x, int *y) {
  if (*x > *y)
    return x;
  else
    return y;
}
void g(int *x) {
  *x = *x * 2;
}
int a = 1;
int b = 2;
main () {
  g(f(&a,&b));
  printf ("%d\n", a+b);
}
```

```
#include <iostream.h>
int &f (int &x, int &y) {
  if (x > y)
    return x;
  else
    return y;
}
void g (int &x) {
  x = x * 2;
}
int a = 1;
int b = 2;
main() {
  g(f(a,b));
  cout << a+b << "\n";
}
```

5. The GNU C compiler supports an extension to ANSI C called "statement expressions," which allows any compound statement to be treated an expression by enclosing it in parentheses. For example, we can write

```
#define maxint(a,b) ({ int a1 = (a), b1 = (b); a1 > b1 ? a1 : b1; })
```

Compare this to the more usual C definition

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

Give an example where these two macros behave differently, and explain why `maxint` is probably a better definition?

Answer: Consider passing expressions that have side-effects. For example,

```
maxint(i++,j++)
```

expands to

```
({ int a1 = (i++); b1 = (j++); a1 > b1 ? a1 : b1;})
```

so `i` **and** `j` **are each incremented just once. But**

```
max(i++,j++)
```

expands to

```
( (i++) > (j++) ? (i++) : (j++);)
```

so either `i` **or** `j` **is incremented twice. The former is more likely to be what the user of the macro expected to happen.**

6. **Control Statements**

Consider a counted `for` loop statement, with the general form

```
for var := exp_first to exp_last by exp_step do stmt
```

We can give the semantics for this statement form by giving its translation into lower-level constructs as follows (where `last`,`step`,`top` and `done` are fresh names):

```
      var := exp_first;
      last := exp_last;
      step := exp_step;
top:  if var > last goto done;
      stmt;
      var := var + step;
      goto top;
done:
```

For example, according to this semantics, the program

```
for i := 1 to 10 by 2 do print i
```

would print `1,3,5,7,9`.

(a) Describe what is printed by the following program, based on the given semantics:

```
j := 8;
i := 3;
for i := 1 to j step i do
  begin
    print i;
    j := 5;
  end;
print i;
```

(b) There are many plausible alternatives to the given semantics that make different choices about where and when the bounds and step value get computed. Give such an alternative which would make the program in part (a) prints `1,2,4,8`. (Keep your modification as simple as possible.)

Answer: (a) 1, 2, 3, 4, 5, 6, 7, 8, 9. (b) Don't evaluate exp_{last} and exp_{step} into variables. Instead, re-evaluate them each time around the loop:

```
      var := exp_first;
top:  if var > exp_last goto done;
      stmt;
      var := var + exp_step;
      goto top;
done:
```

7. **Implementing Block Structure**

Many languages support nested blocks that are not procedures, but have local variable declarations, such as the following (valid in C/C++ or Java):

```
{ int i;
  for (i = 1; i < n; i++) a[i] = i;
}
```

A simple way to implement storage for such a block is to allocate it an activation record just as for procedures. Explain why that is not an efficient approach, and describe a better one. Consider what happens if a function has several local blocks. You may wish to draw one or more simple diagrams.

Answer: Allocating an activation record wastes space, since the return address and old frame pointer are not needed. It is simpler just to allocate space for the block's locals at the end of the activation record of the enclosing procedure. If multiple blocks are nested, the storage for the locals in each is just concatenated. If multiple blocks are declared in parallel scopes (not nested), their locals can overlap in the same storage.

8. **Operational Semantics of Boolean Expressions**

Consider a simple language including boolean expressions:

