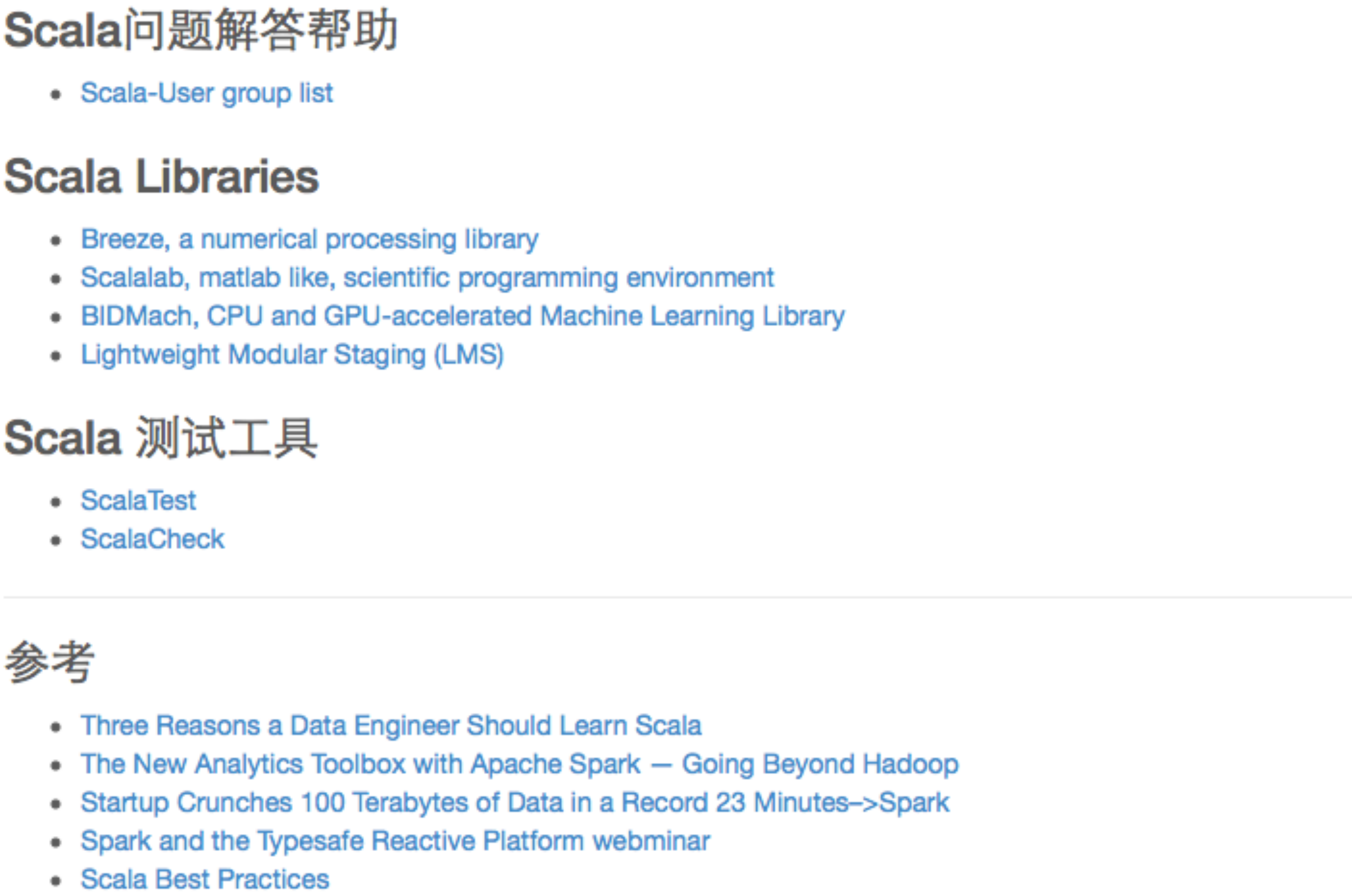
今天的内容

- 作业1讲解
- 可变状态对象举例
  - 离散事件模拟器
- 作业2预览讲解

Scala 测试工具

- ScalaTest
  - http://www.scalatest.org
- ScalaCheck
  - http://www.scalacheck.org



Scala问题解答帮助

- Scala-User group list

Scala Libraries

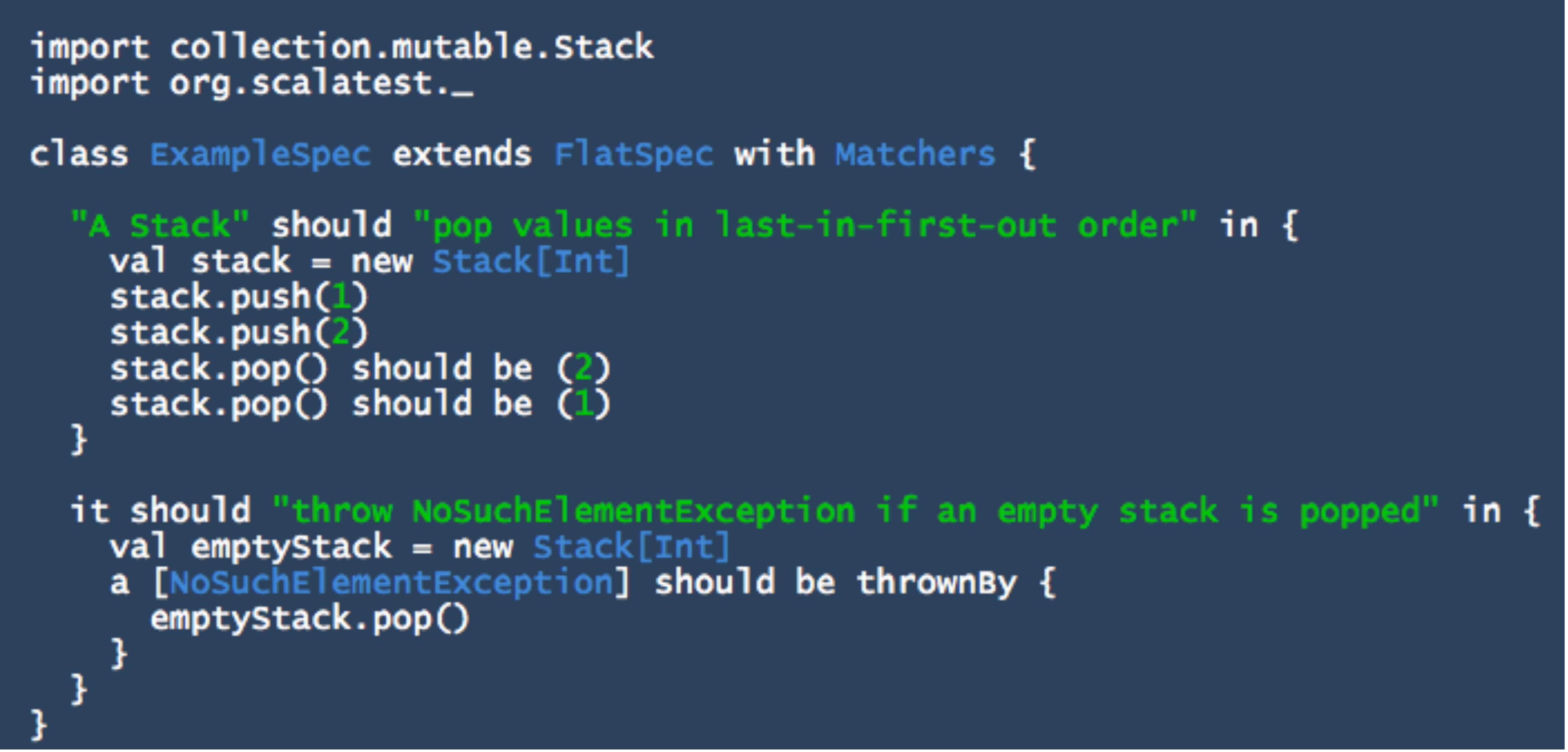
- Breeze, a numerical processing library
- Scalalab, matlab like, scientific programming environment
- BIDMach, CPU and GPU-accelerated Machine Learning Library
- Lightweight Modular Staging (LMS)

