香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 14)

COMP 3031, Fall 2025

Lionel Parreaux

Functions and State

Functions and State

Until now, our programs have been free of side effect.

Therefore, the concept of *time* wasn't important.

For all programs that terminate, any sequence of actions would have given the same results.

This was also reflected in the substitution model of computation.

Reminder: Substitution Model

Programs can be evaluated by *rewriting*.

The most important rewrite rule covers function applications:

$$\begin{array}{l} \texttt{def f}(x_1, \ldots, x_n) = B;\ \ldots\ \texttt{f}(v_1, \ldots, v_n) \\ \rightarrow \\ \texttt{def f}(x_1, \ldots, x_n) = B;\ \ldots\ [v_1/x_1, \ldots, v_n/x_n]\, B \end{array}$$

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if n == 0 then x else iterate(n - 1, f, f(x))
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if n == 0 then x else iterate(n - 1, f, f(x))
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

$\rightarrow$ `if 1 == 0 then 3 else iterate(1-1, square, square(3))`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if n == 0 then x else iterate(n - 1, f, f(x))
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

$\rightarrow$ `if 1 == 0 then 3 else iterate(1-1, square, square(3))`

$\rightarrow$ `iterate(0, square, square(3))`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if n == 0 then x else iterate(n - 1, f, f(x))
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

$\rightarrow$ `if 1 == 0 then 3 else iterate(1-1, square, square(3))`

$\rightarrow$ `iterate(0, square, square(3))`

$\rightarrow$ `iterate(0, square, 3 * 3)`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if n == 0 then x else iterate(n - 1, f, f(x))
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

$\rightarrow$ `if 1 == 0 then 3 else iterate(1-1, square, square(3))`

$\rightarrow$ `iterate(0, square, square(3))`

$\rightarrow$ `iterate(0, square, 3 * 3)`

$\rightarrow$ `iterate(0, square, 9)`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if n == 0 then x else iterate(n - 1, f, f(x))
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

→ `if 1 == 0 then 3 else iterate(1-1, square, square(3))`

→ `iterate(0, square, square(3))`

→ `iterate(0, square, 3 * 3)`

→ `iterate(0, square, 9)`

→ `if 0 == 0 then 9 else iterate(0-1, square, square(9))` → `9`

Observation:

Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution.

This is an important result of the λ-calculus, the theory behind functional programming. (Church–Rosser theorem.)

Example:

```
if 1 == 0 then 3 else iterate(1 - 1, square, square(3))
```

Observation:

Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution.

This is an important result of the λ-calculus, the theory behind functional programming. (Church–Rosser theorem.)

Example:

```
if 1 == 0 then 3 else iterate(1 - 1, square, square(3))
```

Stateful Objects

One normally describes the world as a set of objects, some of which have state that *changes* over the course of time.

An object *has a state* if its behavior is influenced by its history.

Example: a bank account has a state, because the answer to the question

"Can I withdraw 100 HKD?"

may vary over the course of the lifetime of the account.

Implementation of State

Every form of mutable state can be constructed from variables.

A variable definition is written like a value definition, but with the keyword `var` in place of `val`:

```
var x: String = "abc"
var count = 111
```

Just like a value definition, a variable definition associates a value with a name.

However, in the case of variable definitions, this association can be changed later through mutable *assignment*, like in Java:

```
x = "hi"
count = count + 1
```

State in Objects

In practice, objects with state are usually represented by objects that have some variable members. For instance, here is a class modeling a bank account.

```
class BankAccount:
  private var balance = 0

  def deposit(amount: Int): Unit =
    if amount > 0 then balance = balance + amount

  def withdraw(amount: Int): Int =
    if 0 < amount && amount <= balance then
      balance = balance - amount
      balance
    else throw Error("insufficient funds")
```

State in Objects (2)

The class `BankAccount` defines a variable `balance` that contains the current balance of the account.

The methods `deposit` and `withdraw` change the value of the `balance` through mutable assignments.

Note that `balance` is `private` in the `BankAccount` class, it therefore cannot be accessed from outside the class.

