Strictness and Laziness

Laziness can be used to improve the **efficiency** and **modularity** of functional programs

Topics to Cover

- Strictness vs non-strictness
- Lazy lists
- Separating program description from evaluation

How to solve this problem?

Imagine if you had a deck of cards and
you were asked to
remove the odd-numbered cards and flip over all the queens.

Ideally, you'd make **a single pass** through the deck, looking for queens and odd numbered cards **at the same time**.

Program trace for List

```
List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)

⇒ List(11,12,13,14).filter(_ % 2 == 0).map(_ * 3)

⇒ List(12,14).map(_ * 3)

=> List(36,42)
```

Wouldn't it be nice if we could somehow **fuse sequences of transformations** like this **into a single pass** and **avoid creating temporary** data structures?

Strict and Non-strict Functions

- A ***strict*** function always evaluates its arguments.
  - Strict functions are the norm in most programming languages including Scala.

- A ***non-strict*** function may choose ***not*** to evaluate one or more of its arguments.

Non-Strictness in Scala

Short-circuiting Boolean functions **&&** and **||** are non-strict.

Another example of non-strictness is the **if-expression** in Scala:

```
val result = if a > b then f(a) else g(b)
```

[if-function is **strict in its condition parameter**, since it'll always evaluate the condition to determine which branch to take, and **non-strict in the two branches** for the true and false cases, since it'll only evaluate one or the other based on the condition.]

Thunk in FP

- A ***zero-argument*** anonymous function **f : () => A** that acts as an unevaluated form of argument expression to support the effect of **call-by-name**.

```
def if2[A](cond: Boolean, onTrue: () => A, onFalse: () => A): A =
  if cond then onTrue() else onFalse()

if2(a < 22,
  () => println("a"),
  () => println("b")
)
```

The function literal syntax for creating a `() => A`

Call By Value vs. Call By Name

Call by Value

- Calculates all of the arguments before the call and then passes the resulting values to the function.
- *Strict evaluation*

```
square(41.0 + 1.0) => square(42.0)
square(sys.error("failure")) => Boom!
```

Call by Name

- The function receives the unevaluated argument expression and evaluate it if necessary.
- *Lazy evaluation*

By-Name Parameters in Scala

```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
      if (cond) then onTrue else onFalse
```

To evaluate the argument, just reference the identifier as usual

When calling this function, Just use normal function call syntax

```
scala> if2(false, sys.error("fail"), 3)
res2: Int = 3
```


By-Name Parameters in Scala

An argument that's passed unevaluated to a function will be evaluated ***once for each reference*** in the function body.

Scala won't (by default) cache the result of evaluating an argument:

```
scala> def maybeTwice(b: Boolean, i: => Int) = if (b) i+i else 0
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice(true, { println("hi"); 1+41 })
hi
hi
x: Int = 84
```

By-Name Parameters in Scala

An argument that's passed unevaluated to a function will be evaluated ***once for each reference*** in the function body.

Scala won't (by default) cache the result of evaluating an argument:

```
scala> def maybeTwice(b: Boolean, i: => Int) = if b then i + i else 0
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice(true, { println("hi"); 1 + 41 })
hi
hi
x: Int = 84
```


Lazy Val

Adding the **lazy** keyword to a **val** declaration will ***delay*** evaluation until it's first referenced. It will also ***cache* the result** to prevent repeated evaluation.

```
scala> def maybeTwice2(b: Boolean, i: => Int) =
| lazy val j = i
| if b then j + j else 0
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice2(true, { println("hi"); 1 + 41 })
hi
x: Int = 84
```

Formal definition of strictness

- If the evaluation of an expression runs forever or throws an error instead of returning a definite value, we say that the expression doesn't terminate, or that it evaluates to ***bottom***.
- A function f is ***strict*** if the expression $f(x)$ evaluates to bottom for all x that evaluate to bottom.

LazyList

```
enum LazyList[+A]:
  case Empty
  case Cons(h: () => A, t: () => LazyList[A])
```

A nonempty stream consists of a head and a tail, which are both non-strict. Due to technical limitations, these are thunks that must be explicitly forced, rather than by-name parameters.

```
sealed trait Stream[+A]
case object Empty extends Stream[Nothing]
case class Cons[+A](h: () => A, t: () => Stream[A]) extends Stream[A]
```

LazyList Companion Object

```
object LazyList:
  // Smart constructors: cons and empty
  def cons[A](hd: => A, tl: => LazyList[A]): LazyList[A] =
    lazy val head = hd // memoize
    lazy val tail = tl
    Cons(() => head, () => tail)

  def empty[A]: LazyList[A] = Empty

  def apply[A](as: A*): LazyList[A] =
    if as.isEmpty then empty
    else cons(as.head, apply(as.tail*))
```

We cache the head and tail as lazy values to avoid repeated evaluation.

