Queries with For

Queries with `for`

The for notation is essentially equivalent to the common operations of query languages for databases.

Example: Suppose that we have a database `books`, represented as a list of books.

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

A Mini-Database

```
val books: List[Book] = List(
  Book(title   = "Structure and Interpretation of Computer Programs",
       authors = List("Abelson, Harald", "Sussman, Gerald J.")),
  Book(title   = "Introduction to Functional Programming",
       authors = List("Bird, Richard", "Wadler, Phil")),
  Book(title   = "Effective Java",
       authors = List("Bloch, Joshua")),
  Book(title   = "Java Puzzlers",
       authors = List("Bloch, Joshua", "Gafter, Neal")),
  Book(title   = "Programming in Scala",
       authors = List("Odersky, Martin", "Spoon, Lex", "Venners, Bill")))
```

Some Queries

To find the titles of books whose author's name is "Bird":

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

To find all the books which have the word "Program" in the title:

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

Another Query

To find the names of all authors who have written at least two books present in the database.

```
for {
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
} yield a1
```

Another Query

To find the names of all authors who have written at least two books present in the database.

```
for {
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
} yield a1
```

Why do solutions show up twice?

How can we avoid this?

Modified Query

To find the names of all authors who have written at least two books present in the database.

```
for {
  b1 <- books
  b2 <- books
  if b1.title < b2.title
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
} yield a1
```

Problem

What happens if an author has published three books?

```
0    The author is printed once
0    The author is printed twice
0    The author is printed three times
0    The author is not printed at all
```

Problem

What happens if an author has published three books?

```
0    The author is printed once
0    The author is printed twice
0    The author is printed three times
0    The author is not printed at all
```

Modified Query (2)

Solution: We must remove duplicate authors who are in the results list twice.

This is achieved using the `distinct` method on sequences:

```
{ for {
    b1 <- books
    b2 <- books
    if b1.title < b2.title
    a1 <- b1.authors
    a2 <- b2.authors
    if a1 == a2
  } yield a1
}.distinct
```

Modified Query

Better alternative: Compute with sets instead of sequences:

```
val bookSet = books.toSet
for {
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
} yield a1
```

Translation of For

For-Expressions and Higher-Order Functions

The syntax of `for` is closely related to the higher-order functions `map`, `flatMap` and `filter`.

First of all, these functions can all be defined in terms of `for`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  for (x <- xs) yield f(x)

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

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

Translation of For (1)

In reality, the Scala compiler expresses for-expressions in terms of `map`, `flatMap` and a lazy variant of `filter`.

Here is the translation scheme used by the compiler (we limit ourselves here to simple variables in generators)

1. A simple for-expression

```
for (x <- e1) yield e2
```

is translated to

```
e1.map(x => e2)
```

Translation of For (2)

2. A for-expression

```
for (x <- e1 if f; s) yield e2
```

where `f` is a filter and `s` is a (potentially empty) sequence of generators and filters, is translated to

```
for (x <- e1.withFilter(x => f); s) yield e2
```

(and the translation continues with the new expression)

You can think of `withFilter` as a variant of `filter` that does not produce an intermediate list, but instead filters the following `map` or `flatMap` function application.

Translation of For (3)

3. A for-expression

```
for (x <- e1; y <- e2; s) yield e3
```

where `s` is a (potentially empty) sequence of generators and filters,
is translated into

```
e1.flatMap(x => for (y <- e2; s) yield e3)
```

(and the translation continues with the new expression)

Example

Take the for-expression that computed pairs whose sum is prime:

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

Applying the translation scheme to this expression gives:

```
(1 until n).flatMap(i =>
  (1 until i).withFilter(j => isPrime(i+j))
   .map(j => (i, j)))
```

This is almost exactly the expression which we came up with first!

Exercise

Translate

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

into higher-order functions.

Exercise

Translate

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

into higher-order functions.

Generalization of `for`

Interestingly, the translation of for is not limited to lists or sequences, or even collections;

It is based solely on the presence of the methods `map`, `flatMap` and `withFilter`.

