CSC 347 - Concepts of Programming Languages

Exercises

Instructor: James Riely

Exercise Topics

- Tail recursion
- Safety, scope and lifetime
- Algebraic datatypes
- Argument passing
- Closures
- Inheritance and delegation
- JavaScript objects and scope

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?

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
```

- `f` is recursive, but not tail-recursive (multiplication `n*f(n-1)`)
- `g` is recursive, but not tail-recursive (multiplication `f(n) * g(n-1)`)
- `h` is tail-recursive (recursive call to `h` last)
- `i` is not recursive at all

Continuation Passing

```
1 def fact(n: Int) : BigInt =
2   require (n>=0)
3   if n <= 1 then 1
4   else n * fact(n-1)
5 end fact
```

```
1 def factCPS[X](n: Int, k: _____________) : X =
2   require (n>=0)
3   if n <= 1 then _________________
4   else factCPS(_______, _________)
5 end factCPS
```

- Fill in the blanks in the continuation passing style

Continuation Passing

```
1 def fact(n: Int) : BigInt =
2   require (n>=0)
3   if n <= 1 then 1
4   else n * fact(n-1)
5 end fact
```

```
1 def factCPS[X](n: Int, k: BigInt=>X) : X =
2   require (n>=0)
3   if n <= 1 then k(1)
4   else factCPS(n-1, r => k(n*r))
5 end factCPS
```

Scope and Lifetime: Dangling Pointers

```
1  int* f(int x) { return &x; }
2  int* g(int x) {
3    int* y = (int*) malloc (sizeof (int));
4    *y=x;
5    return y;
6  }
7  int main(void) {
8    int* p = f(5);
9    printf("%d\n", *p);
10   int* q = g(5);
11 }
```

- 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?

Scope and Lifetime: Dangling Pointers

```
1  int* f(int x) { return &x; }
2  int* g(int x) {
3    int* y = (int*) malloc (sizeof (int));
4    *y=x;
5    return y;
6  }
7  int main(void) {
8    int* p = f(5);
9    printf("%d\n", *p);
10   int* q = g(5);
11 }
```

- `p` points to the stack
- `p` is dangling, and dereferencing it is not safe
- `p` points already to freed memory
- `q` points to the heap
- `q` is not dangling, it can be dereferenced safety
- `q` needs to be freed

Unsafe Memory Access

```
 1 int* f(int* x) {
 2   *x = *x+1;
 3   return x;
 4 }
 5 int main(void) {
 6   int x = 5;
 7   double y = 5.2;
 8   printf("%d\n", *f(&x));
 9   printf("%f\n", *f(&y));
10 }
```

- Does the program make an unsafe memory access?
- If so, where?

Unsafe Memory Access

```c
int* f(int* x) {
  *x = *x+1;
  return x;
}
int main(void) {
  int x = 5;
  double y = 5.2;
  printf("%d\n", *f(&x));
  printf("%f\n", *f(&y));
}
```

- Program makes an unsafe memory access when dereferencing `x`, because it accesses memory that stores a `double` as if it were an `int`

Option Types

```
1 case class Person(name: String, email: Option[String])
2
3 def invite(p: Person, mailSender: String=>Boolean) : Boolean = p.email match
4    case __________ => ___________________
5    case __________ => ___________________
6 end invite
7
8 def email(p: Person) : String = ____________________________________
```

- Send an email to person `p` using the `mailSender`, depending on whether `p` has an email address
- Return the email address of person `p` as a string, or `"unknown"` if they don't have an address

Option Types

```scala
case class Person(name: String, email: Option[String])

def invite(p: Person, mailSender: String=>Boolean) : Boolean = p.email match
   case None    => false
   case Some(a) => mailSender(a)
end invite

def invite2(p: Person, mailSender: String=>Boolean) : Boolean =
  p.email.map(mailSender).getOrElse(false)
end invite2

def email(p: Person) : String = p.email.getOrElse("unknown")
```

- Send an email to person `p` using the `mailSender`, depending on whether `p` has an email address
- Return the email address of person `p` as a string, or `"unknown"` if they don't have an address

Static vs. Dynamic Scope

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

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

Static vs. Dynamic Scope

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

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

```
1 enum Expr:
2   case Literal(x:Int)
3   case Variable(n:String, v: Option[Expr])
4   // TODO
```

❓ Extend the algebraic datatype for arithmetic expressions with operators `+`, `-`, `*`

❓ Are there more than one `-` operator?

Algebraic Datatypes

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

❓ Extend the algebraic datatype for arithmetic expressions with operators `+`, `-`, `*`

❓ Are there more than one `-` operator?

Algebraic Datatypes

❓ Finish the `toInt` method for arithmetic expressions

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

- Method `toInt`

```
1 def toInt(e: Expr) : Int = e match
2   case Literal(x)           => x
3   case Variable(n, None)    => ???
4   case Variable(_, Some(e)) => toInt(e)
5   // TODO
```

Algebraic Datatypes

❓ Finish the `toInt` method for arithmetic expressions

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

- Method `toInt`

```
1 def toInt(e: Expr) : Int = e match
2   case Literal(x)           => x
3   case Variable(n, None)    => ???
4   case Variable(_, Some(e)) => toInt(e)
5   case Plus(l, r)           => toInt(l) + toInt(r)
6   case Minus(l, r)          => toInt(l) - toInt(r)
7   case Neg(e)               => 0-toInt(e)
8   case Times(l, r)          => toInt(l) * toInt(r)
```

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 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
```