Scala 测试工具

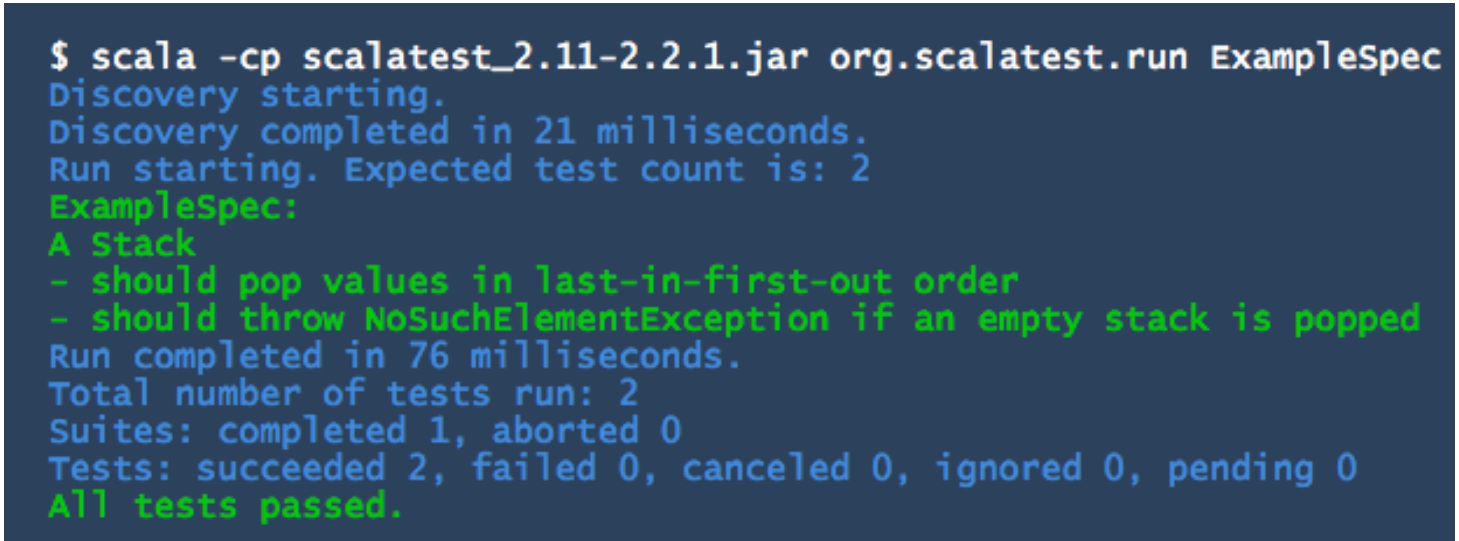
- ScalaTest
- ScalaCheck

参考

- Three Reasons a Data Engineer Should Learn Scala
- The New Analytics Toolbox with Apache Spark — Going Beyond Hadoop
- Startup Crunches 100 Terabytes of Data in a Record 23 Minutes–>Spark
- Spark and the Typesafe Reactive Platform webminar
- Scala Best Practices



```
import collection.mutable.Stack
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {

  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should be (2)
    stack.pop() should be (1)
  }

  it should "throw NoSuchElementException if an empty stack is popped" in {
    val emptyStack = new Stack[Int]
    a [NoSuchElementException] should be thrownBy {
      emptyStack.pop()
    }
  }
}
```



```
$ scala -cp scalatest_2.11-2.2.1.jar org.scalatest.run ExampleSpec
Discovery starting.
Discovery completed in 21 milliseconds.
Run starting. Expected test count is: 2
ExampleSpec:
A Stack
- should pop values in last-in-first-out order
- should throw NoSuchElementException if an empty stack is popped
Run completed in 76 milliseconds.
Total number of tests run: 2
Suites: completed 1, aborted 0
Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
```

ScalaCheck

Shift in viewpoint: Instead of writing tests, write *properties* that are assumed to hold.

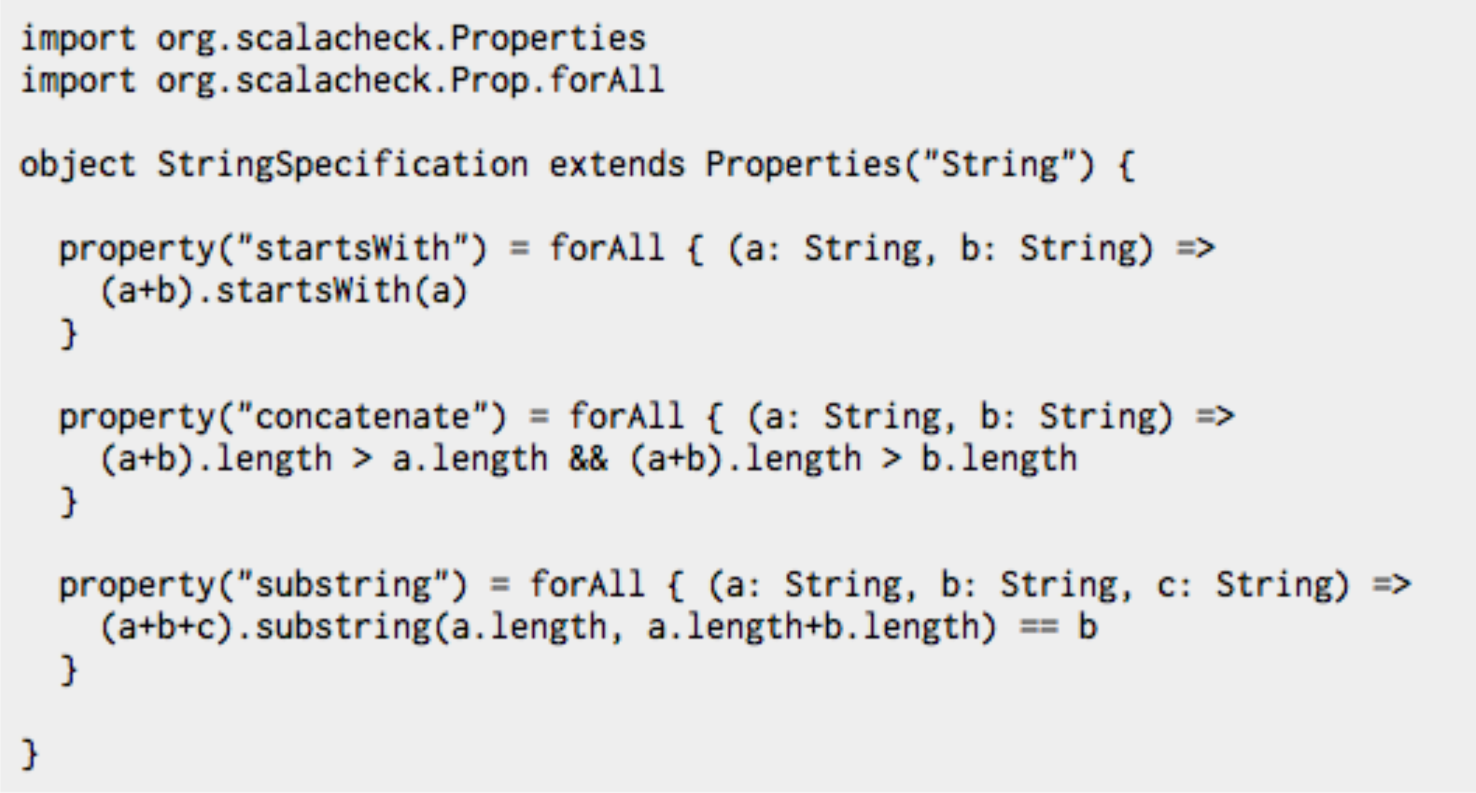
This idea is implemented in the `ScalaCheck` tool.

```
forAll { (l1: List[Int], l2: List[Int]) =>
  l1.size + l2.size == (l1 ++ l2).size
}
```

