HT2026 Imperative Programming notes

Remaining TODOs: 2

Contents

1. Introduction to scala

This course uses scala v2.

Functions declared with `def func(...): T = { ... }`. Argument types are required, and return types are required when the function is recursive.

The last expression in a function body is returned; but `return` can be used for explicit or early returns.

Example:

```scala
def fact(n: Int) : BigInt = {
  if (n == 0) 1
  else fact(n - 1) * n
}
```

Semicolons are usually optional at the end of statements.

If/else require brackets `()` around the condition, and optional curly braces around the body. This leads to if/else if/else naturally looking like `if (...) { ... } else if (...) { ... } else { ... }`.

While loops are similar to the above. For loops are written as `for (x <- sequence) { ... }`. The sequence can be something like `1 to 4` (inclusive range) or `1 until 4` (exclusive range).

Types are PascalCase, e.g. `Boolean`, `Int` (32-bit), `BigInt` (arbitrary precision). Other numerical types are standard (long, float, double, char).

Argument preconditions can be expressed using the `require` function. This should only be used for argument preconditions, not for internal assertions. Note that this is a runtime-only check.

Comments are C-style, using `//` and `/* */`.

We can define objects, typically in a file of the same name.

When an object has a `main` method, that method is the entrypoint. As an example:

```scala
object Factorial {
  def fact(n: Int) : BigInt = {
    require (n >= 0)
    if (n==0) 1
    else n * fact(n - 1)
  }

  def main(args: Array[String]) : Unit = {
    print("Please input a number: ")
    val n = scala.io.StdIn.readInt()
    if (n >= 0) {
      val f = fact(n)
      println("The factorial of " + n + " is " + f)
    } else {
      println("Negative numbers are not allowed")
    }
  }
}
```

Listing 1: Factorial.scala

Each main object should have a function with signature `main (args: Array[String]) : Unit`.

Immutable variables are declared using `val` mutable variables are declared using `var`. Types for values and variables are optional.

2. Invariants

Consider a looping variant of `fact`:

```
object FactorialLoop {
  def fact(n: Int) : BigInt = {
    require (n >= 0)
    var f: BigInt = 1
    var i = 0
    while (i < n) {
      i += 1
      f *= i
    }
    f
  }

  def main(args: Array[String]) : Unit = {
    print("Enter a number: ")
    val n = scala.io.StdIn.readInt()
    val f = fact(n)
    println("n! = " + f)
  }
}
```

To prove the correctness, we want to prove an invariant: at the beginning and end of each iteration, `f = i!` and `i <= n`.

Initially, `i = 0, f = 1 = 0!`. Assuming that the invariant holds, at the end of the loop, `i = i + 1, f = f * (i+1) = i! * (i+1) = (i+1)!`. Moreover, since `i<n` at the beginning of the loop, `i<=n` at the end of the loop. When the loop terminates, `f = i! && i <= n && not (i < n)`. Therefore `i = n` so `f = n!`. Hence the program is correct (assuming termination).

Definition 2.1: An invariant is a property that holds at the beginning and end of every iteration of a loop.

For a program with an invariant `I`,

```
// pre
Init
// I
while (test) {
  // I and test
  Body
  // I
}
// I && not test
// post
```

we need to check:

- `Init` establishes `I`, assuming `pre`;
- `Body` maintains `I`;
- the loop terminates;

- `I && not test` implies `post`.

In scala, `Array`s have a fixed size. They are 0-indexed using parentheses; the `i`th element of `a: Array[T]` is `a(i)`. Arrays are mutable, i.e. `a(i) = a(i) + 1` is valid.

When an object has a nullary operator it can be called as `a.op()` or just `a.op`. If it is not nullary (i.e. arity >= 1), it must be called with parentheses, e.g. `a.op(x)`.

Example (Summing elements of an array): Invariant `I` is `s = sum a[0..i) && 0 <= i <= a.size`

```
def sum(a: Array[Int]) : Int = {
  // pre: true
  var s = 0
  var i = 0
  // I
  while (i < a.size) {
    // I && i < a.size
    s += a(i)
    i += 1
    // I
  }
  // I && not (i < a.size)
  // s = sum a[0..a.size)
  s
}
```

`I` holds after initialisation because `0` is equal to the sum over the empty range.