This lets you use the for syntax for your own types as well – you must only define `map`, `flatMap` and `withFilter` for these types.

There are many types for which this is useful: arrays, iterators, databases, XML data, optional values, parsers, etc.

For and Databases

For example, `books` might not be a list, but a database stored on some server.

As long as the client interface to the database defines the methods `map`, `flatMap` and `withFilter`, we can use the `for` syntax for querying the database.

This is the basis of the Scala data base connection frameworks ScalaQuery and Slick.

Similar ideas underly Microsoft's LINQ.

More On For-Expressions

Recap: Collections

Core classes

```
Iterable--+--Seq--+--List
          |       +--Stream
          |       +--Vector
          |       +--Range
          |       +~~Array
          |       +~~String
          |
          +--Set--+--HashSet
          |       +--TreeSet
          |
          +--Map--+--HashMap
                  +--TreeMap
```

Recap: Collection Methods

Core methods:

```
map
flatMap
filter
```

and also

```
foldLeft
foldRight
```

For-Expressions

Simplify combinations of core methods `map`, `flatMap`, `filter`.

Instead of:

```
(1 until n) flatMap (i =>
   (1 until i) map (j => (i, j))) filter ( pair =>
      isPrime(pair._1 + pair._2))
```

one can write:

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

Other Uses of For-Expressions

Operations of sets, or databases, or options.

Question: Are for-expressions tied to collections?

Answer: No! All that is required is some interpretation of `map`, `flatMap` and `withFilter`.

There are many domains outside collections that afford such an interpretation.

Tow examples: Random values and futures.

Random Values

You know about random numbers:

```
import java.util.Random
val rand = new Random
rand.nextInt
```

Question: What is a systematic way to get random values for other domains:

```
booleans
strings
pairs and tuples
lists
sets
```

?

Generators

Let's define a class Generator[T] that can generate random values of type T:

```
trait Generator[+T] {
  def generate: T
}
```

Some instances:

```
val integers = new Generator[Int] {
  def generate = scala.util.Random.nextInt()
}
val booleans = new Generator[Boolean] {
  def generate = integers.generate >= 0
}
val pairs = new Generator[(Int, Int)] {
  def generate = (integers.generate, integers.generate)
```

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, would like to write:

```
val pairs = for {
  x <- integers
  y <- integers
} yield (x, y)
```

Need `map` and `flatMap` for that!

Generator with Map and FlatMap

Here's a more convenient version of `Generator`:

```
trait Generator[+T] {
  self =>       // an alias for ”this”.

  def generate: T

  def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S] {
    def generate = f(self.generate).generate
  }

  def map[S](f: T => S): Generator[S] = new Generator[S] {
    def generate = f(self.generate)
  }
}
```

Some Generators

```
implicit def integers: Generator[Int] = new Generator[Int] {
  def generate = scala.util.Random.nextInt()
}

implicit def choose(lo: Int, hi: Int): Generator[Int] = new Generator[Int] {
  def generate = scala.util.Random.nextInt(hi - lo) + lo
}

implicit def single[T](x: T): Generator[T] = new Generator[T] {
  def generate = x
}
```

More Generators

```
implicit def booleans: Generator[Boolean] = integers.map(_ >= 0)

implicit def pairs[T, U](implicit t: Generator[T], u: Generator[U]): Generator[(T, U)
  x <- t
  y <- u
} yield (x, y)
```

Application: Random Testing

You know about units tests:

- Come up with some some test inputs to program functions and a *postcondition*.
- The postcondition is a property if the expected result.
- Verify that the program satisfies the postcondition.

Question: Can we do without the test inputs?

Yes, by generating random test inputs

Random Test Function

Using generators, we can write a random test function:

```
def test[T](g: Generator[T], numTimes: Int = 100)
    (test: T => Boolean): Unit = {
  for (i <- 0 until numTimes) {
    val value = g.generate
    assert(test(value), "test failed for "+value)
  }
  println("passed "+numTimes+" tests")
}
```

Example usage:

```
test(lists[Int]) {(xs: List[Int]) =>
  xs.reverse == xs
}
```

ScalaCheck

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

This idea is implemented in the `ScalaCheck` tool.

