Programming Languages: Recitation 4

Goktug Saatcioglu

02.21.2019

1 HW2 Problem 1

- We begin by reasoning about the program if it were using Scala static scoping rules. When the execution of the program starts `main` is called and $x = 2$ and $y = 3$ as this is the declared values of the variables. Thus, the two print statements at line 15 refer to line 2 for x and line 13 for y. Next, we consider the call to `middle` which declares $y = x$ at line 4 and $x = 1$ at line 9. Thus, the two print statements at line 11 refer to line 9 for x and line 4 for y. Finally, the call to `inner` declares $x = 1$ at line 7. The two print statements at line 6 refer to line 4 for y and line 7 for x. Overall, the program will identify two semantic errors in the program. Firstly, in `inner` the declaration at line 7 is valid for the whole block but we try to use x at line 6 before declaring it by trying to print it (leading to a forward declaration error). Secondly, in `middle` the declaration at line 9 is valid for the whole blocl bu we try to use x at line 4 before declaring it by trying to assign it to the declaration of y (again, leading to a forward declaration error). The Scala compiler can give the following error `error: forward reference extends over definition of variable` and point to the line numbers and variables violating the scoping rules.

- We now reason about the program if it were using Java static scoping rules. When the execution of the program starts `main` is called and $x = 2$ and $y = 3$ as this is the declared values of the variables. Thus, the two print statements at line 15 refer to line 2 for x and line 13 for y. Next, we consider the call to `middle` which declares $y = x$ at line 4 and $x = 1$ at line 9. Thus, the two print statements refer to line 9 for x and line 4 for y. Finally, the call to `inner` declares $x = 1$ at line 7. The two print statements refer to line 4 for y and line 2 for x. Overall, the program will correctly compile and print output when run. This is because at line 4 the declaration of 4 using x refers to line 2 where x was declared. Similarly, for line 6 the the printing of variable x will refer to line 2 where the x was declared. Overall, we end up printing:

```
2
2
1
2
2
3
```

2 HW2 Problem 2

- We begin by reasoning about the program if it were using static scoping. In this case, the active binding of a name is determined by the synctactic structure of the program where the scope of the variable is the smallest block (or subtree) where the variable is declared. Thus, when the program is started `main()` is executed followed by `set_x(1)` and global x is 1 for the whole block. When `first(2)` is called we set $x = 2$ through `set_x(2)` meaning that x is now 2 for the whole block. The call to `print_x()` and its print statement inside will print 2 and then the next call to `print_x()` will also print 2. Next, `second()` is called and a local variable x is declared which is valid for that block and then `first(3)` is called. For `set_x(3)` in `first(3)`, the smallest block the variable is declared is the global context meaning the global variable x is now 3. Similarly, this holds for `print_x()` meaning the global variable $x = 3$ which gets us 3 printed. Finally, this also applies to the last `print_x()` and we again get 3 printed out. Summarizing the result, we end up printing:

```
2
2
3
3
```

- For dynamic scoping, the active binding of name is determined by its most recent declaration at run-time. Thus, when the program is started `main()` is executed and there is a global variable x that is unitialized. All references to x from this point on will refer to this x. Thus, when `set_x(1)` is called we set the global x to 1 and continue the program. Then there is a call to `first(2)` and the same global x is still valid. Thus, `set_x(2)` will set global x to 2 and then the call to `print_x()` will print 2. Similarly, the next call to `print_x()` will also print 2. Next, we call `second()` and introduce a local variable x. Because of dynamic scoping we now know that all references to x will be this local one until the `second()` method ends. Thus, `set_x` will set this local x to 3 via the call to `first(3)` and `print_x()` will print out this local x, meaning we get 3 printed out. Upon completion of `second()` we now lose the binding to the local x since the stack-allocation for `second()` is popped and now all references to x are to the global x. Thus, the last `print_x()` will print out 1 (since it was set to that using `first(2)`) and the program will end. Summarizing the result, we end up printing:

```
2
2
3
2
```