To create bank accounts, we use the usual notation for object creation:

```
val account = BankAccount()
```

Working with Mutable Objects

Here is a worksheet that manipulates bank accounts.

```
val account = BankAccount()          // account: BankAccount = ...
account.deposit(50)                  //
account.withdraw(20)                 // : Int = 30
account.withdraw(20)                 // : Int = 10
account.withdraw(15)                 // java.lang.Error: insufficient funds
```

Applying the same operation to an account twice in a row produces different results. Clearly, accounts are stateful objects.

Statefulness and Variables

Remember the implementation of `TailLazyList`. Instead of using a lazy val, we could also implement non-empty lazy lists using a mutable variable:

```
def cons[T](hd: T, tl: => TailLazyList[T]) = new TailLazyList[T]:
  def head = hd
  private var tlOpt: Option[TailLazyList[T]] = None
  def tail: T = tlOpt match
    case Some(x) => x
    case None => tlOpt = Some(tl); tail
```

Question: Is the result of `cons` a stateful object?

- Yes
- No

Statefulness and Variables

Remember the implementation of `TailLazyList`. Instead of using a lazy val, we could also implement non-empty layz lists using a mutable variable:

```
def cons[T](hd: T, tl: => TailLazyList[T]) = new TailLazyList[T]:
  def head = hd
  private var tlOpt: Option[TailLazyList[T]] = None
  def tail: T = tlOpt match
    case Some(x) => x
    case None => tlOpt = Some(tl); tail
```

Question: Is the result of `cons` a stateful object?

```
O        Yes
O        No
X        It depends: No, if the rest of the program is purely functional
```

Statefulness and Variables (2)

Consider the following class:

```
class BankAccountProxy(ba: BankAccount):
  def deposit(amount: Int): Unit = ba.deposit(amount)
  def withdraw(amount: Int): Int = ba.withdraw(amount)
```

Question: Are instances of `BankAccountProxy` stateful objects?

0 Yes

0 No

Statefulness and Variables (2)

Consider the following class:

```
class BankAccountProxy(ba: BankAccount):
  def deposit(amount: Int): Unit = ba.deposit(amount)
  def withdraw(amount: Int): Int = ba.withdraw(amount)
```

Question: Are instances of `BankAccountProxy` stateful objects?

X Yes

O No

Identity and Change

Identity and Change

Mutable assignments pose the new problem of deciding whether two expressions are "the same"

When one excludes mutable assignments and one writes:

```
val x = E; val y = E
```

where `E` is an arbitrary expression, then it is reasonable to assume that `x` and `y` are the same. That is to say that we could have also written:

```
val x = E; val y = x
```

(This property is usually called *referential transparency*)

Identity and Change (2)

But once we allow mutable assignments, the two formulations are different. For example:

```
val x = BankAccount()
val y = BankAccount()
```

Question: Are `x` and `y` the same?

0 Yes
0 No

Operational Equivalence

To respond to the last question, we must specify what is meant by "the same".

The precise meaning of "being the same" is defined by the property of *operational equivalence*.

In a somewhat informal way, this property is stated as follows.

Suppose we have two definitions `x` and `y`.

`x` and `y` are operationally equivalent if *no possible test* can distinguish between them.

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence `f` of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

```
val x = BankAccount()
val y = BankAccount()
S' = [x/y]S
```

- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

```
val x = BankAccount()
val y = BankAccount()
S' = [x/y]S
```

- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`
- If the results are visibly different, then the expressions `x` and `y` are certainly different.

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

```
val x = BankAccount()
val y = BankAccount()
S' = [x/y]S
```

- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`
- If the results are visibly different, then the expressions `x` and `y` are certainly different.
- On the other hand, if all possible pairs of sequences `(S, S')` produce visibly similar results, then `x` and `y` are the same.

Counterexample for Operational Equivalence

Based on this definition, let's see if the expressions

```
val x = BankAccount()
val y = BankAccount()
```

define values `x` and `y` that are the same.

Let's follow the definitions by a test sequence:

```
val x = BankAccount()
val y = BankAccount()
x.deposit(30)                    // : Int = 30
y.withdraw(20)                   // java.lang.Error: insufficient funds
```

Counterexample for Operational Equivalence (2)

Now rename all occurrences of `y` with `x` in this sequence. We obtain:

```
val x = BankAccount()
val y = BankAccount()
x.deposit(30)                    // : Int = 30
x.withdraw(20)                   // : Int = 10
```

The final results are different. We conclude that `x` and `y` are not the same.

