今天的内容

- For包含
- 可变状态对象

For-comprehensions(for 语句包含)

为什么用for comprehension

- Map, flatMap, Filter 抽象性会使得程序难以理解
- 增强可读性
- Build a bridge between set comprehensions in mathematics and for-loops in imperative language
- Resembles query notation of relational databases

表现行式

```
for (p <- persons if p.age > 20) yield p.name
```

```
persons filter (p => p.age > 20) map (p => p.name)
```

- For (s) yield e
  - S 是一系列generators, definitions, filters
  - Generator: val x <- e, e is a list-valued expression; 值绑定
  - Definition: val x = e; 引入一个别名
  - Filter：Boolean-typed expression; 过滤值

举例

- 找到所有质数整数对（i，j），1<=j<i<n，such that i+j is prime

```
for { i <- List.range(1, n)
      j <- List.range(1, i)
      if isPrime(i+j) } yield {i, j}
```

```
sum(for ((x, y) <- xs zip ys) yield x * y)
```

- Compute scalar product of two vectors xs and ys

求解组合问题：N-皇后问题

- Place a queen in each row without attacking other queens
- Assume already generated all solutions of placing k-1 queens

```
def queens(n: Int): List[List[Int]] = {
  def placeQueens(k: Int): List[List[Int]] =
    if (k == 0) List(List())
    else for { queens <- placeQueens(k - 1)
               column <- List.range(1, n + 1)
               if isSafe(column, queens, 1) } yield column :: queens
  placeQueens(n)
}
```

```
def isSafe(col: Int, queens: List[Int], delta: Int): Boolean
```

查询搜索

- Equivalent to common database query languages

```
case class Book(title: String, authors: List[String])
```

A database of books

```
val books: List[Book] = List(
  Book("Structure and Interpretation of Computer Programs",
       List("Abelson, Harold", "Sussman, Gerald J.")),
  Book("Principles of Compiler Design",
       List("Aho, Alfred", "Ullman, Jeffrey")),
  Book("Programming in Modula-2",
       List("Wirth, Niklaus")),
  Book("Introduction to Functional Programming"),
       List("Bird, Richard")),
  Book("The Java Language Specification",
       List("Gosling, James", "Joy, Bill", "Steele, Guy", "Bracha, Gilad")))
```

查询搜索

- To find titles of all books whose author's last name is "Ullman"

```
for (b <- books; a <- b.authors if a startsWith "Ullman")
yield b.title
```

Titles have string "Program"

```
for (b <- books if (b.title indexOf "Program") >= 0)
yield b.title
```

```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

Authors who have written at least two books in the database.

转换翻译

- 可以用高阶函数map, flatMap和filter来实现

```
for (x <- e) yield e'
```

```
e.map(x => e')
```

```
for { i <- range(1, n)
      j <- range(1, i)
      if isPrime(i+j)
} yield {i, j}
```