3 HW2 Problem 3

- We note that when we enter a sub-block we need to allocate some amount of space for the declared variables. Upon exiting that sub-block we can free up the allocated space since those variables are not accessible out of that block and thus are no longer used. Applying this reasoning to the given pseudocode, the top-level block declares two integers meaning we need $2 \times 4 = 8$ bytes, the next sub-block declares two integers meaning we need $2 \times 4 = 8$ bytes and then the third sub-bloc declares one more integer meaning we need $1 \times 4 = 4$ bytes. At this point, we need in total $8 + 8 + 4 = 20$ bytes. Now we can free 4 bytes once the third sub-bloc ends and then free 8 more bytes when the second sub-block ends. Next, we notice another new sub-block that declares three more integers which means we need $3 \times 4 = 12$ bytes. We realize that we had allocated 20 bytes and freed 12 bytes meaning we have 8 bytes allocated. Instead of freeing those 8 bytes we could just re-use the already available free space for the next 3 integers for this sub-block or free and allocate the space again. Either way, the total amount of space required for the variables when the program is executed is 20 bytes.

4 HW2 Problem 4

1. We reason about the memory representation of the program. The first `Node* n1` declaration creates a node with `data`$= 1$ and its `next` set to `NULL`. Then the second `Node* n2` declaration creates a node with `data`$= 2$ and its `next` set to `n1`. Then we have the call `foo(n2)` which deletes the `next` of `n2`

which is also pointed to by variable `n1`. At this point we have a dangling pointer and in C++ this is undefined behavior. The `delete` command in C++ indicates to the OS that this area of memory can be reclaimed as the program no longer needs it. Next, the declarations `Node* n3` creates a node with `data`= 42 and its `next` set to `NULL`. The OS has to allocate space for this new node. It could either (1) use the space freed by the deallocation from the call to `foo(n2)` which means that `n1` will now point to the space allocated for `n3` or (2) allocate new space for `n3` which means that `n1` will now point to space that is not allowed to be used by the program (as we told the OS that we do not need this space anymore). In the case of (1) we will end up printing the value of 42 since `n1->data = 42;` will access the memory allocated for `n3` and modify that space (which in turn means `n3->data`= 42). In the case of (2) we will end up trying to access memory that we no longer have access to because `n1->data` will point to a space that was freed meaning we may get a segmentation fault and the program crashes. However, as a note, since working with a dangling pointer is undefined behavior in the case of (2) the program may also not crash and keep running.

2. Now, again, the program may crash, print 3 or do something else. This is because `n1`, `n2` and `n3` will all point to distinct locations in memory since the declaration `Node* n3` happens first meaning the area allocated for `Node* n1` is still valid and the OS must give some new space. Then we will free the space `n1` is pointing to via the call `foo(n2)` which involves telling the OS we no longer need this space which in turn makes the space inaccessible to the program. Finally, `n1->data` will try to access a memory location no longer available to the program which may lead to a segmentation fault and the program will crash. If the program does not crash at this point and `n1->data` does not end up modifying the contnet of `n3` (because it may even though they are not aliased) then we will end up printing 3.

5 Class 4

5.1 Overview

- Subroutines provide the basic abstraction mechanism in programs.
- Functions correspond to the mathematical notion of computation, i.e. they are viewed as mapping from input to output values. They can be viewed as abstractions of side-effect free expressions. Procedures can be viewed as abstractions over statements. That is, they affect the environment (mutable variables, hard disk, network, ...), and are called for their side-effects.

5.2 Stack frames / activation records

- Each time a subroutine is called we need to allocate space on the stack for the objects needed by the subroutine. We call this space a stack frame or an activation record.
- Stack pointer contains the address to the last used location or next unused location on the stack.
- Frame pointer contains points to the activiation record of a subroutine so that any object on the stack can be referenced using a stack offset from this pointer.
- Q: Why not use an offset from the stack pointer to reference subroutine objects? A: There may be objects that are allocated on the stack whose size is unknown at compile time.
- Prologue and epilogues + calling sequence (see class04 notes)
- Callee vs. caller performed tasks. If possible, have the callee perform tasks: task code needs to occur only once, rather than at every call site. However, some tasks such as parameter passing must be done by the caller.

