CSC 347 - Concepts of Programming Languages

Exercises

Instructor: Stefan Mitsch

Exercise Topics

- Tail recursion
- Scope and lifetime
- Algebraic datatypes
- Argument passing
- Closures
- Inheritance and delegation
- JavaScript

Tail Recursion

```
1 def f (n:Int) = if n<=1 then 1 else n * f (n-1)
2 def g (n:Int) = f (n) * g (n-1)
3 def h (n:Int) = h (g (n), f (n))
4 def i (n:Int) = n * n
```

- Is f recursive? tail-recursive?
- Is g recursive? tail-recursive?
- Is h recursive? tail-recursive?
- Is i recursive? tail-recursive?

Scope and Lifetime: Dangling Pointers

```c
int* f(int x) { return &x; }
int* g(int x) {
  int* y = (int*) malloc (sizeof (int));
  *y=x;
  return y;
}
int main(void) {
  int* p = f(5);
  printf("%d\n", *p);
  int* q = g(5);
}
```

- Is `p` pointing to the heap or stack?
- Is `p` dangling? Is dereferencing it safe?
- Does `p` need to be freed?

- Is `q` pointing to the heap or stack?
- Is `q` dangling? Is dereferencing it safe?
- Does `q` need to be freed?

Static vs. Dynamic Scope

❓ What is printed in a statically-scoped vs. dynamically-scoped language?

```
1 var x:Int = 10
2 def foo() =
3   x = 20
4 def bar() =
5   var x:Int = 30
6   foo()
7   println(x)
8 def zee() =
9   var x:Int = 40
10  bar()
11  println(x)
12 print("bar: "); bar(); print(" " + x)
13 print("zee: "); zee(); print(" " + x)
14 print("foo: "); foo(); print(" " + x)
```

- Static Scope

```
1 bar: 30 20
2 zee: 30 40 20
3 foo: 20
```

- Dynamic Scope

```
1 bar: 20 10
2 zee: 20 40 10
3 foo: 20
```

Algebraic Datatypes

❓ Define an algebraic datatype for arithmetic expressions with literals, variables, and operators `+`, `-`, `*`

- In Scala

```scala
enum Expr:
  case Literal(x:Int)
  case Variable(n:String, v: Option[Expr])
  case Neg(e:Expr)
  case Plus(l:Expr, r:Expr)
  case Minus(l:Expr, r:Expr)
  case Times(l:Expr, r:Expr)
```

Algebraic Datatypes

❓ Define a `toInt` method for arithmetic expressions

```
1 enum Expr:
2   case Literal(x:Int)
3   case Variable(n:String, v:Option[Expr])
4   case Neg(e:Expr)
5   case Plus(l:Expr, r:Expr)
6   case Minus(l:Expr, r:Expr)
7   case Times(l:Expr, r:Expr)
```

- Method `toInt`

```
1 def toInt : Int = this match
2   case Literal(x)           => x
3   case Variable(n, None)    => ???
4   case Variable(_, Some(e)) => e.toInt
5   case Neg(e)               => -e.toInt
6   case Plus(l,r)            => l.toInt + r.toInt
7   case Minus(l,r)           => l.toInt - r.toInt
8   case Times(l,r)           => l.toInt * r.toInt
```

Call-by-value and Call-by-reference

❓ What is printed in a call-by-value vs. a call-by-reference language?

```
1 def f(a:Int, b:Int) = {
2   a = b
3   b = b + 1
4 }
5
6 var x = 5
7 var y = 10
8 f(x,y); println(s"x=$x, y=$y")
9 f(x,x); println(s"x=$x")
10 f(x,x+2); println(s"x=$x")
```

- Call-by-value

```
1 x=5, y=10
2 x=5
3 x=5
```

- Call-by-reference

```
1 x=10, y=11
2 x=11
3 x=13
```

Closures

```
1 def f(): String=>String = {
2   var s = ""
3   (t:String) => { s = s + t; s }
4 }
5 val a = f()
6 println(a("x"))
7 println(a("y"))
8 println(a("z"))
```

- Does the program compile?
- What is the type of a?
- What is printed?
- What is an OOP approach to the code above?

- Type of `a` is function `String=>String`
- Prints

```
1 x
2 xy
3 xyz
```

- Implement with a class

```
1 public class f {
2   private String s = "";
3   public String fn(String t) {
4     s = s+t;
5     return s;
6   }
7 }
8 f a = new f()
9 System.out.println(a.fn("x"));
```

Nested Closures

```
1 def f(): ()=>String=>String = {
2   var s = ""
3   () => {
4     var i = 0
5     (t:String) => { s = s + t + i; i = i+1; s }
6   }
7
8 }
9 val factory = f()
10 val a = factory()
11 val b = factory()
12 println(a("x"))
13 println(a("y"))
14 println(b("z"))
```

- Do `a` and `b` share `s`?
- Do `a` and `b` share `i`?
- What is printed?

Inheritance

```
 1 class A {
 2   def f() = { println("A.f") }
 3 }
 4 class B extends A {
 5   override def f() = { println("B.f"); super.f() }
 6            def g() = { println("B.g") }
 7 }
 8
 9 val b:A = new B() // b:B?
10 b.f()
```

- Does the program compile? If yes, what is printed?
- Does `b.g()` compile? If yes, what is printed?
- If no, would it compile with `val b:B = new B()`?

Parametric Types

```
1 class B extends A {}
2 var as: List[A] = List(new A(), new B())
```

Which code compiles and why/why not?

- ? `val bs: List[B] = as`
- ? `val bs: List[B] = List(new B(), new B())`
- ? `as = bs`

  `val a1: Any = as(0) // a1: A // a1: B`

JavaScript

```
1 function f() {
2   return {
3     a: 1,
4     b: "hello"
5     c: function () { return 2; }
6   }
7 }
```

- What is the result type of function f?
- What is the result of `f().a`?
- What is the result of `f().d`?
- What is the result of `f().c()`?

JavaScript: Hoisting

```
1 function f(n) {
2   var fs = [];
3   for (var i = 0; i < n; i++) {
4     var x = i;
5     fs.push(x);
6   }
7   return fs;
8 }
```

- What is the result of `f(3)` and why?
- How can we make the result of `f(3)` be `[0,1,2]`?

JavaScript: Closures

```
1 function f(n) {
2   var x = n;
3   return {
4     get: function(y) { x = x+y; return x; }
5   }
6 }
```

- What is the result of `f(3).get(1)` and why?
- What are the values of `b` and `c` in

```
1 var a = f(3);
2 var b = a.get(1);
3 var c = a.get(2);
```

Scala Lists

```scala
val xs = 1 :: 2 :: Nil
val ys = List(1, 2)
val zs = 0 :: xs
val as = List(0) :: xs
val bs = List(0) ::: xs
```

- Do `xs` and `ys` contain the same elements?
- What is `zs`? Does it contain the same elements as `0 :: ys`?
- What is `as`?
- What is `bs`?

Missing Types

```
1 def f(xs:List[?3]): ?4 = xs.map(x => x+1)
2
3 def g(ys:?2): ?1 = ys.foldLeft(0)(_ + _)
4
5 g (f (List (1,2,3,4)))
```

- In which order should we try to resolve the types?
- What is the type `?1`
- What is the type `?2`
- What is the type `?3`
- What is the type `?4`