Establishing Operational Equivalence

On the other hand, if we define

```
val x = BankAccount()
val y = x
```

then no sequence of operations can distinguish between `x` and `y`, so `x` and `y` are the same in this case.

Mutable Assignment and Substitution Model

The preceding examples show that our model of computation by substitution cannot be used.

Indeed, according to this model, one can always replace the name of a value by the expression that defines it. For example, in

```
val x = BankAccount()
val y = x
```

the `x` in the definition of `y` could be replaced by `BankAccount()`

Mutable Assignment and The Substitution Model

The preceding examples show that our model of computation by substitution cannot be used.

Indeed, according to this model, one can always replace the name of a value by the expression that defines it. For example, in

```
val x = BankAccount()
val y = x
```

```
val x = BankAccount()
val y = BankAccount()
```

the `x` in the definition of `y` could be replaced by `BankAccount()`

But we have seen that this change leads to a different program!

The substitution model ceases to be valid when we add mutation.

It is possible to adapt the substitution model by introducing a *store*, but this becomes considerably more complicated.

Loops

Loops

Proposition: Variables are enough to model all imperative programs.

But what about control statements like loops?

We can model them using functions.

Example: Here is a Scala program that uses a `while` loop:

```
def power(x: Double, exp: Int): Double =
  var r = 1.0
  var i = exp
  while i > 0 do { r = r * x; i = i - 1 }
  r
```

In Scala, `while-do` is a built-in control construct

How could we define `while` using a function (call it `whileDo`)?

Definition of `whileDo`

The function `whileDo` can be defined as follows:

```
def whileDo(condition: => Boolean)(command: => Unit): Unit =
  if condition then
    command
    whileDo(condition)(command)
  else ()
```

Example use (in Scala, `foo{E}` is equivalent to `foo({E})`):

```
whileDo (i > 0) { r = r * x; i = i - 1 }
```

Note: The condition and the command must be passed by name so that they're reevaluated in each iteration.

Note: `whileDo` is tail recursive, so it can operate with a constant stack size.

Exercise

Write a function implementing a `repeat` loop that is used as follows:

```
repeatUntil {
  command
} ( condition )
```

It should execute `command` one or more times, until `condition` is true.

Exercise

Write a function implementing a `repeat` loop that is used as follows:

```
repeatUntil {
  command
} ( condition )
```

It should execute `command` one or more times, until `condition` is true.

Solution:

```
def repeatUntil(command: => Unit)(condition: => Boolean): Unit =
  command
  if !condition then repeatUntil(command)(condition)
  else ()  // (can be omitted)
```

Exercise

Is it also possible to obtain the following syntax?

```
repeat {
  command
} until condition
```

?

Exercise

Is it also possible to obtain the following syntax?

```
repeat {
  command
} until condition
```

?

Yes!

```
def repeat(body: => Unit) = Repeat(body)
class Repeat(body: => Unit):
  infix def until(cond: => Boolean): Unit =
    body
    if !cond then until(cond)
```

For-Loops

The classical `for` loop in Java can *not* be modeled simply by a higher-order function.

The reason is that in a Java program like

```
for (int i = 1; i < 3; i = i + 1) { System.out.print(i + " "); }
```

the arguments of `for` contain the *declaration* of the variable `i`, which is visible in other arguments and in the body.

However, in Scala there is a kind of `for` loop similar to Java's extended for loop:

```
for i <- 1 until 3 do System.out.print(s"$i ")
```

This displays `1 2`.

Translation of For-Loops

For-loops translate similarly to for-expressions, but using the `foreach` combinator instead of `map` and `flatMap`.

`foreach` is defined on collections with elements of type `T` as follows:

```
def foreach(f: T => Unit): Unit =
  // apply ‘f‘ to each element of the collection
```

Example

```
for i <- 1 until 3; j <- ”abc” do println(s”$i $j”)
```

translates to:

```
(1 until 3).foreach(i => ”abc”.foreach(j => println(s”$i $j”)))
```

Discrete Event Simulation

Advanced Example: Discrete Event Simulation

We now consider an example of how assignments and higher-order functions can be combined in interesting ways.

We will construct a digital circuit simulator.

This example also shows how to build programs that do 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`.