Assuming `s = sum a[0..i)` at the beginning of an iteration, at the end `i = i + 1` and `s = s + a(i) = sum a[0..i) + a(i) = sum a[0..i+1)` hence `I` is maintained.

At termination, `I && not (i < a.size)` holds so `s = sum a[0..i) && 0 <= i <= a.size && not (i < a.size)`. Therefore, `i = a.size` so `s = sum a[0..a.size)` (assuming termination).

Definition 2.2: A variant is an expression `v` that typically measures how much more work needs to be done.

To prove termination, establish a variant `v` such that:
- `v` is integer-valued, assuming the invariant;
- `v >= 0`, assuming the invariant;
- and `v` is decreased at each iteration.

If these conditions are met, the loop must terminate as it can only be executed a finite number of times.

Definition 2.3: A Hoare triple is $\{P\}$ `Prog` $\{Q\}$, meaning that if `prog` is executed from a state that satisfies P, it terminates satisfying Q. P is the precondition and Q is the postcondition.

Theorem 2.4: Given conditions `pre` and `test`, programs `Init` and `Body`, an invariant I and variant v, then if:

- $\{\text{pre}\}\ \text{Init}\ \{I\}$;
- $\{I \land \text{test}\}\ \text{Body}\ \{I\}$;
- $I \land \neg\ \text{test} \Longrightarrow \text{post}$;
- $I \Longrightarrow v \in \mathbb{N}$;
- and $\{I \land \text{test} \land v = V\}\ \text{Body}\ \{v < V\}$ where V is a logical constant,

then

$$\{\text{pre}\}\ \text{Init}; \text{while}\ (\text{test})\ \text{Body}\ \{\text{post}\}.$$

Example: For the summing example, let `v = a.size - i`. TODO

Remark: Recursion and iteration are similar in many ways. Proving recursion is correct is done by induction; proving iteration is correct is proved with invariants. We need to show that recursion depth is bounded and that loops terminate.

Remark: `for (i <- a until b) Body` is equivalent to `var i = a; while (i < b) { Body; i += 1 }`, so we can reason about `for` loops using the same techniques as for `while` loops.

Before `var i = a` is executed, we use the invariant `I[a/i]`, meaning "`I` with any mention of `i` replaced with `a`", or `I` with `a` substituted for `i`.

So:

```
Init             // I[a/i]
var i = a        // I
while (i < b) { // I && a <= i <= b
  Body           // I[i+1/i]
  i += 1         // I
}                // I[b/i]
```

and

```
Init                    // I[a/i]
for (i <- a until b) { // I && a<=i<b
  Body                  // I[i+1/i]
}                       // I[b/i]
```

Anonymous function literals (for unary functions only) in scala can be written using an underscore where the argument would be used. E.g. given a list `a` we can write `args.map(_.toInt)` which is equivalent to `args.map(x => x.toInt)`. Types can be provided for anonymous functions using e.g. `(_: String).toInt` or `(x: String) => x.toInt`.

3. Testing

We can classify errors into 3 levels in a hierarchy:

- Level 1: the error can be fixed safely. Fix it; if needed, warn
- Level 2: the error could be caused by user input. Throw an exception up to calling code, which should have enough context to fix the problem
- Level 3: the error should not happen under normal circumstances; trip an assertion (i.e. crash the program)

Exception handling in scala uses `try`/`catch` and `throw`. `catch` is similar to a `match` expression; we match the type of the exception with `case identifier: AssertionType`.

`require` throws an `IllegalArgumentException`; `assert` throws an `AssertionError` and is intended to be more serious.

An example of handling a fallible function that might use `require` or `assert`:

```scala
def foo(): Unit = {
  try {
    bar(1)
  } catch {
    case iae: IllegalArgumentException => {
      println("illegal argument :(")
      // some sort of sensible fallback
    }
    case ae: AssertionError => {
      println("assertion failed!")
      // do something to stop execution
    }
  }
}
```

Assertions can be useful as sanity checks for checking invariants.