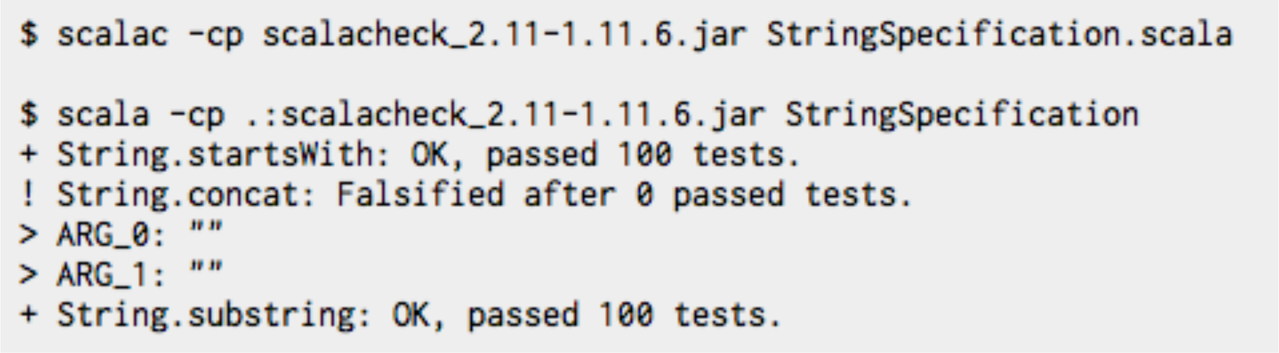
It can be used either stand-alone or as part of ScalaTest.

See ScalaCheck tutorial on the course page.

ScalaCheck



```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll

object StringSpecification extends Properties("String") {

  property("startsWith") = forAll { (a: String, b: String) =>
    (a+b).startsWith(a)
  }

  property("concatenate") = forAll { (a: String, b: String) =>
    (a+b).length > a.length && (a+b).length > b.length
  }

  property("substring") = forAll { (a: String, b: String, c: String) =>
    (a+b+c).substring(a.length, a.length+b.length) == b
  }

}
```



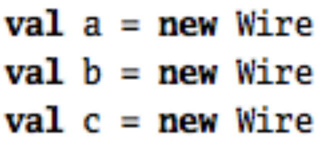
```
$ scalac -cp scalacheck_2.11-1.11.6.jar StringSpecification.scala

$ scala -cp .:scalacheck_2.11-1.11.6.jar StringSpecification
+ String.startsWith: OK, passed 100 tests.
! String.concat: Falsified after 0 passed tests.
> ARG_0: ""
> ARG_1: ""
+ String.substring: OK, passed 100 tests.
```

可改变的状态(Mutable State)

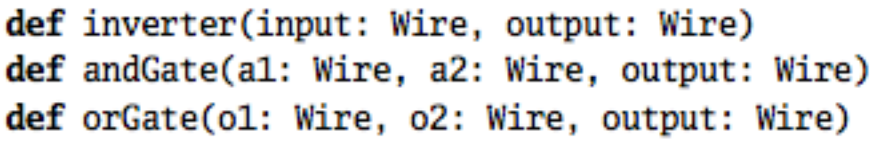
举例：Discrete Event Simulation

举例：离散事件模拟

- 电子电路模拟器
  - 线路（信号），函数方盒（与，或，非门）
  - 信号：布尔值 **true**，**false**
  - 方盒有延迟
- 描述语言



```
val a = new Wire
val b = new Wire
val c = new Wire
```



```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

Extended Example: Discrete Event Simulation

Here’s an example that shows how assignments and higher-order functions can be combined in interesting ways.

We will construct a digital circuit simulator.

The simulator is based on a general framework for discrete event simulation.

Digital Circuits

Let’s start with a small description language for digital circuits.

A digital circuit is composed of *wires* and of functional components.

Wires transport signals that are transformed by components.

We represent signals using booleans true and false.

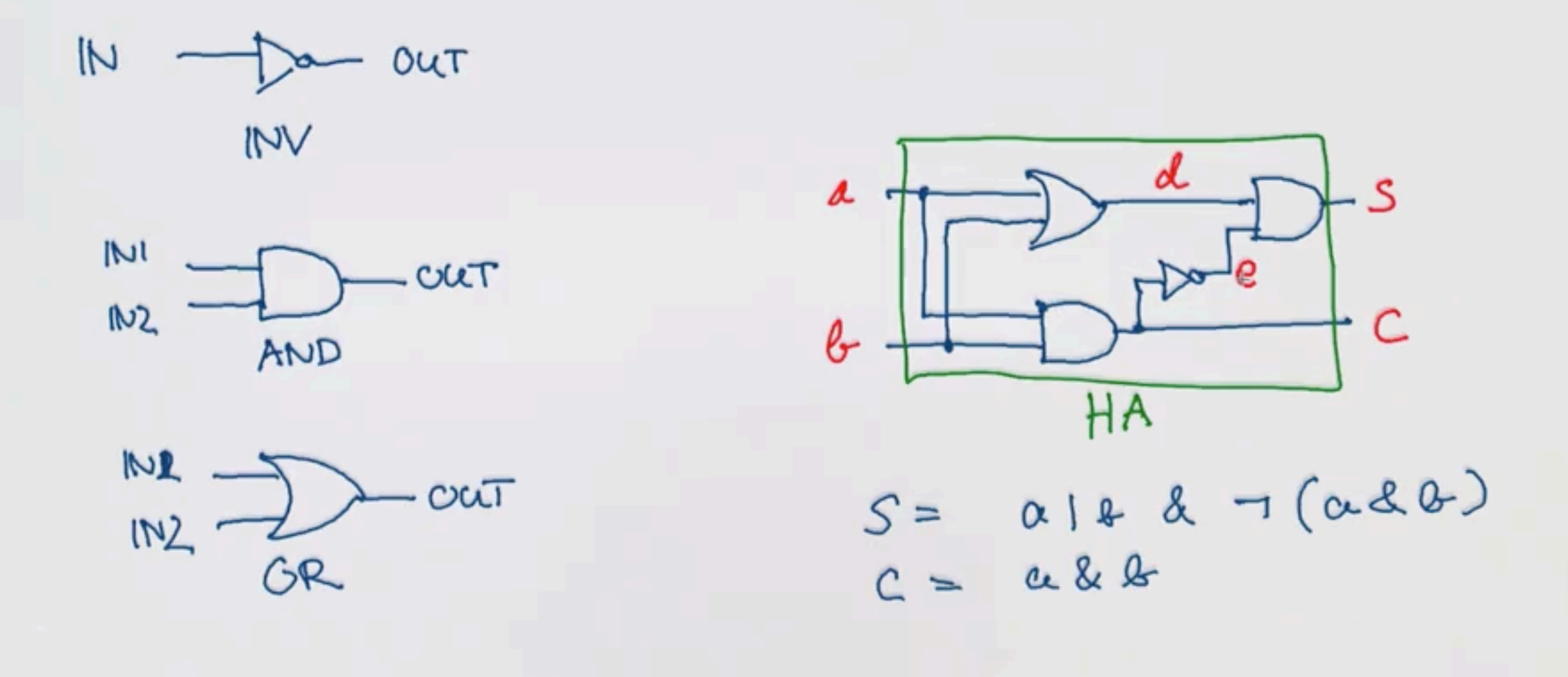
The base components (gates) are:

- The *Inverter*, whose output is the inverse of its input.
- The *AND Gate*, whose output is the conjunction of its inputs.
- The *OR Gate*, whose output is the disjunction of its inputs.

Other components can be constructed by combining these base components.

The components have a reaction time (or *delay*), i.e. their outputs don’t change immediately after a change to their inputs.

Digital Circuit Diagrams



$S = a \mid b \ \& \ \neg(a \& b)$

$C = a \& b$

A Language for Digital Circuits

We describe the elements of a digital circuit using the following Scala classes and functions.

To start with, the class Wire models wires.

Wires can be constructed as follows:

```
val a = new Wire; val b = new Wire; val c = new Wire
```

or, equivalently:

```
val a, b, c = new Wire
```

Gates

Then, there are the following functions. Each has a side effect that creates a gate.

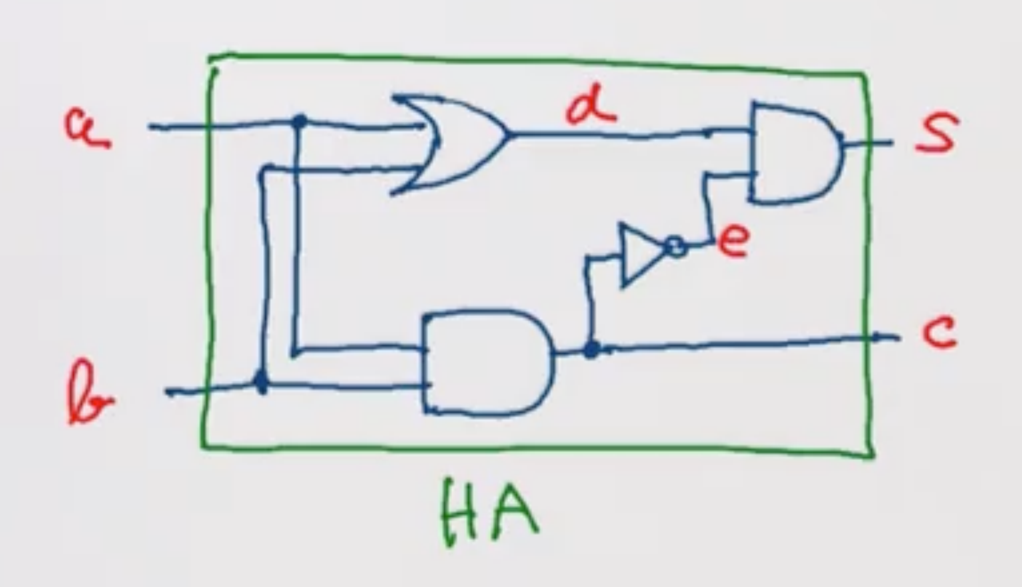
```
def inverter(input: Wire, output: Wire): Unit

def andGate(a1: Wire, a2: Wire, output: Wire): Unit
def orGate(o1: Wire, o2: Wire, output: Wire): Unit
```

Constructing Components

More complex components can be constructed from these.

For example, a half-adder can be defined as follows:

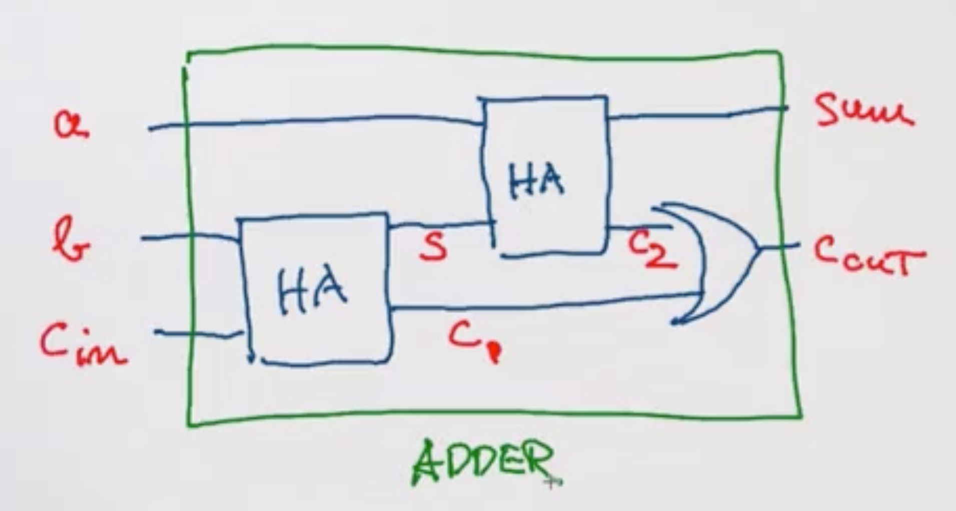
```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire): Unit = {
  val d = new Wire
  val e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)

  andGate(d, e, s)
}
```



More Components

This half-adder can in turn be used to define a full adder:

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire): Unit = {
  val s = new Wire
  val c1 = new Wire
  val c2 = new Wire

  halfAdder(b, cin, s, c1)
  halfAdder(a, s, sum, c2)
  orGate(c1, c2, cout)

}
```



Exercise

What logical function does this program describe?

```
 def f(a: Wire, b: Wire, c: Wire): Unit = {
     val d, e, f, g = new Wire
     inverter(a, d)
     inverter(b, e)
     andGate(a, e, f)
     andGate(b, d, g)
     orGate(f, g, c)
}
```

○ a & ~b

○ a & ~(b & a)

○ b & ~a

○ a == b

○ a != b

○ a * b

Discrete Event Simulation: API and Usage

How to Make it Work?

The class Wire and the functions inverter, andGate, and orGate represent a small description language of digital circuits.

We now give the implementation of this class and its functions which allow us to simulate circuits.

These implementations are based on a simple API for discrete event simulation.

Discrete Event Simulation

A discrete event simulator performs *actions*, specified by the user at a given *moment*.