```
range(1, n)
  .flatMap(i =>
    range(1, i)
      .filter(j => isPrime(i+j))
      .map(j => (i, j)))
```

转换翻译

Map,flatmap, filter也可用for-comprehension来实现

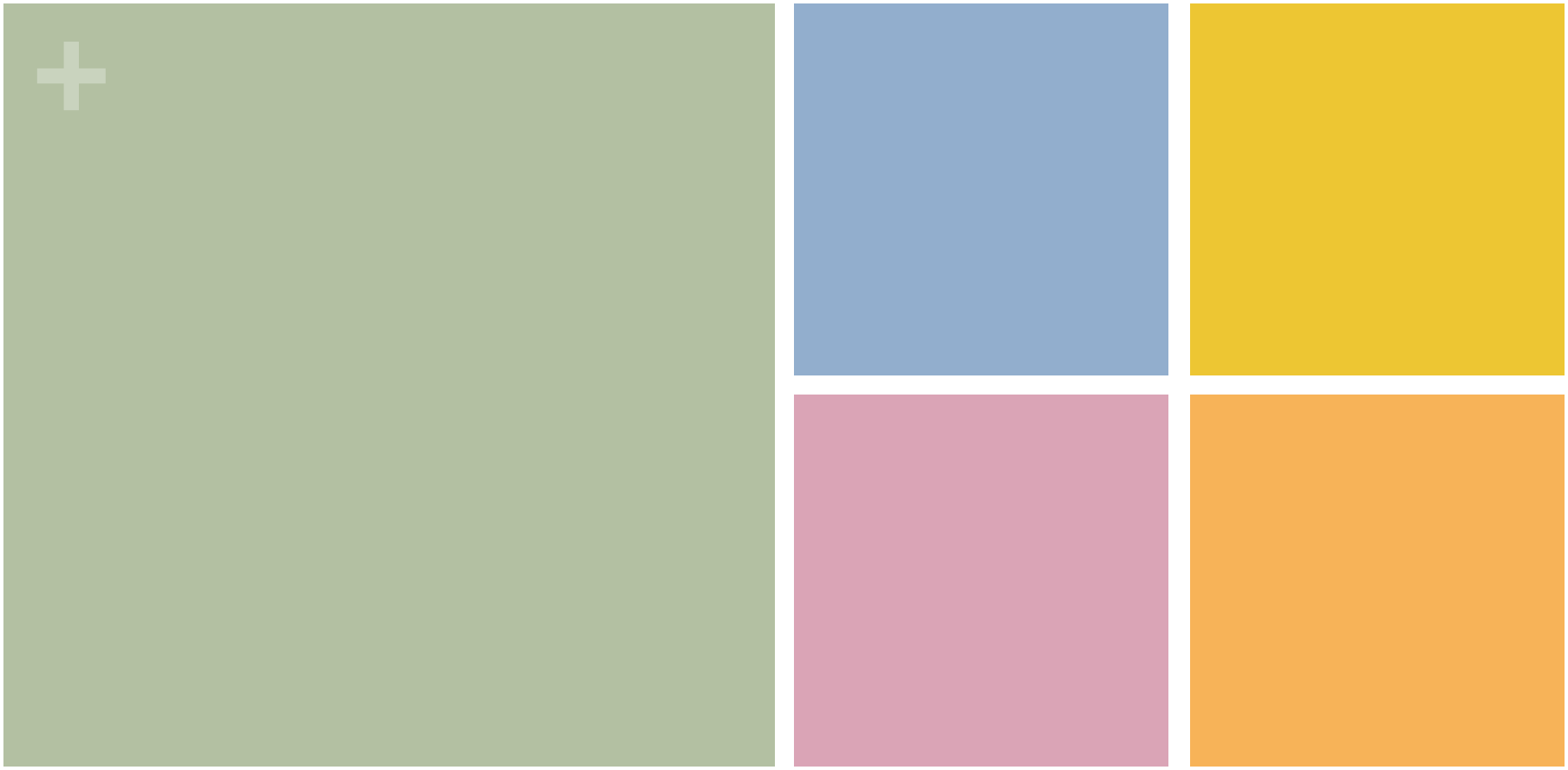
```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

For 循环

- A variant of the for-comprehension syntax
- For (s) e ; key yield is missing

```
for (xs <- xss) {
  for (x <- xs) print(x + "\t")
  println()
}
```

打印显示一个矩阵（列表的列表）的元素



可改变的状态(Mutable State)

Functions and State

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

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:

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

Rewriting Example

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

```
def iterate(n: Int, f: Int => Int, x: Int) =
  if (n == 0) 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) 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) 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) 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.

Example:

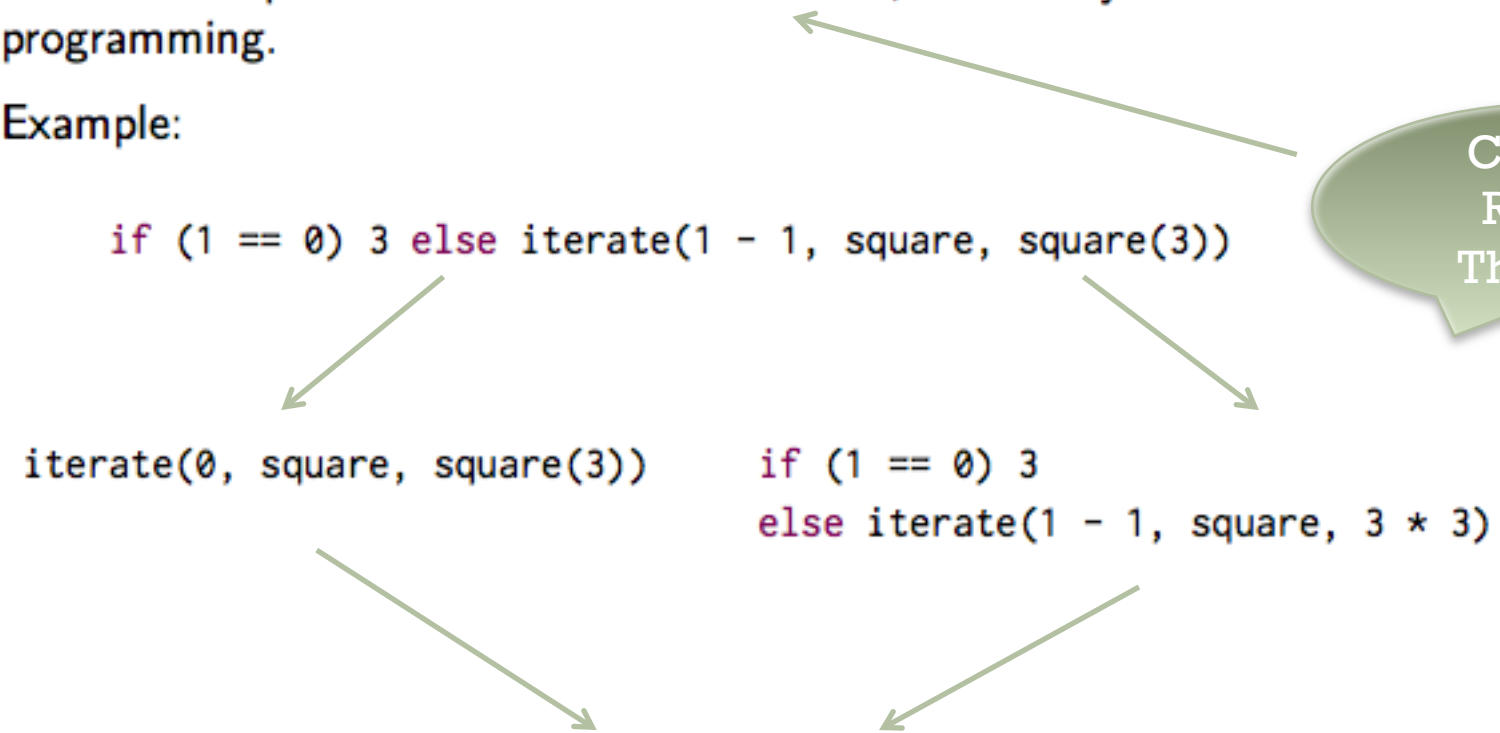
```
if (1 == 0) 3 else iterate(1 - 1, square, square(3))
```

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.

Example:



Church-Rosser Theorem

Stateful Objects(有状态的对象)

- View the world as a set of objects, some have state that changes over time.
- An object has a state if its behavior is influenced by its history.
  - E.g. , A bank account object has state, "can I withdraw ￥100?" depends on different time
  - May vary over the course of the lifetime of the account

Implementation of State

- Every form of mutable state is constructed from variables.

```
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 an *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. **Example**: Here is a class modeling a bank account.

```
class BankAccount {
  private var balance = 0
  def deposit(amount: Int): Unit = {
    if (amount > 0) balance = balance + amount
  }
  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance = balance - amount
      balance
    } else throw new Error("insufficient funds")
}
```

变量代表可能会改变的状态

State in Objects, cont.

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 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 = new BankAccount
```

Working with Mutable Objects

Here is a worksheet that manipulates bank accounts.

```
val account = new BankAccount          // account: BankAccount = BankAccount
account deposit 50                     //
account withdraw 20                    // res1: Int = 30
account withdraw 20                    // res2: 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.