- What to do about registers? Who should save what? Difficult question in general. Most arhcitectures will do half and half (i.e. half saved by caller and half saved by callee). Alternative idea is to use register windows where each subroutine has access to only a small window of a large number of registers and they are only responsible for this poriton. Overlapping windows is equivalent to parameter passing. When anohter subroutine is called the window will need to move.
- Optimizations: Leaf routines = routine that does not call any more subroutines meaning we do not need to push the return address on the stack (and leave it in the register instead). Inlining = inserting the code for the function at every call site. Positives include less overhead and more compiler optimization. Negatives include the increase in the code size and recursive procedures cannot be inlined.

5.3 Evaluation Strategy and Parameter Passing Modes

- Formal parameters: these are the names that appear in the declaration of the subroutine.
- Actual parameters or arguments: these refer to the expressions passed to a subroutine at a particular call site.

```
// formal paramters: base, pow
def exp(base: Int, pow: Int): Int = { ... }

// actual parameters: f(i,j), i*5
exp(f(i,j), i*5)
```

- What does a reference to a formal parameter in the subroutine mean in terms of the actual parameters? We need to think about the evaluation strategy.
- Strict evaluation: the actual is evaluated before the call to the function.
- Lazy evaluation: the actual is evaluated only if and when its value is needed during execution of the call.
- Strict:
  - by value: formal is bound to a copy of the value of actual, if assignment to formal is allowed then the value at the copy is changed and not at the actual
  - by reference: formal is bound to location of actual (aliasing), if assignment to formal is allowed then the value at the actual is also changed, actual must be an l-value
  - copy-return: formal is bound to copy of value of actual, upon return from the routine the actual equals the copy of the formal
- Lazy:
  - by name: formal is bound to expression for actual, the expression is (re)evaluated each time the formal is used in the callee, is equivlent to textual subsitution of occurence of formal to the expression of actual, no assignment to formal allowed (why?)
  - by need: formal is bound to expression for actual, the expression is evaluated the first time the formal is used in the calle, subsequent uses use the computed value from earlier, no assignment to formal allowed (why?)
- Exercise:

```
def foo(a: Int, b: Int) = {
  a = b * (b + 1);
  println(a);
  println(b);
  a = a + 1;
}
var x = 3;
var y = 2;
foo(x, x*y);
println(x);
println(y);
```

What happens for the following cases:

- a is call by value, b is call by value
- a is call by reference, b is call by value
- a is call by value, b is call by name
- a is call by reference, b is call by name

Answers:

- 42, 6, 3, 2
- 42, 6, 43, 2
- 42, 6, 3, 2
- 42, 84, 43, 2

- What is the advantages/disadvantages of used passing by need? Advantage: we only evaluate an expression if we use it during a call. Disadvantage: the implementation is more complex and behavior can be confusing if we have side effects.

5.4 Passing Subroutines as Parameters

- What should we do if there are nested subroutines which are passed as a parameter? What should the program below print?

```
def a(i: Int, p: () => Unit): Unit = {
  def b(): Unit = println(i)

  if (i > 1) p()
  else a(2, b)
}

def c(): Unit = ()

a(1, c)
```

- We have two possible options.
  - Deep binding: We create a close and pass it in place of the subroutine. The closure references the subroutine along with its referencing environment. When the subroutine is called the referencing enviornment from when the closure was created neds to be restored. So for the above program we end up printing 1 because i was bound to 1 when the closure for b was created.

  - Shallow binding: When a subroutine is called, it uses the curretn referencing enviornment at the call site. So for the above program we end up printing 2 because i was bound to 2 when b was called.

- Exercise:

```
var x: Int = 0;
def printer(y: Int): Unit = {
  println(x);
  println(y);
}
def foo(f: Int => Unit, y: Int): Unit = {
  var x: Int = 1;
  f(x);
}
foo(printer, 2)
```

  What happens for the following cases:

  - we have static scoping semantics for names and deep binding semantics for functions that are passed as arguments to other functions
  - we have dynamic scoping for names and shallow binding semantics for functions passed as arguments

  Answers:

  - 0, 1
  - 1, 1

- Static scoping demands the use of deep binding for nested subroutines. Shallow binding is typically the default in languaes with dynamic scoping.