An *action* is a function that doesn't take any parameters and which returns Unit:

```
type Action = () => Unit
```

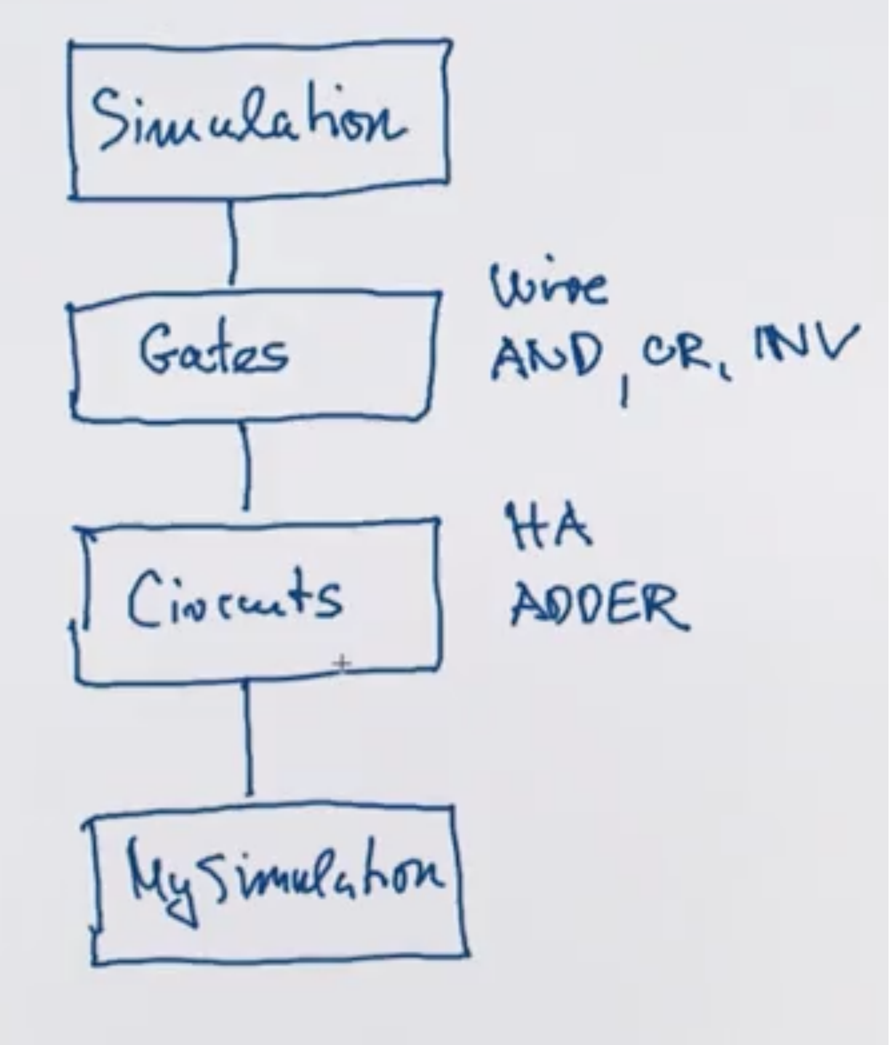
The *time* is simulated; it has nothing to with the actual time.

Simulation Trait

A concrete simulation happens inside an object that inherits from the trait Simulation, which has the following signature:

```
trait Simulation {
  /** The current simulated time */
  def currentTime: Int = ???


  /** Registers an action ‘block‘ to perform after a given delay
   *  relative to the current time */
  def afterDelay(delay: Int)(block: => Unit): Unit = ???

  /** Performs the simulation until there are no actions waiting */
  def run(): Unit = ???
}
```

Class Diagram



The Wire Class

A wire must support three basic operations:

getSignal: Boolean

Returns the current value of the signal transported by the wire.

setSignal(sig: Boolean): Unit

Modifies the value of the signal transported by the wire.

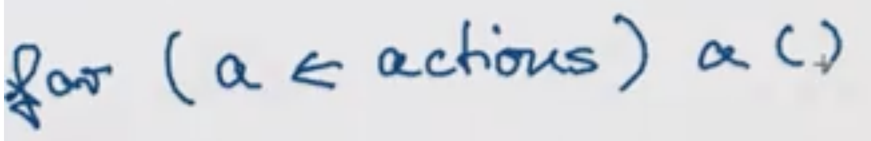
addAction(a: Action): Unit

Attaches the specified procedure to the *actions* of the wire. All of the attached actions are executed at each change of the transported signal.

Implementing Wires

Here is an implementation of the class Wire:

```
class Wire {
  private var sigVal = false
  private var actions: List[Action] = Nil
  def getSignal: Boolean = sigVal
  def setSignal(s: Boolean): Unit =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_())
    }
  def addAction(a: Action): Unit = {
    actions = a :: actions
    a()
  }
}
```



```
for (a <- actions) a()
```

Play the action immediately

State of a Wire

The state of a wire is modeled by two private variables:

sigVal represents the current value of the signal.

actions represents the actions currently attached to the wire.

The Inverter

We implement the inverter by installing an action on its input wire.

This action produces the inverse of the input signal on the output wire.

The change must be effective after a delay of `InverterDelay` units of simulated time.

The Inverter

We implement the inverter by installing an action on its input wire.

This action produces the inverse of the input signal on the output wire.

The change must be effective after a delay of InverterDelay units of simulated time.

We thus obtain the following implementation:

```
def inverter(input: Wire, output: Wire): Unit = {
  def invertAction(): Unit = {
    val inputSig = input.getSignal

    afterDelay(InverterDelay) { output setSignal !inputSig }
  }
  input addAction invertAction
}
```

The AND Gate

The AND gate is implemented in a similar way.

The action of an AND gate produces the conjunction of input signals on the output wire.

This happens after a delay of `AndGateDelay` units of simulated time.

The AND Gate

The AND gate is implemented in a similar way.

The action of an AND gate produces the conjunction of input signals on the output wire.

This happens after a delay of `AndGateDelay` units of simulated time.

We thus obtain the following implementation:

```
def andGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def andAction(): Unit = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal

    afterDelay(AndGateDelay) { output setSignal (in1Sig & in2Sig) }
  }
  in1 addAction andAction
  in2 addAction andAction
}
```

The OR Gate

The OR gate is implemented analogously to the AND gate.

```
def andGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def andAction(): Unit = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal

    afterDelay(AndGateDelay) { output setSignal (in1Sig & in2Sig) }
  }
  in1 addAction andAction
  in2 addAction andAction
}
```

The OR Gate

The OR gate is implemented analogously to the AND gate.

```
def orGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def orAction(): Unit = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal

    afterDelay(OrGateDelay) { output setSignal (in1Sig | in2Sig) }
  }
  in1 addAction orAction
  in2 addAction orAction
}
```

Exercise

What happens if we compute in1Sig and in2Sig inline inside afterDelay instead of computing them as values?

```
def orGate2(in1: Wire, in2: Wire, output: Wire): Unit = {
  def orAction(): Unit = {
    afterDelay(OrGateDelay) {
      output setSignal (in1.getSignal | in2.getSignal) }
    }
  in1 addAction orAction
  in2 addAction orAction
}
```

○ ‘orGate‘ and ‘orGate2‘ have the same behavior.
○ ‘orGate2‘ does not model OR gates faithfully.

Implementation and Test

The Simulation Trait

All we have left to do now is to implement the Simulation trait.

The idea is to keep in every instance of the Simulation trait an *agenda* of actions to perform.

The agenda is a list of (simulated) *events*. Each event consists of an action and the time when it must be produced.

The agenda list is sorted in such a way that the actions to be performed first are in the beginning.

```
trait Simulation {
  type Action = () => Unit
  case class Event(time: Int, action: Action)
  private type Agenda = List[Event]
  private var agenda: Agenda = List()
```

Handling Time

There is also a private variable, curtime, that contains the current simulation time:

```
private var curtime = 0
```

It can be accessed with a getter function currentTime:

```
def currentTime: Int = curtime
```

An application of the afterDelay(delay)(block) method inserts the task

```
Event(curtime + delay, () => block)
```

into the agenda list at the right position.