A convenient variable-argument method for constructing a Stream from multiple elements.


```
def headOption: Option[A] = this match
  case Empty => None
  case Cons(h, _) => Some(h())
```

Explicit forcing of the h thunk using h()

- This ability of `LazyList` to evaluate only the portion actually demanded (we don't evaluate the tail of the `Cons`) is useful.

Memoizing & *Avoiding Recomputation*

- ***Smart* constructors**
  - Ensures some additional invariant or
  - Provides a slightly different signature than the "real" constructors used for pattern matching.
  - By convention, smart constructors typically lowercase the first letter of the corresponding data constructor.

```
def cons[A](hd: => A, tl: => LazyList[A]): LazyList[A] = {
      lazy val head = hd
      lazy val tail = tl
      Cons(() => head, () => tail)
}
```

cons smart constructor takes care of memoizing the by-name arguments for the head and tail of the Cons.

Avoiding Recomputation

- If we use Cons directly, this code will actually compute **expensive(x)** twice.

```
val x = Cons(() => expensive(y), tl)
val h1 = x.headOption
val h2 = x.headOption
```

```
def headOption: Option[A] = this match
  case Empty => None
  case Cons(h, _) => Some(h())
```

- The **empty** smart constructor just returns **Empty**, but annotates **Empty** as a **LazyList[A]**.

```
def empty[A]: LazyList[A] = Empty
```

- Scala takes care of wrapping the arguments to **cons** in thunks, so the **as.head** and **apply(as.tail*)** expressions won't be evaluated until we force the **LazyList**.

```
def apply[A](as: A*): LazyList[A] = {
  if (as.isEmpty) then Empty
  else cons(as.head, apply(as.tail*))
}
```

Beware that `as` itself is strict!

Implement a helper Function and some useful functions to inspect the LazyList

```
// Convert LazyList to List
def toList: List[A]
// Take the first n elements of a LazyList
def take(n: Int): LazyList[A]
// Skip the first n elements of a LazyList
def drop(n: Int): LazyList[A]
// Take the first elms of a LazyList that match the given predicate
def takeWhile(p: A => Boolean): LazyList[A]
// Skip the first elms of a LazyList that match the given predicate
def dropWhile(p: A => Boolean): LazyList[A]
```

Separating Program Description from Evaluation

Separation of concerns: to separate the description of computations from actually running them.

Laziness lets us separate the description of an expression from the evaluation of that expression.

➔ Can describe a "larger" expression, and then evaluate only a portion of it.

```
def exists(p: A => Boolean): Boolean = this match
  case Cons(h, t) => p(h()) || t().exists(p)
  case _ => false
```

Remember also that the tail of the lazy list is a lazy val. So not only does the traversal terminate early, the tail of the lazy list is never evaluated at all!

Lazy foldRight for LazyLists

If f doesn't evaluate its second argument, the recursion never occurs.

```
def foldRight[B](z: => B)(f: (A, => B) => B): B =
  this match
    case Cons(h, t) => f(h(), t().foldRight(z)(f))
    case Empty => z
```

The arrow => in front of the argument type B means that the function f takes its second argument by name and may choose not to evaluate it.

We can see this by using **foldRight** to implement exists:

```
def exists(p: A => Boolean): Boolean =
  foldRight(false)((a, b) => p(a) || b)
```

Exercises

- **[Exercise 5.4]** Implement `forAll`, which checks that all elements in the `LazyList` match a given predicate. Your implementation should terminate the traversal as soon as it encounters a nonmatching value.

```
def forAll(p: A => Boolean): Boolean = ???
```

- **[Exercise 5.5]** Use `foldRight` to implement `takeWhile`.
- **[Exercise 5.6]** Implement `headOption` using `foldRight`.
- **[Exercise 5.7]** Implement `map`, `filter`, `append`, and `flatMap` using `foldRight`. The `append` method should be non-strict in its argument.