```
exp := ...
     |  '(' 'and' exp exp ')'
     |  '(' 'or' exp exp ')'
```

Short-circuit evaluation of boolean expressions works as follows: the left operand is always evaluated, but the right operand is evaluated *only* if this is necessary to determine the overall value of the expression.

(Note: Most modern languages, including Java, Python, and OCaml, use short-circuit evaluation. For example, evaluating the Java expression `((p != null) && p.b)` can never cause a null pointer violation when trying to dereference `p`, since `p.b` is only evaluated if `p` is non-null.)

Here are operational semantics rules, written in a style similar to that of lecture 3, that describe short-circuit evaluation for `or` expressions.

$$\frac{\langle e_1, E\rangle \Downarrow \langle \texttt{false}, E'\rangle \quad \langle e_2, E'\rangle \Downarrow \langle v_2, E''\rangle}{\langle \texttt{(or } e_1\ e_2\texttt{)}, E\rangle \Downarrow \langle v_2, E''\rangle} \text{ (Or1)}$$

$$\frac{\langle e_1, E\rangle \Downarrow \langle \texttt{true}, E'\rangle}{\langle \texttt{(or } e_1\ e_2\texttt{)}, E\rangle \Downarrow \langle \texttt{true}, E'\rangle} \text{ (Or2)}$$

Write similar rules that describe short-circuit implementation of `and` expressions.

Answer:

$$\frac{\langle e_1, E\rangle \Downarrow \langle \texttt{true}, E'\rangle \quad \langle e_2, E'\rangle \Downarrow \langle v_2, E''\rangle}{\langle \texttt{(and } e_1\ e_2\texttt{)}, E\rangle \Downarrow \langle v_2, E''\rangle} \text{ (And1)}$$

$$\frac{\langle e_1, E\rangle \Downarrow \langle \texttt{false}, E'\rangle}{\langle \texttt{(and } e_1\ e_2\texttt{)}, E\rangle \Downarrow \langle \texttt{false}, E'\rangle} \text{ (And2)}$$

9. **Axiomatic Semantics**

Consider a language that includes `while` statements, assignment statements, and statement composition (written `begin` S_1`;` S_2 `end`). Suppose that the usual proof rules (from lecture and homework) for these statement types are valid for this language.

Suppose we add to the language a C/C++/Java-style `for` statement:

`for (`S_1`;` E`;` S_2`)` S_3

The semantics of this statement are exactly equivalent to

```
begin
  S1;
  while E do begin S3; S2 end
end
```

Here is a valid triple describing the beheavior of a particular program fragment involving `for`:

```
{ x = c }
for (y := 0; x != 0; x := x - 1)
   y := y + 1
{ y = c }
```

(a) Give the strongest proof rule you can for `for` statements. It should be strong enough to justify the above triple. (Hint: The pre- and post-conditions mentioned in your rule should be built from three general propositions P, Q, and R, as well as the expression E.)

(b) What happens when $c < 0$? Why doesn't this make the above triple invalid?

Answer: (a)

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\ \wedge\ E\}\ S_3\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ \texttt{for}\ (S_1;\ E;\ S_2)\ S_3\ \{Q\ \wedge\ \neg\ E\}}$$

(b) There are two reasonable answers. If we assume indefinitely large magnitude integers, then the program will loop forever if $c < 0$. The assertion remains true in the sense of *partial* correctness: it says that *if* the fragment terminates, then `y = c`; it says nothing in the case when the fragment doesn't terminate.

Alternatively, if we assume machine-like integers of fixed size, they will eventually "wrap around," and the fragment will terminate with `y = c`.

10. **Errors**

Consider the following Java code fragment.

```
static int foo(int i,int a[]) {
  String s;
  int j = i - i - 1;
  if (a[j] > 2) then
    s = s + "bar";
  else
    return 0;
}
```

Identify at least three language violations in this code. For each one, indicate whether it is a static error, checked runtime error, or unchecked runtime error.

Answer: There are four violations:

1. **The** `then` **keyword is not used in Java (static error).**
2. **Local variable** `s` **is used before being initialized (static error).**
3. **There is no** `return` **statement for the first branch of the** `if` **(static error).**
4. **Since** `j = -1`**, the reference** `a[j]` **will be out of bounds (checked runtime error).**

Note that Java has no unchecked runtime errors.