Implementing AfterDelay

```
def afterDelay(delay: Int)(block: => Unit): Unit = {
  val item = Event(currentTime + delay, () => block)
  agenda = insert(agenda, item)

}
```

Implementing AfterDelay

```
def afterDelay(delay: Int)(block: => Unit): Unit = {
  val item = Event(currentTime + delay, () => block)
  agenda = insert(agenda, item)

}
```

The insert function is straightforward:

```
private def insert(ag: List[Event], item: Event): List[Event] = ag match {
  case first :: rest if first.time <= item.time =>
    first :: insert(rest, item)
  case _ =>
    item :: ag
}
```

The Event Handling Loop

The event handling loop removes successive elements from the agenda, and performs the associated actions.

```
private def loop(): Unit = agenda match {
  case first :: rest =>
    agenda = rest

    curtime = first.time
    first.action()
    loop()

  case Nil =>
}
```

The Run Method

The run method executes the event loop after installing an initial message that signals the start of simulation.

```
def run(): Unit = {
  afterDelay(0) {
    println(”*** simulation started, time = ”+currentTime+” ***”)
  }
  loop()
}
```

Probes

Before launching the simulation, we still need a way to examine the changes of the signals on the wires.

To this end, we define the function probe.

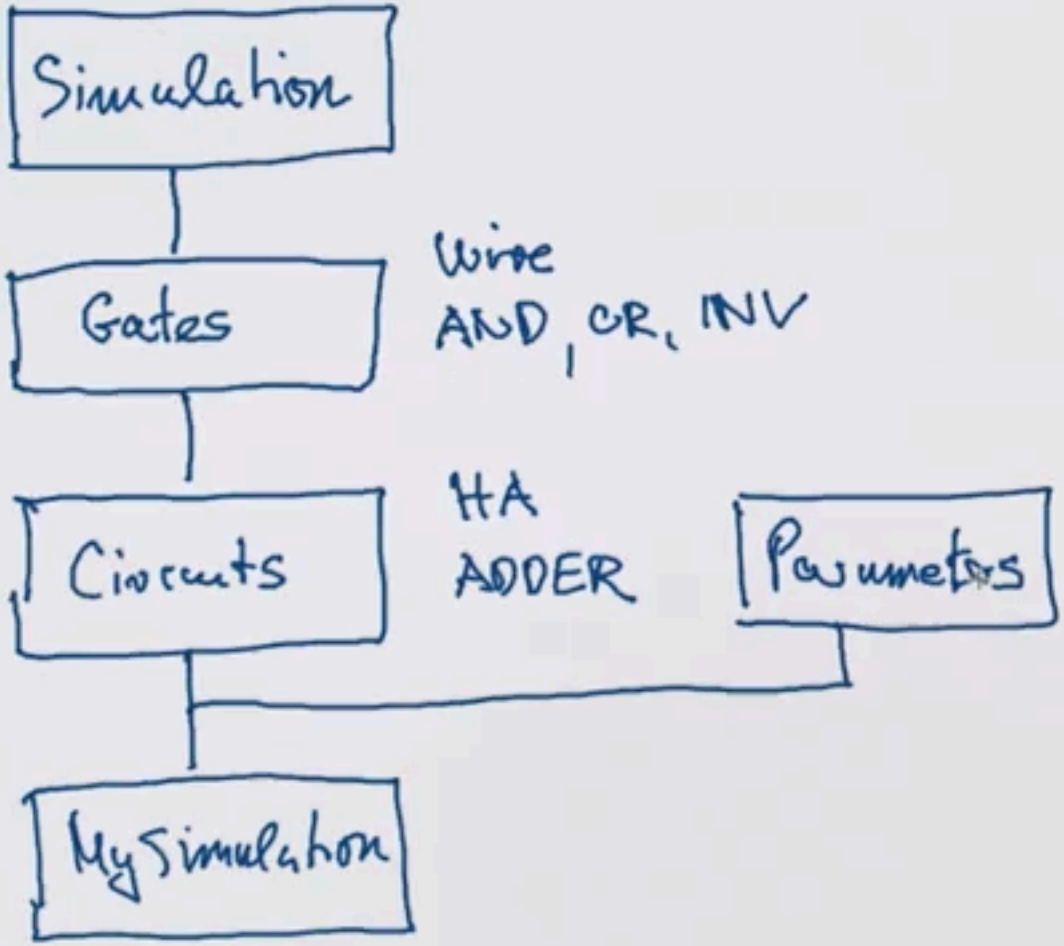
```
def probe(name: String, wire: Wire): Unit = {
  def probeAction(): Unit = {
    println(s”$name $currentTime value = ${wire.getSignal}”)
  }
  wire addAction probeAction
}
```

Defining Technology-Dependent Parameters

It’s convenient to pack all delay constants into their own trait which can be mixed into a simulation. For instance:

```
trait   Parameters  {
   def InverterDelay  =  2
   def AndGateDelay  =  3
   def OrGateDelay  =  5

}

object  sim  extends  Circuits  with  Parameters
```

Class Diagram



Setting Up a Simulation

Here’s a sample simulation that you can do in the worksheet.

Define four wires and place some probes.

```
import sim._

val input1, input2, sum, carry = new Wire
probe(”sum”, sum)

probe(”carry”, carry)
```

Next, define a half-adder using these wires:

```
halfAdder(input1, input2, sum, carry)
```

Launching the Simulation

Now give the value true to input1 and launch the simulation:

```
input1.setSignal(true)
run()
```

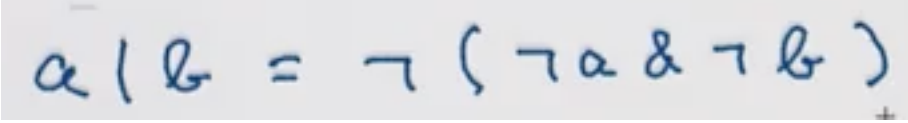
To continue:

```
input2.setSignal(true)
run()
```

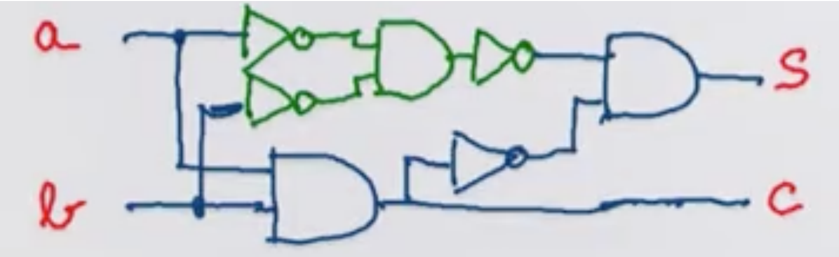
A Variant

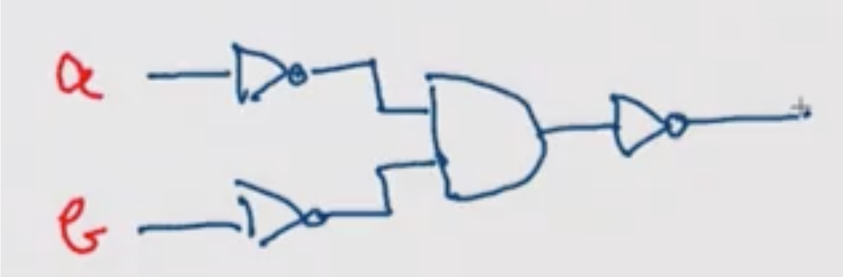
An alternative version of the OR-gate can be defined in terms of AND and INV.

```
def orGateAlt(in1: Wire, in2: Wire, output: Wire): Unit = {
  val notIn1, notIn2, notOut = new Wire
  inverter(in1, notIn1); inverter(in2, notIn2)
  andGate(notIn1, notIn2, notOut)
  inverter(notOut, output)
}
```



$a \mid b = \neg(\neg a \,\&\, \neg b)$





Exercise

Question: What would change in the circuit simulation if the implementation of orGateAlt was used for OR?