Definition 3.1: Unit testing is testing performed on an individual unit of a program, such as a function. Unit tests are nice because tests are independent of each other and, as much as possible, independent from the implementation.

ScalaTest is a testing framework. Example usage:

```scala
import org.scalatest.funsuite.AnyFunSuite

object Add {
  def add(a: Int, b: Int): Int = {
    a + b
  }
}

class TestAdd extends AnyFunSuite {
  test("0+5 = t"){ assert(Add.add(0,5) === 5) }
}
```

When running tests, we can use === rather than == to get better error messages. This is defined specifically for ScalaTest.

To check that a function throws a particular exception, use `intercept`: `intercept[IllegalArgumentException]{ foo(1) }`.

To test floats, which are imprecise, we can use e.g. `assert(a === b +- 1e-25)`.

Definition 3.2: Black-box unit testing treats a component as a "black box":
- Influenced only by knowledge of the component specification
- No knowledge of internal organisation, to avoid testing by rewriting the implementation.

With this approach, tests can be written before development even starts.

Definition 3.3: White-box unit testing has knowledge of the implementation of components, possibly with privileged access to private data:
- influenced by knowledge of component's internals
- can test that invariants are not violated.

This allows better code/path coverage, and allows us to start testing before a specification exists.

Definition 3.4: Equivalence class testing divides all possible inputs into equivalence classes in which the behaviour of the component should be the same. We test one input from each equivalence class assuming that two inputs from the same class behave in the same way.

Definition 3.5: Weak equivalence class testing tests one representative from each equivalence class for each input, but with arbitrary values for the other inputs at any one time.

Definition 3.6: Strong equivalence class testing tests one representative from each member of the Cartesian product of the equivalence classes of the inputs. This requires more tests but has better coverage for when the inputs might interact.

Remark: Equivalence class testing can take into account or ignore internal boundaries where equivalence classes are further divided by the implementation.

Definition 3.7: Boundary-value testing tests values near the boundaries of equivalence classes as that is where things are most likely to break. This works well with things like integers, but not with general equivalence tests, e.g. browser environment has no sensible boundary between firefox and chrome.

Some useful IO methods:
- `scala.io.Source.fromFile(fileName)` - returns a file object
- `file.getLines()` - returns a sequence of lines

4. Modularisation and Object Orientation

In scala, the identifier `this` refers to the current object. `this.op(args)` can be abbreviated to `op(args)`.

`eq` tests whether two objects are the same object - i.e. they refer to the same thing. Note that `a == b` does not imply that `a eq b`!

We can declare a class with private and static data as so:

```
class Point(x: Int, y: Int)
```

However, if we would like to have public static data, use `class Point(val x: Int, val y: Int)`; these can be mutable with `Point(var x: Int, var y: Int)`. We can have private mutable fields with `class Point(private var x: Int, private var y: Int)`.

We can then initialise `Points` with `val point = new Point(0, 0)`, and if we specified `x` and `y` as non-private, access them using `point.x` and `point.y`.

Also note that `equals` is an alias for `==`. The default implementation for `equals` is `eq`.

To override a default definition, use the `override` keyword:

```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any): Boolean = other match {
    case p: Point => this.x == y.x && this.y == p.y
    case _        => false
  }
```

Internal data and methods can be marked as such with the `private` keyword.

Definition 4.1: A singleton object are created without a corresponding class and used to hold utility methods or values that are global to the application; they are defined with the `object` keyword.

Definition 4.2: Standard objects are created from a corresponding class and used to represent instances of that class; they are defined using the `class` keyword and created using the `new` keyword followed by the class name.

We can create subclasses using the `extends` keyword; subclasses with inherit all data and methods, but can also add more, or `override` some behaviours.

We can access superclass methods via the `super` identifier; this acts like `this` in that we don't need to give it the current object.

Definition 4.3: When a function in a superclass calls a method that is overridden in a subclass, the method from the subclass is called; this is called dynamic binding.