Fold

```
1 val l = List(1,2,3,4,5)
2
3 val sum = l.foldLeft(___) {
4   ____________________
5 }
6
7 val count = l.foldLeft(____) {
8   ____________________
9 }
10
11 val countEven = l.foldLeft(____) {
12   ____________________
13 }
```

- Implement functions using fold

Fold

```
1 val l = List(1,2,3,4,5)
2
3 val sum = l.foldLeft(0) {
4   (s, n) => s+n // _ + _
5 }
6
7 val count = l.foldLeft(0) {
8   (c, _) => c+1
9 }
10
11 val countEven = l.fold(0) {
12   (c, n) => if n%2 == 0 then c+1 else c
13 }
```

- Implement functions using fold

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?

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?

- `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 end f
8 val factory = f()
9 val a = factory()
10 val b = factory()
11 println(a("x"))
12 println(a("y"))
13 println(b("z"))
```

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

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 end f
8 val factory = f()
9 val a = factory()
10 val b = factory()
11 println(a("x"))
12 println(a("y"))
13 println(b("z"))
```

- `a` and `b` share `s`, but do not share `i` (each has their own `i`)
- `a` and `b` append to the same `s`, but increment their own `i`, so prints `x0`, `x0y1`, and `x0y1z0`

Inheritance vs. Delegation

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

```
1 let a = {
2   f: function() {
3     console.log("A.f"); }
4 }
5 let b = {
6   f: function() {
7     console.log("B.f");
8     this.__proto__.f(); },
9   g: function() {
10    console.log("B.g"); }
11 }
12 b.__proto__ = a
13 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()`?

Inheritance vs. Delegation

```
class A:
  def f() = println("A.f")
class B extends A:
  override def f() =
    println("B.f")
    super.f()
  end f
  def g() = println("B.g")

val b:A = new B() // b:B?
b.f()
```

```
let a = {
  f: function() {
    console.log("A.f"); }
}
let b = {
  f: function() {
    console.log("B.f");
    this.__proto__.f(); },
  g: function() {
    console.log("B.g"); }
}
b.__proto__ = a
b.f();
```

- The program compiles and prints `B.f A.f`
- `b.g()` does not compile, because variable `b` is of static type `A`
- The program would compile with `val b:B = new B()` and would print `B.g`
- For JavaScript: program executes and also prints `B.f A.f`

Parametric Types

```
1 class A
2 class B extends A
3 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` (with `bs: List[B]`)
- ? `val a1: Any = as(0) // a1: A // a1: B`

Parametric Types

```
1 class A
2 class B extends A
3 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` (Scala: `List` is covariant, `List[B] <: List[A]` from `B <: A`)
- ✓ `val a1: Any = as(0)`, ✓ `val a1: A = as(0)`, ⚡ `val a1: B = as(0)`

Parametric Wildcard Types

```
1 class A
2 class B extends A
3 val as: Array[A] = Array(new A(), new B())
```

Which code compiles and why/why not?

- ❓ `val bs: Array[B] = as` ?

❓ With subtype wildcard

- `val bs: Array[? <: A] = as` ?
- `bs(0) = new B()` ?
- `val b: B = bs(0)` ?
- `val b: A = bs(0)` ?
- `val b: Any = bs(0)` ?

❓ With supertype wildcard

- `val bs: Array[? >: B] = as` ?
- `bs(0) = new B()` ?
- `bs(0) = new A()` ?
- `val b: B = bs(0)` ?
- `val b: Any = bs(0)` ?

Parametric Wildcard Types

```
1 class A
2 class B extends A
3 val as: Array[A] = Array(new A(), new B())
```

Which code compiles and why/why not?

- ⚡ `val bs: Array[B] = as` (arrays are type-invariant)

❓ With subtype wildcard

- ✓ `val bs: Array[? <: A] = as` ?
- ⚡ `bs(0) = new B()` (subtype wildcard makes array read-only)
- ⚡ `val b: B = bs(0)` (parameter type of array is `A`)
- ✓ `val b: A = bs(0)` (parameter type of array is `A`)
- ✓ `val b: Any = bs(0)` (`Any` is a supertype of `A`)

❓ With supertype wildcard

- ✓ `val bs: Array[? >: B] = as`
- ✓ `bs(0) = new B()` (supertype wildcard makes array "write-only")
- ⚡ `bs(0) = new A()` (`B` could have other supertypes)
- ⚡ `val b: B = bs(0)` (some supertype of `B` may have additional subtypes)
- ✓ `val b: Any = bs(0)` (degraded read to `Any` is allowed)

JavaScript Objects

```
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 Objects

```
function f() {
  return {
    a: 1,
    b: "hello"
    c: function () { return 2; }
  }
}
```

- Function `f` returns an object
- `f().a` returns `1`
- `f().d` returns `undefined` (because property `d` is not present in the object)
- `f().c()` returns `2`

JavaScript: Function-Scoped Variables

```
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: Function-Scoped Variables

```
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 }
```

- `f(3)` returns `[()=>2, ()=>2, ()=>2]` because `var` makes `x` function-scoped and `var x` is therefore hoisted outside the loop to the start of function `f`
- `f(3)` becomes `[()=>0,()=>1,()=>2]` when we use `const` or `let` (block-scope)