○ Nothing. The two simulations behave the same.

○ The simulations produce the same events, but the indicated times are different.

○ The times are different, and orGateAlt may also produce additional events.

○ The two simulations produce different events altogether.

Summary

State and assignments make our mental model of computation more complicated.

In particular, we lose referential transparency.

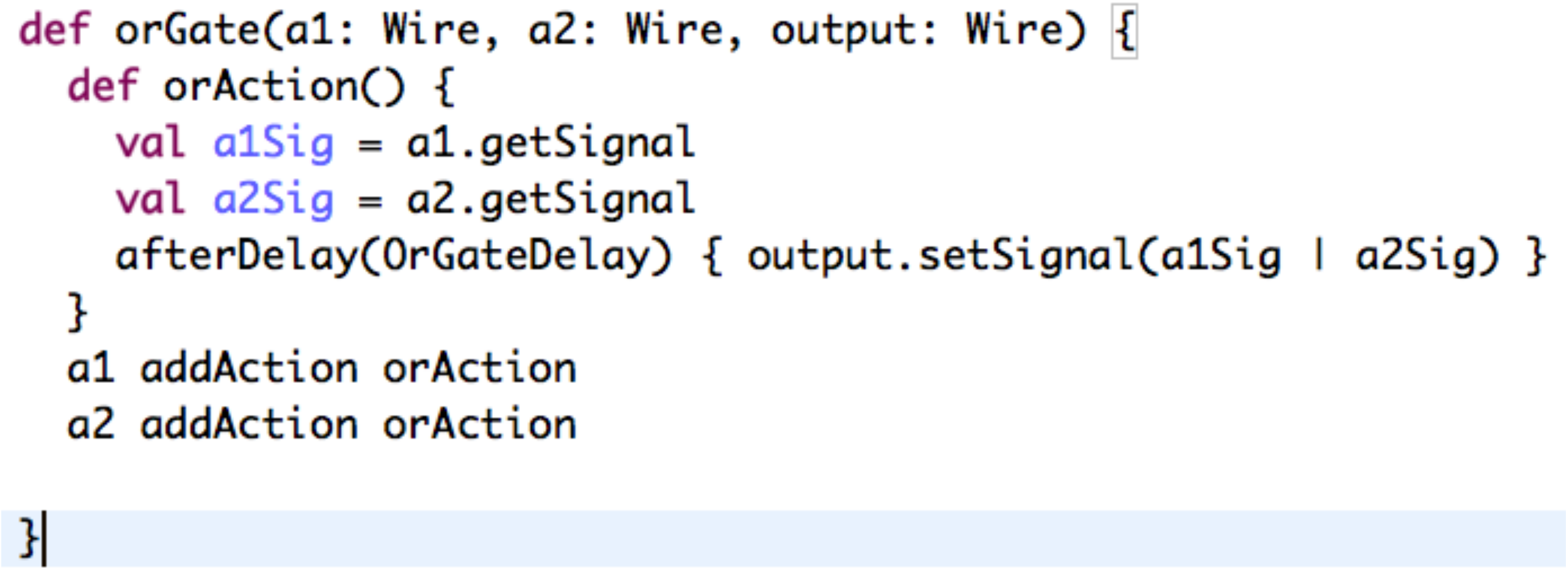
On the other hand, assignments allow us to formulate certain programs in an elegant way.

Example: discrete event simulation.

- Here, a system is represented by a mutable list of *actions*.
- The effect of actions, when they're called, change the state of objects and can also install other actions to be executed in the future.