Quiz:

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?

- O Yes
- O No

Identity and Change

Identity and Change

Assignment poses the new problem of deciding whether two expressions are “the same”

When one excludes 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 the assignment, the two formulations are different. For example:

```
val x = new BankAccount
val y = new BankAccount
```

Question: Are x and y the same?

O Yes
O 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 of operations that involves x and y, observing the possible outcomes.

```
val x = new BankAccount
val y = new BankAccount
f(x, y)
```

```
val x = new BankAccount
val y = new BankAccount
f(x, x)
```

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

Counterexample for Operational Equivalence

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

```
val x = new BankAccount
val y = new BankAccount
```

define values x and y that are the same.

Let's follow the definitions by a test sequence:

```
val x = new BankAccount
val y = new BankAccount
```

```
x deposit 30                    // val res1: 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 = new BankAccount
val y = new BankAccount

x deposit 30                    // val res1: Int = 30
x withdraw 20                   // val res2: 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 = new 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.

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 = new BankAccount
val y = x
```

the x in the definition of y could be replaced by new BankAccount

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 = new BankAccount          val x = new BankAccount
val y = x                   ⟶    val y = new BankAccount
```

the x in the definition of y could be replaced by new BankAccount

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

The substitution model ceases to be valid when we add the assignment.

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

有状态对象的相同（sameness）比较

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

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

x，y 相同

```
val x = new BankAccount; val y = new BankAccount
```

这里的x和y相同吗？

- E: arbitrary expression
- 操作结果比较法（operational equivalence）

有状态对象的相同（sameness）比较

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> y withdraw 20
java.lang.RuntimeException: insufficient funds
```

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> x withdraw 20
10
```

- 操作的结果不同，说明**x**和**y**不相同
- 之前的替代计算模型在这里不能被使用

```
val x = new BankAccount; val y = x
```

这样定义则相同。

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) { r = r * x; i = i - 1 }
  r
}
```

In Scala, while is a keyword.

But how could we define while using a function (call it WHILE)?

Definition of while

The function WHILE can be defined as follows:

```
def WHILE(condition: => Boolean)(command: => Unit): Unit =
  if (condition) {
    command
    WHILE(condition)(command)
  }
  else ()
```

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

Note: WHILE 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:

```
REPEAT {
  command
} ( condition )
```

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

The REPEAT function starts like this:

```
def REPEAT(command: => Unit)(condition: => Boolean) =
```

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) { System.out.print(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”) println(i + ” ” + j)
```

translates to:

```
(1 until 3) foreach (i => ”abc” foreach (j => println(i + ” ” + j)))
```

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.

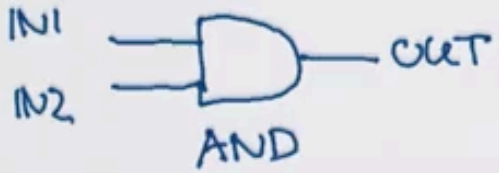
The base components (gates) are:

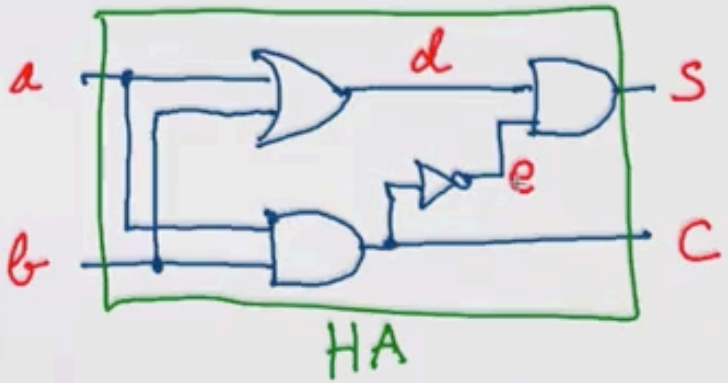
- The *Inverter*, whose output is the inverse of its input.
- The *AND Gate*, whose output is the conjunction of its inputs.
- The *OR Gate*, whose output is the disjunction of its inputs.

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.

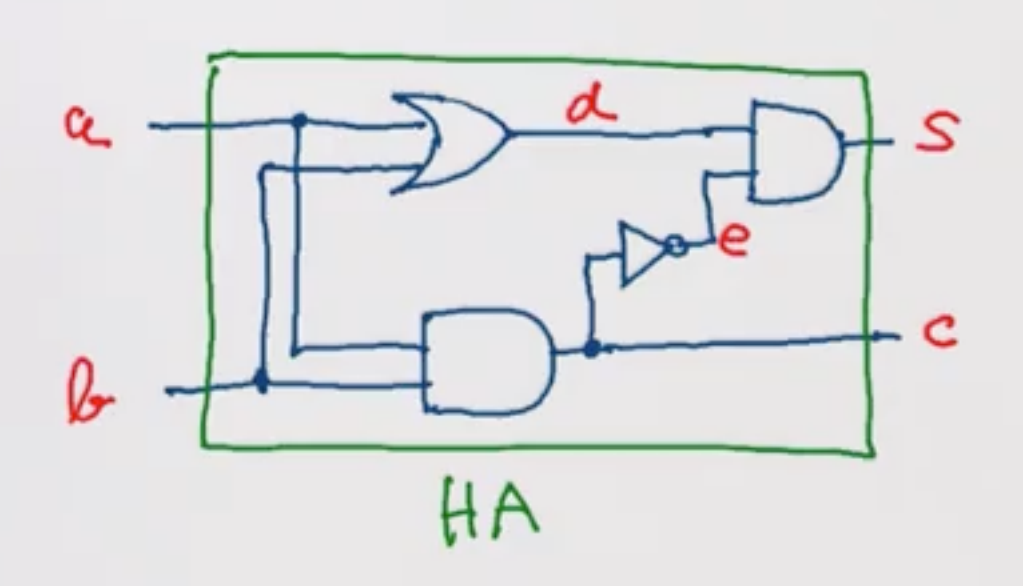
```
def inverter(input: Wire, output: Wire): Unit

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:

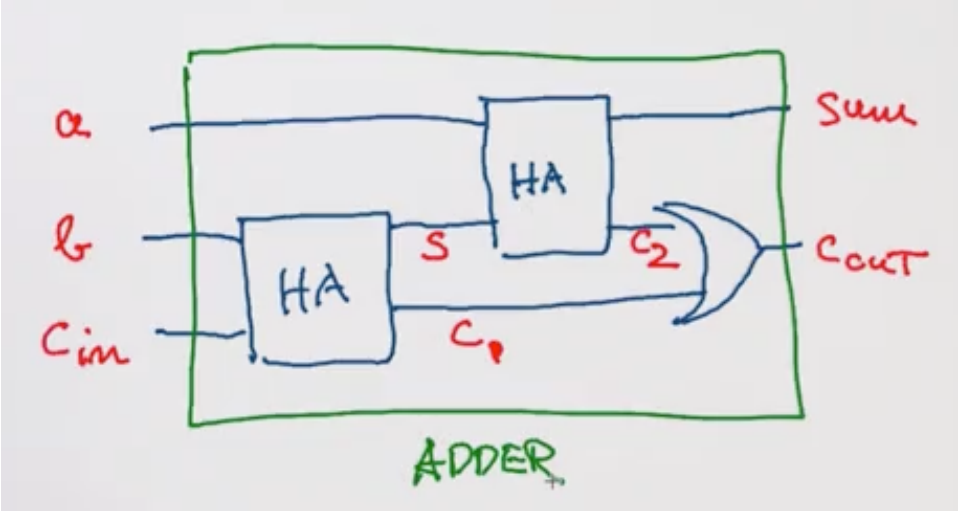
```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire): Unit = {
  val d = new Wire
  val e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)

  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)
}
```