Program Trace for LazyList

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, LazyList(4).map(_ + 10)).filter(_ % 2 ==
0).toList
12 :: LazyList(4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(14, LazyList().map(_ + 10)).filter(_ % 2 == 0).toList
12 :: 14 :: LazyLis
12 :: 14 :: List()
```

Apply `map` to the first element.

Apply `filter` to the first element.

Apply `map` to the second element.

Apply `filter` to the second element. Produce the first element of the result.

Apply `filter` to the fourth element and produce the final element of the result.

`map` and `filter` have no more work to do, and the empty stream becomes the empty list.

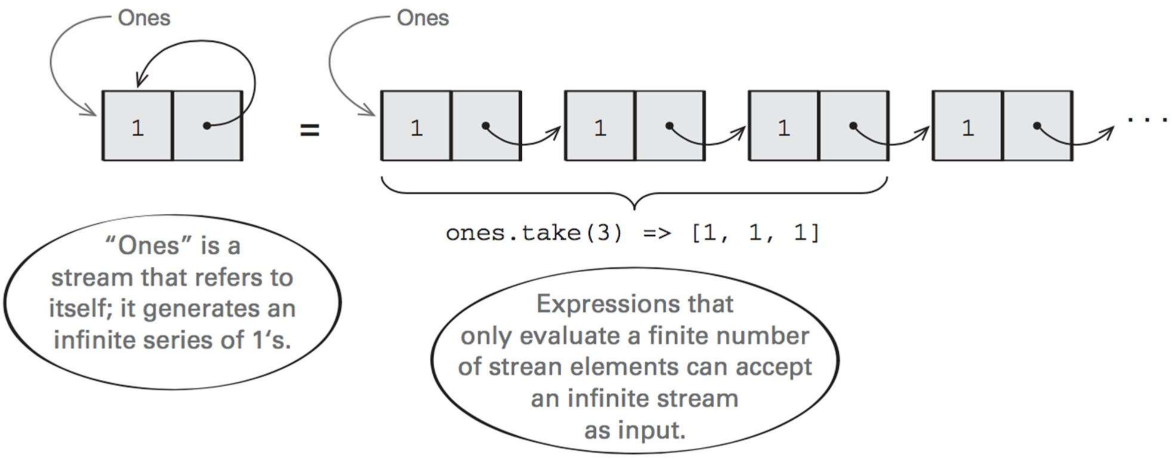
What We Observed - Incremental nature of lazy list transformations

- The thing to notice in this trace is how the filter and map transformations are interleaved.
- Note that we don't fully instantiate the intermediate lazy list that results from the map.
- Since intermediate lazy lists aren't instantiated, it's easy to reuse existing operations in novel ways without having to worry about needless processing.

```
def find(p: A => Boolean): Option[A] =
  filter(p).headOption
```


Infinite LazyLists

```
val ones: LazyList[Int] = LazyList.cons(1, ones)
```



Many functions can be evaluated using finite resources even if their inputs generate infinite sequences.

```
val ones: LazyList[Int] = LazyList.cons(1, ones)
```

Although ones is infinite, the functions we've written so far only inspect the portion of the stream needed to generate the demanded output. For example:

```
scala> ones.take(5).toList
res0: List[Int] = List(1, 1, 1, 1, 1)

scala> ones.exists(_ % 2 != 0)
res1: Boolean = true
```

Try playing with a few other examples:

- `ones.map(_ + 1).exists(_ % 2 == 0)`
- `ones.takeWhile(_ == 1)`
- `ones.forAll(_ != 1)`

How about?

`ones.forAll(_ == 1)`

Exercises

- **[Exercise 5.8]** Generalize ones slightly to the function continually, which returns an infinite `LazyList` of a given value.

  ```
  def continually[A](a: A): LazyList[A]
  ```

- **[Exercise 5.9]** Write a function that generates an infinite lazy list of integers, starting from n, then n + 1, n + 2, and so on.

  ```
  def from(n: Int): LazyList[Int]
  ```

- **[Exercise 5.10]** Write a function `fibs` that generates the infinite lazy list of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, and so on.

[Exercise 5.11] Unfold: Corecursive Function

- A ***corecursive function*** produces data and need not terminate so long as they remain productive.
- The `unfold` is a very general LazyList-building function.
- `unfold` takes an initial state, and a function for producing both the next state and the next value.

```
def unfold[A,S](state: S)(f: S => Option[(A,S)]): LazyList[A]
```

- `Option` is used to indicate when the `LazyList` should be terminated, if at all.

More Exercises

- **[Exercise 5.12]** Write `fibs`, `from`, `continually`, and `ones` in terms of `unfold`.

- **[Exercise 5.13]** Use `unfold` to implement `map`, `take`, `takeWhile`, `zipWith` (as in chapter 3), and `zipAll`. The `zipAll` function should continue the traversal as long as either lazy list has more elements—it uses `Option` to indicate whether each lazy list has been exhausted.

```
def zipAll[B](that: LazyList[B]): LazyList[(Option[A], Option[B])]
```

Visit “hasSequence” again

- Implement hasSequence in terms of startswith and tails:

```
def hasSubsequence[A](l: LazyList[A]): Boolean =
  tails.exists(_.startsWith(l))

def startsWith[A](prefix: LazyList[A]): Boolean = ???

def tails: LazyList[LazyList[A]] = ???
```

// LazyList(1, 2, 3).tails = LazyList(LazyList(1, 2, 3), LazyList(2, 3), LazyList(3), LazyList())