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

See ScalaCheck tutorial on the course page.

Asynchronous Processing

Programs are often *asynchronous*: Several tasks, some results need waiting.

Examples:

- I/O
- Webservices
- Inter-process communication

Want to avoid blocking waits.

```
SlowService(request).get()
  // System hangs until SlowService has finished
```

Futures

A `Future` represents a value that will be computed in the future.

First version:

```
class Future[+T] {
  def get: T
}
```

If `SlowService` returns a future, we can now do something useful in the meantime:

```
val myFuture = MySlowService(request) // returns right away
...do other things...
val result = myFuture.get() // blocks until service ”fills in” myFuture
```

Asynchronous Use of Futures

Problem: Once we call `get`, we still block!

Would like to use a *call-back*, be notified when future is ready.

Here's how this works:

```
val future = MySlowService(request)
future onSuccess { reply =>
  // when the future gets ”filled”, use its value
  println(reply)
}
```

This assumes an `onSuccess` operation in class `Future`:

```
def onSuccess[U](cont: T => U): U
```

Downside of Callbacks

Problem with too many callbacks: spaghetti-code.

Would like to write code like:

```
val user = getUserById(id)
val orders = getOrdersForUser(user.email)
val products = getProductsForOrders(orders)
val stock = getStockForProducts(products)
```

But have it work asynchronously out of the box.

Composition of Futures

We can do better with (you guessed it!) for expressions.

```
for {
  user <- getUserById(id)
  orders <- getOrdersForUser(user.email)
  products <- getProductsForOrders(orders)
  stock <- getStockForProducts(products)
} yield stock
```

To make this work, futures need `map` and `flatMap` operations.

Outline of Class Future

```scala
class Future[+T] { self =>
  def get: T
  def onSuccess[U](cont: T => U): U
  def map[U](f: T => U): Future[U] =
  def flatMap[U](f: T => Future[U]): Future[U]
}
```

Monads

Data structures with `map` and `flatMap` seem to be quite common.

In fact there's a name that describes this class of a data structures together with some algebraic laws that they should have.

They are called *monads*.

Monads are very popular in the Haskell programming language.

What is a Monad?

A monad `M` is a parametric type M[T] with two operations, `flatMap` and `unit`, that have to satisfy some laws.

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

In the literature, `flatMap` is more commonly called `bind`.

Examples of Monads

- `List` is a monad with `unit(x) = List(x)`
- `Set` is monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`
- `Generator` is a monad with `unit(x) = single(x)`

`flatMap` is an operation on each of these types, whereas `unit` in Scala is different for each monad.

Monads and map

`map` can be defined for every monad as a combination of `flatMap` and `unit`:

```
m map f  ==  m flatMap (x => unit(f(x)))
         ==  m flatMap (f andThen unit)
```

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m flatMap f flatMap g  ==  m flatMap (x => f(x) flatMap g)
```

Left unit

```
unit(x) flatMap f  ==  f(x)
```

Right unit

```
m flatMap unit  ==  m
```

Checking Monad Laws

Let's check the monad laws for Option.

Here's `flatMap` for `Option`:

```
abstract class Option[+T] {

  def flatMap[U](f: T => Option[U]): Option[U] = this match {
    case Some(x) => f(x)
    case None => None
  }
}
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
Some(x) flatMap f
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
      Some(x) flatMap f

==    Some(x) match {
        case Some(x) => f(x)
        case None => None
      }
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
      Some(x) flatMap f

==    Some(x) match {
        case Some(x) => f(x)
        case None => None
      }

==    f(x)
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
opt flatMap Some
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
      opt flatMap Some

==    opt match {
        case Some(x) => Some(x)
        case None => None
      }
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
      opt flatMap Some

==    opt match {
        case Some(x) => Some(x)
        case None => None
      }