O `a & ~b` O `a & ~(b & a)` O `b & ~a`

O `a == b` O `a != b` O `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
```

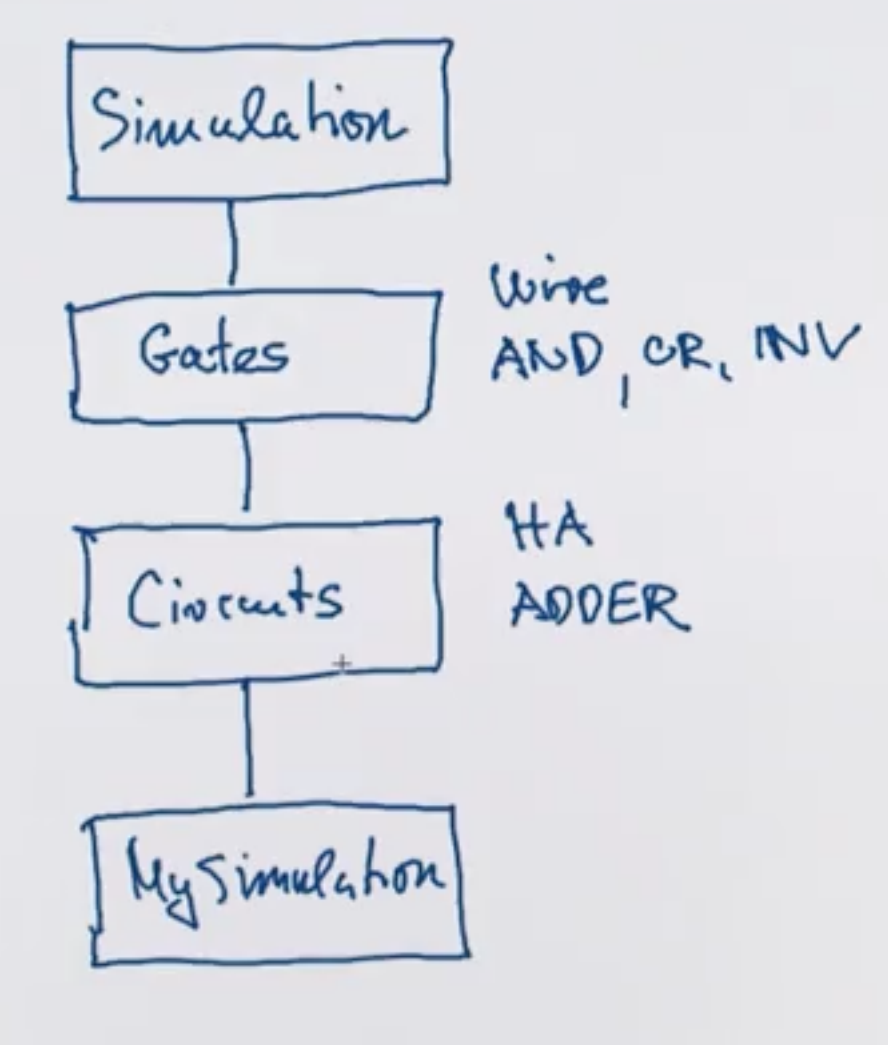
The *time* is simulated; it has nothing to with the actual time.

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
}
```