Remark: Class inheritance can be dangerous, as we sometimes rely on the definitions of superclass methods calling methods of the current class; this assumption could be broken if we aren't careful. An alternative to inheritance (an "is-a" relationship) is composition, a "has-a" relationship, whereby we simply add fields to the class so that we can add their methods.

Definition 4.4: An abstract datatype provides a state, an initial representation of the state, and pre-/post-conditions on operations.

Definition 4.5: A trait gives an interface of what a module should do, but is not an implementation of these promises. A class can `extend` a trait to provide a concrete implementation of it.

Remark: Traits can provide default implementations for methods (concrete methods), that can be overridden. Compare this to traditional interfaces, which only provide abstract methods (just the type signature).

Remark: Traits cannot have instance variables, so cannot maintain state. Concrete methods in traits can access state, but the state must be instantiated in a concrete implementation.

Remark: Classes can implement multiple traits. Note that trait extension is not commutative; if multiple traits have a method with the same name, the trait that was "mixed in" last takes precedence.

Remark: Scala has a built in mutable set trait, `scala.collection.mutable.Set`, instantiated using `Set[T](...els)`. There are different concrete implementations; one is `scala.collection.mutable.HashSet[T]`. This is initialised using `new HashSet[T]`.

Remark: When specifying postconditions for methods involving state S, we refer to the state at the start of the operation as S_0.

Remark: `for` loops can include a predicate to skip certain iterations:

```
for (el <- container; if predicate(el)) {
  ???
}
```

Remark: Scala has a map trait, `scala.collection.mutable.Map[A, B]`, corresponding to a finite partial map from $A \mapsto B$. Useful methods are:

- `m.apply(key: A) -> B` (or just `m(key: A) -> B`)
- `m.update(key: A, val: B): Unit` (or `m += ((k, v))` or `m += k -> v`)
- `m.contains(key: A) -> Boolean`

Definition 4.6: A datatype invariant (DTI) is a property that is true initially nd is maintained by each operation, so is true after each operation.

Definition 4.7: An abstraction function is a function abs that takes the concrete state c (i.e. the actual scala value) and returns the corresponding abstract state a (the mathematical representation of the state). We write $a = \text{abs}(c)$.

Any operation should change c such that $a = \text{abs}(c)$ is changed in a way allowed by the specification.

Remark: We need to check that:

- the abstraction function produces a result of the type of the abstract state (e.g. if it should return a function, the function should be well-defined, for example from the DTI);
- and the initial concrete state agrees with the initial abstract state, and satisfies the DTI.

Theorem 4.8: In general, to prove that a concrete implementation meets a specification over the abstract state space, we need to prove that:

- c_0 is an initial concrete state that satisfies the DTI,
- and $a_0 = \text{abs}(c_0)$ satisfies the abstract precondition

implies that

- c satisfies the DTI,
- and the abstract postcondition is satisfied by $(a_0, a := \text{abs}(c), \text{abs}(\text{res}))$ where res is the result of the operation.

That is, if we have an abstract specification with state a

$$\text{pre}(a) \implies \text{returns res s.t. post}(a_0, a, \text{res}),$$

then the concrete implementation with state c should satisfy that

$$[\text{pre}(\text{abs}(c_0)) \land \text{DTI}] \Longrightarrow [\text{returns res s.t. post}(\text{abs}(c_0), \text{abs}(c), \text{res}) \land \text{DTI}].$$

Remark: The user of the function is responsible for enforcing the precondition. If the precondition is not satisfied, we don't specify the behaviour and anything is reasonable.

Definition 4.9: A case class is like a normal class, declared with `case class`, with all the normal features plus:

- you can pattern match/destructure on it (this is by a provided `unapply` method, which you can also implement yourself, returning an `Option` of a tuple - or just a `Boolean` if there are no variables to bind)
- constructor parameters are `public val` by default
- you don't need to use `new`
- default `copy` (clones, possibly with changed parameters/fields as specified, e.g. `val p = Point(1, 3); val q = p.copy(x = 3);`), `equals`, `hashCode` and `toString` methods are generated with more expected behaviour than the default implementations for normal classes

These are useful for FP-esque things. But especially pattern-matching!