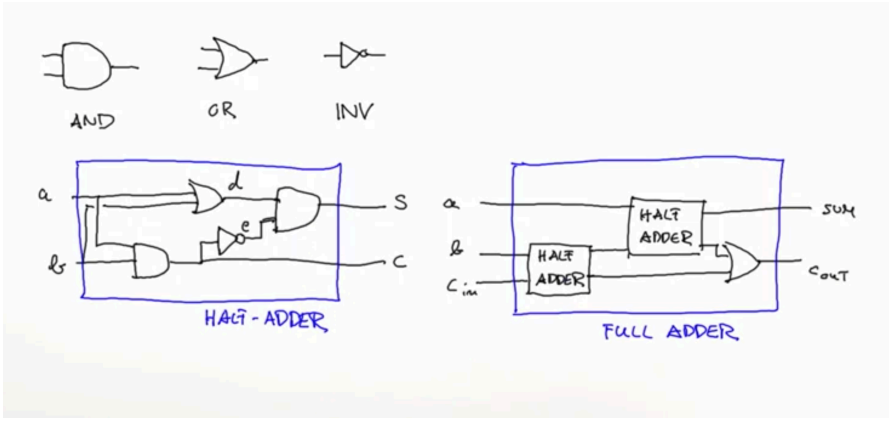
The base components (gates) are:

- 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 (Example)



A Language for Digital Circuits

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

Class `Wire` models wires. Wires can be constructed as follows:

```
val a = Wire(); val b = Wire(); val c = Wire()
```

or, equivalently:

```
val a, b, c = Wire()
```

The following functions create base components as a side effect:

```
def inverter(input: Wire, output: Wire): Unit
def andGate(in1: Wire, in2: Wire, output: Wire): Unit
def orGate(in1: Wire, in2: 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 = Wire()
  val e = 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 = Wire()
  val c1 = Wire()
  val c2 = Wire()
  halfAdder(a, cin, s, c1)
  halfAdder(b, 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 = 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`

Exercise

What logical function does this program describe?

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

O	`a & ~b`	O	`a & ~(b & a)`	O	`b & ~a`
O	`a == b`	X	`a != b`	O	`a * b`

Implementation

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.

Actions

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 abstract `Simulation` trait, which has the following signature:

```
trait Simulation:
  type Action = () => Unit
  def currentTime: Int
  def afterDelay(delay: Int)(block: => Unit): Unit
  def run(): Unit
```

Here,

`currentTime` returns the current simulated time in the form of an integer.

`afterDelay` registers an action to perform after a certain delay (relative to the current time, `currentTime`).

`run` performs the simulation until there are no more actions waiting.

Class Diagram

```
trait Simulation { ... }

trait Gates extends Simulation { ... }

trait Circuits extends Gates { ... }

object sim extends Circuits { ... }
```

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.

Implemening Wires

Here is an implementation of the class `Wire`:

```
class Wire:
  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal(): Boolean = sigVal

  def setSignal(s: Boolean): Unit =
    if s != sigVal then
      sigVal = s
      actions.foreach(_())

  def addAction(a: Action): Unit =
    actions = a :: actions
    a()
```

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.

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.

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 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.

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)
```

O ‘orGate‘ and ‘orGate2‘ have the same behavior.

X ‘orGate2‘ does not model OR gates faithfully.

The Simulation Trait

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

Idea: keep in `Simulation` an *agenda* of actions to perform.

The agenda is a list of `Events`. Each event is composed 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:
  ...
  private 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
```

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 =>
```

Implementing Run

An application of the `run` method removes successive elements from the agenda, and performs the associated actions.

This process continues until the agenda is empty:

```
def run(): Unit =
  afterDelay(0) {
    println(s"*** 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 Delays:
  def InverterDelay = 2
  def AndGateDelay = 3
  def OrGateDelay = 5

object sim extends Circuits, Delays
```

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 = 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

Alternative version of OR gate: can be defined in terms of AND and INV.

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

Exercise

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

0 Nothing. The two simulations behave the same.

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

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

0 The two simulations produce different events altogether.

Summary

State and mutable assignment make our mental model of computation more complicated.

In particular, we lose referential transparency.

On the other hand, mutable assignment allows 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.

As always, the choice between functional and imperative programming must be made depending on the situation.