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

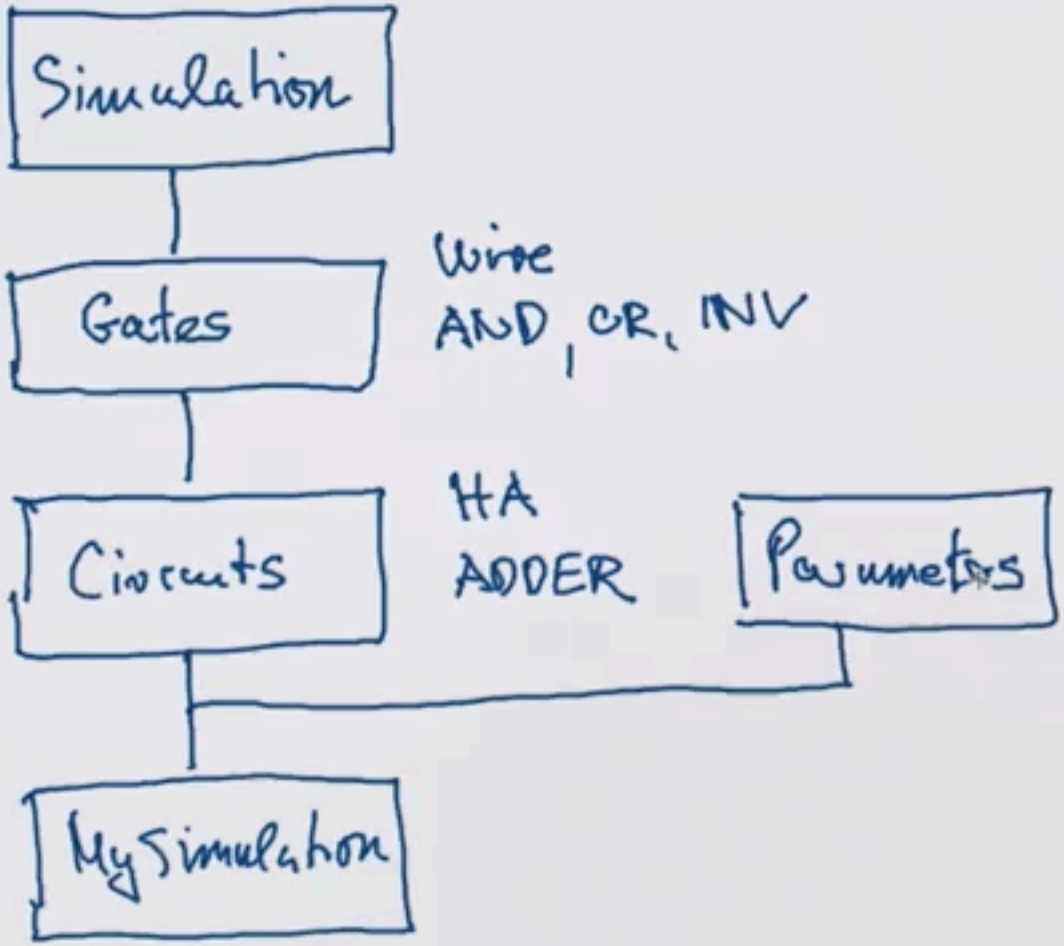
```
def probe(name: String, wire: Wire): Unit = {
  def probeAction(): Unit = {
    println(s"$name $currentTime value = ${wire.getSignal}")
  }
  wire addAction probeAction
}
```

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:

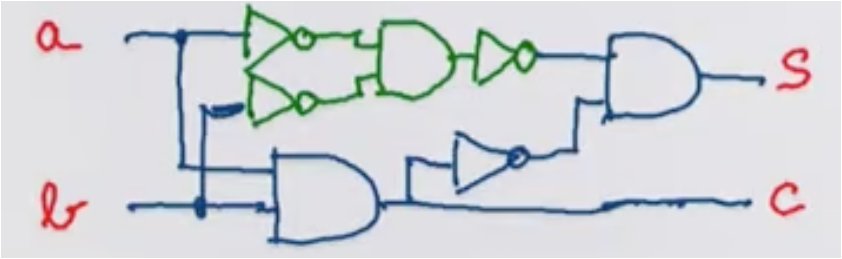
```
input2.setSignal(true)
run()
```

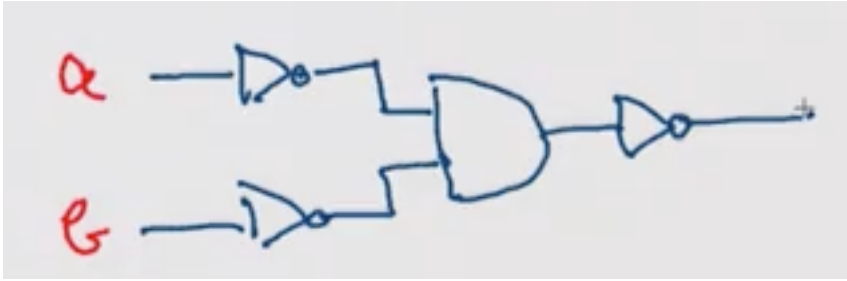
A Variant

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?

O Nothing. The two simulations behave the same.

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

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

O 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.