Example (Linked lists): We can define a linked list as a `class Node[T](datum: T, next: Node)`, with `next` being `null` if there is no next node. We can represent the abstract state of a linked list a as $L(a)$, the list of nodes accessible from a. We can represent $L(a)$ using Haskell list notation:

$$L(a) := \begin{cases} [] & \text{if } a = \text{null} \\ a.\text{datum} : L(a.\text{next}) & \text{otherwise.} \end{cases}$$

Remark: Scala is garbage-collected, so an object will be cleared up once there are no references to it; we can forcibly drop a reference by setting an occurence to `null`.

Remark: In scala, `null` is the null pointer, not a null value. So it can be used anywhere where we'd usually have an object reference.

Remark: If we want to have an internal class used inside another class, we should declare is as a `private class` inside a companion object to the outer class, so that every member of the class uses the same class, but it is not visible to outer scopes.

The `List[A]` type is an immutable sequence, like Haskell lists, with a cons operator `::`, `head`,`tail` and `isEmpty` operators, `:::` to concat two lists, and a `List.Nil` constructor.

`scala.collection.mutable.Queue` is a FIFO data structure. Operations are `q.enqueue(x)`, `d.dequeue()` and `q.isEmpty`. TODO abstract spec

Definition 4.10: Factory methods allow us to do something resembling partial application of functions. Example:

```
def mul(a: Int)(b: Int): Int = {
  a * b
}
```

And we can call this as e.g. `a(3)(7)`. We can sort of partially apply it as `val mul10 = mul(10)(_)`.

Remark: We can have variable length arguments as the last argument of a parameter list, denoted by `T*`.

```
object BitMapSet{
  def apply(N: Int)(xs: Int*) : BitMapSet = {
    val s = new BitMapSet(N); for(x <- xs) s.add(x); s
  }
}
```

If we want to pass an existing sequence `s` in to a function expecting a variable number of arguments, use the notation `s: _*`; so to create a bitmap set factory, we could do something like `val bitMapSetFactory = BitMapSet(10)(_: _*)`.

Definition 4.11: A hash table stored elements of type `T` in `N` buckets, determining which bucket to put some `t: T` in by computing `hash(t)`, where `hash: T => [0..N)`. Then if the buckets are small, searching for an item is inexpensive.

We could implement each bucket as a linked list. The hash table itself is then an array of such linked lists.

We want the hash function to give uniformly distributed integers, so that we don't get too many hash collisions and the bucket sizes stay as small as possible.

Remark: Defining a good hash function is difficult. We define a hash function for a string $s = c_0 c_1 ... c_{n-1}$

$$(p^n + c_0.\text{toInt} \cdot p^{n-1} + c_1.\text{toInt} \cdot p^{n-2} + ... + c_{n-1}.\text{toInt}) \bmod N,$$

where p is an odd prime (and N is the size of our hash table).

By Horner's rule, this polynomial is equivalent to

$$((...(((p + c_0.\text{toInt}) \cdot p + c_1.\text{toInt}) \cdot p + c_2.\text{toInt})...) \cdot p + c_{n-1}.\text{toInt}) \bmod N$$

which is equal to

$$((...(((p + c_0.\text{toInt}) \bmod N \cdot p + c_1.\text{toInt}) \bmod N \cdot p + c_2.\text{toInt}) \bmod N...) \cdot p + c_{n-1}.\text{toInt}) \bmod N$$

to avoid overflow.

No need to understand whether/why this is a good hash function.

Remark: If we find that our hash table is getting a lot of elements per bucket (on average), we could dynamically increase the number of buckets and redistribute elements at a certain threshold.

This threshold should be below 1 element per bucket, e.g. 0.75 elements per bucket, because as soon as we get more than one element per bucket, operations start getting a lot slower.

Resizing the table is $O(N)$, but only happens every $\frac{N}{2}$ insertions (after the first resize), so has constant amortised cost.

Index