==    opt
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
opt flatMap f flatMap g
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
       opt flatMap f flatMap g

==     opt match { case Some(x) => f(x) case None => None }
           match { case Some(y) => g(y) case None => None }
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
       opt flatMap f flatMap g

  ==   opt match { case Some(x) => f(x) case None => None }
           match { case Some(y) => g(y) case None => None }

  ==   opt match {
         case Some(x) =>
           f(x) match { case Some(y) => g(y) case None => None }
         case None =>
           None match { case Some(y) => g(y) case None => None }
       }
```

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }
```

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }

==    opt match {
        case Some(x) => f(x) flatMap g
        case None => None
      }
```

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }

==    opt match {
        case Some(x) => f(x) flatMap g
        case None => None
      }

==    opt flatMap (x => f(x) flatMap g)
```

Significance of the Laws for For-Expressions

We have seen that monad-typed expressions are typically written as `for` expressions.

What is the significance of the laws with respect to this?

1. Associativity says essentially that one can "inline" nested for expressions:

```
      for (y <- for (x <- m; y <- f(x)) yield y
           z <- g(y)) yield z

==    for (x <- m;
           y <- f(x)
           z <- g(y)) yield z
```

Significance of the Laws for For-Expressions

2. Right unit says:

```
      for (x <- m) yield x
==    m
```

3. Left unit does not have an analogue for for-expressions.

Another type: Try

In the later parts of this course we will need a type named `Try`.

`Try` resembles `Option`, but instead of `Some`/`None` there is a `Success` case with a value and a `Failure` case that contains an exception:

```
abstract class Try[+T]
case class Success[T](x: T)       extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]
```

`Try` is used to pass results of computations that can fail with an exception between threads and computers.

Creating a Try

You can wrap up an arbitrary computation in a `Try`.

```
Try(expr)     // gives Success(someValue) or Failure(someException)
```

Here's an implementation of `Try`:

```
object Try {
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch {
      case NonFatal(ex) => Failure(ex)
    }
```

Composing `Try`

Just like with `Option`, `Try`-valued computations can be composed in for expresssions.

```
for {
  x <- computeX
  y <- computeY
} yield f(x, y)
```

If `computeX` and `computeY` succeed with results `Success(x)` and `Success(y)`, this will return `Success(f(x, y))`.

If either computation fails with an exception `ex`, this will return `Failure(ex)`.

Definition of `flatMap` and `map` on `Try`

```
abstract class Try[T] {
  def flatMap[U](f: T => Try[U]): Try[U] = this match {
    case Success(x) => try f(x) catch { case NonFatal(ex) => Failure(ex) }
    case fail: Failure => fail
  }

  def map[U](f: T => U): Try[U] = this match {
    case Success(x) => Try(f(x))
    case fail: Failure => fail
  }}
```

So, for a `Try` value `t`,

```
t map f  ==  t flatMap (x => Try(f(x)))
         ==  t flatMap (f andThen Try)
```

Exercise

It looks like `Try` might be a monad, with `unit` = `Try`.

Is it?

```
O     Yes
O     No, the associative law fails
O     No, the left unit law fails
O     No, the right unit law fails
O     No, two or more monad laws fail.
```

Solution

It turns out the left unit law fails.

```
Try(expr) flatMap f  !=  f(expr)
```

Indeed the left-hand side will never raise a non-fatal exception whereas the right-hand side will raise any exception thrown by `expr` or `f`.

Hence, `Try` trades one monad law for another law which is more useful in this context:

> *An expression composed from 'Try', 'map', 'flatMap' will never throw a non-fatal exception.*

Call this the "bullet-proof" principle.

Conclusion

We have seen that for-expressions are useful not only for collections.

Many other types also define `map`,`flatMap`, and `withFilter` operations and with them for-expressions.

Examples: `Generator`, `Option`, `Try`.

Many of the types defining `flatMap` are monads.

(If they also define `withFilter`, they are called "monads with zero").

The three monad laws give useful guidance in the design of library APIs.