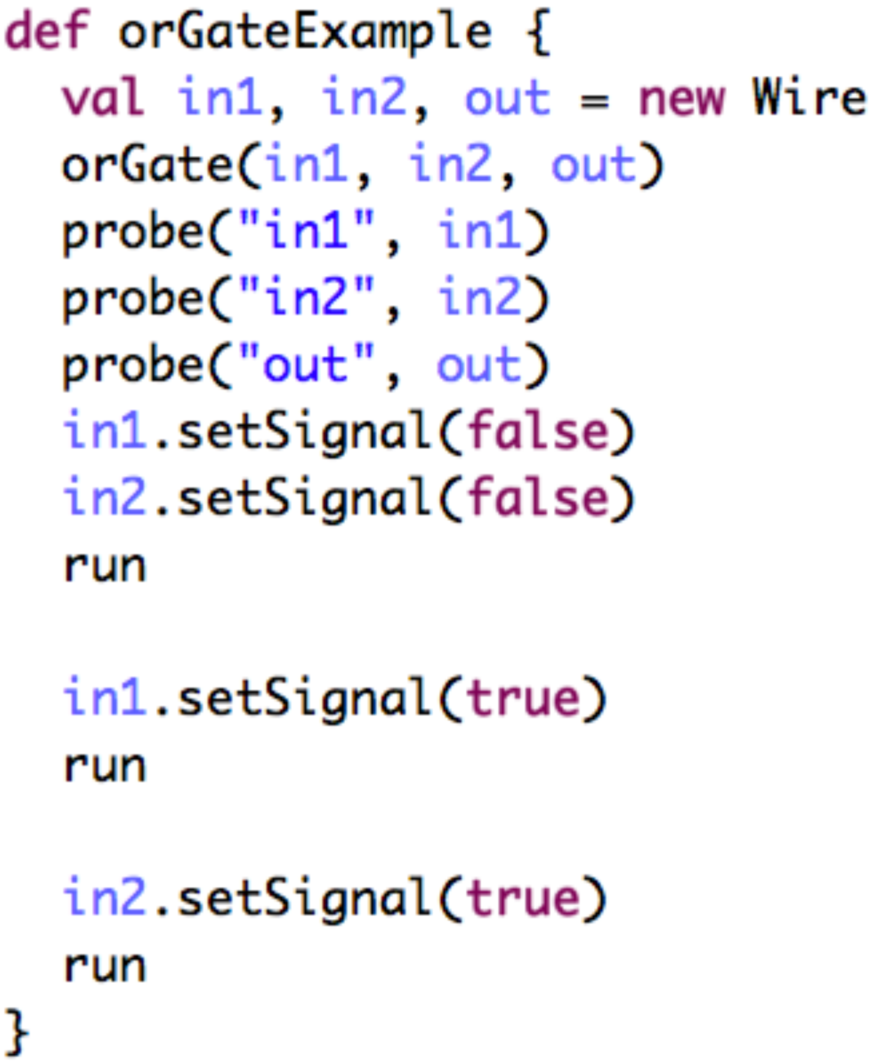
作业2 提示

orGate 已实现；orGate2需实现；demux函数选做



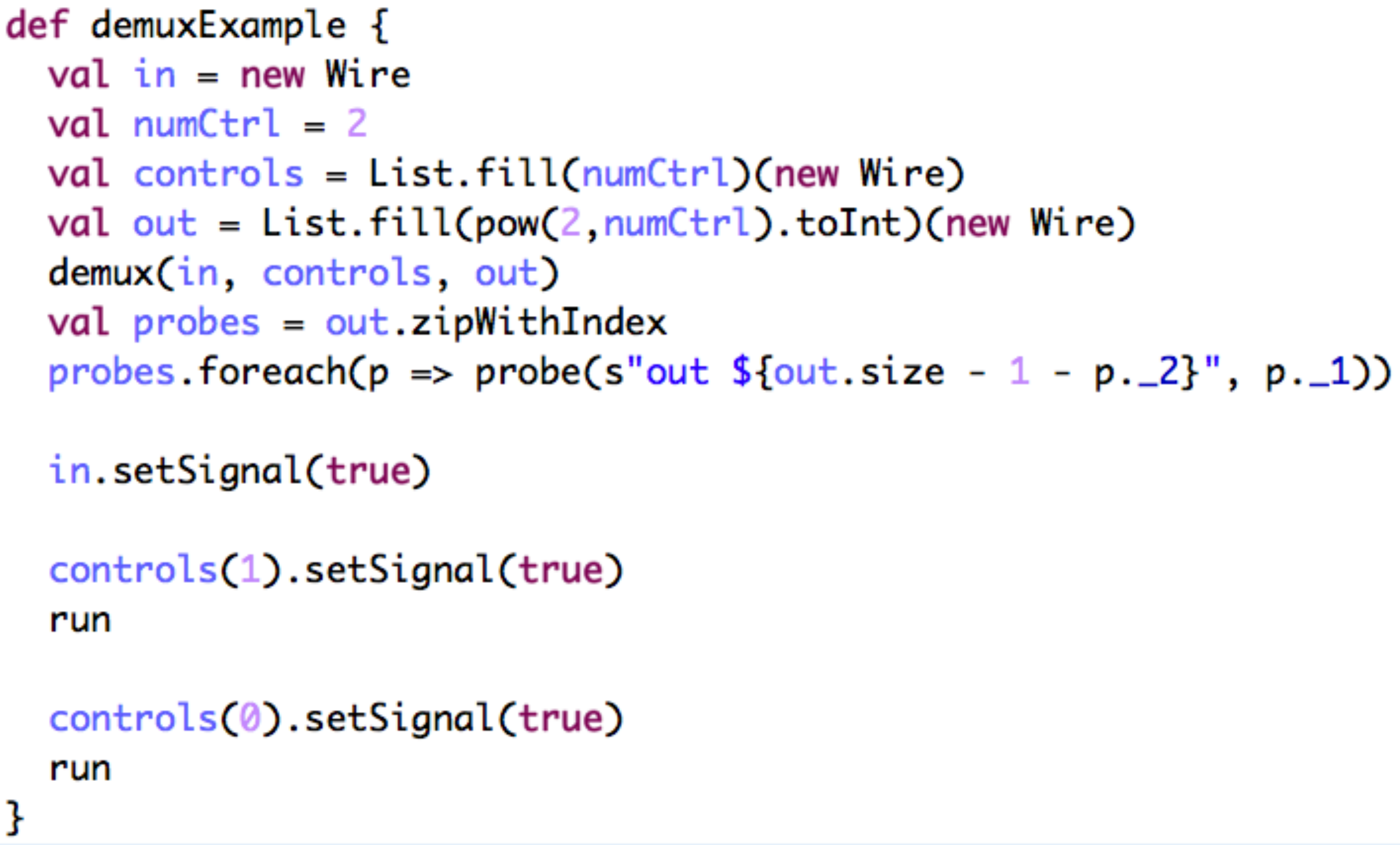
```
def orGate(a1: Wire, a2: Wire, output: Wire) {
  def orAction() {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(OrGateDelay) { output.setSignal(a1Sig | a2Sig) }
  }
  a1 addAction orAction
  a2 addAction orAction

}
```

orGate的测试例子



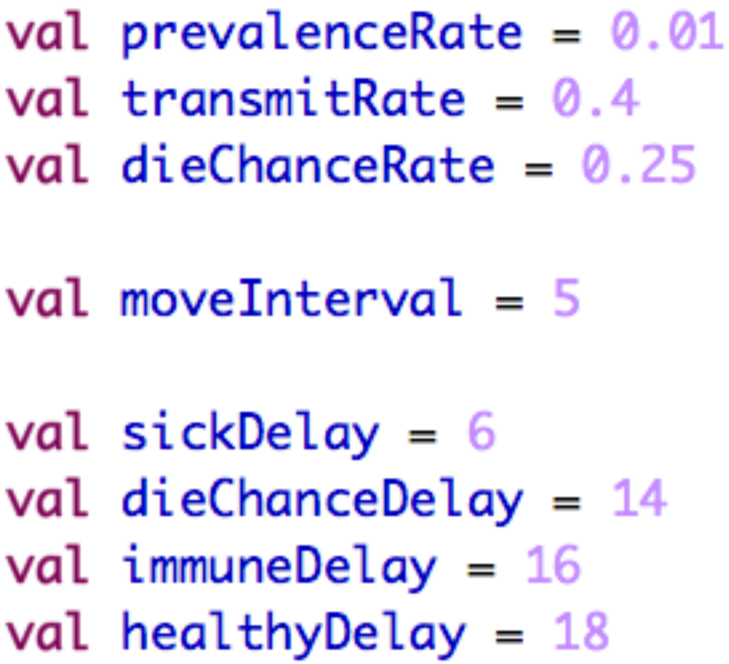
```
def orGateExample {
  val in1, in2, out = new Wire
  orGate(in1, in2, out)
  probe("in1", in1)
  probe("in2", in2)
  probe("out", out)
  in1.setSignal(false)
  in2.setSignal(false)
  run

  in1.setSignal(true)
  run

  in2.setSignal(true)
  run
}
```

Demux的测试例子



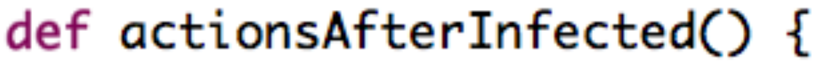
```scala
def demuxExample {
  val in = new Wire
  val numCtrl = 2
  val controls = List.fill(numCtrl)(new Wire)
  val out = List.fill(pow(2,numCtrl).toInt)(new Wire)
  demux(in, controls, out)
  val probes = out.zipWithIndex
  probes.foreach(p => probe(s"out ${out.size - 1 - p._2}", p._1))

  in.setSignal(true)

  controls(1).setSignal(true)
  run

  controls(0).setSignal(true)
  run
}
```

疾病控制策略模拟器 // to complete: additional parameters of simulation

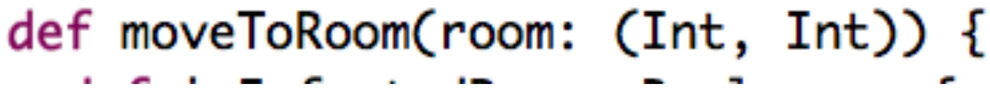


```
val prevalenceRate = 0.01
val transmitRate = 0.4
val dieChanceRate = 0.25

val moveInterval = 5

val sickDelay = 6
val dieChanceDelay = 14
val immuneDelay = 16
val healthyDelay = 18
```

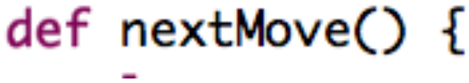
疾病控制策略模拟器
To complete with simulation logic:



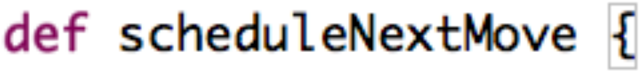
```
def actionsAfterInfected() {
```



```
def moveToRoom(room: (Int, Int)) {
```



```
def nextMove() {
```



```
def scheduleNextMove {
```