Functional Programming

Agenda

- Call-by-name (vs. call-by-value)
- Essential collections: (immutable) lists, sets, maps, and options
- Parametric polymorphism (called generics in OO terminology)
- Iteration with accumulators (vs. using mutable state)
- Control abstractions and data processing with higher-order functions
- Making the MiniScala implementation more "functional"
  - Avoiding mutable state
  - Implementing recursive environments using higher-order functions

the only new language features for this week
(not added to MiniScala)

Call-by-name – motivating example

Assume we want to implement a function `log` that writes a string to the console, but only if a global flag is set

```
def log(s: String) =
  if (loggingEnabled)
    println(s)
```

Works fine, but may waste computing time if the flag is not set:

```
log("A big number: " + ack(5,5))
```

(Recall the call-by-value mechanism: argument expressions are evaluated before the function body)

Call-by-name – another motivating example

We have seen that booleans could have been defined as an inductive data type instead of being hardwired into the language

Could we then implement if-then-else as a function?
(for simplicity, let's just make it work for integer expressions)

```
sealed abstract class Boolean
case object True extends Boolean
case object False extends Boolean
```

Naive attempt:

```
def ifThenElse(cond: Boolean, thenexp: Int, elseexp: Int) =
  cond match {
    case True => thenexp
    case False => elseexp
  }
```

Example:

```
def gcd(p: Int, q: Int): Int =
  ifThenElse(q == 0, {
    p
  },{
    val t = p % q
    gcd(q, t)
  })
```

It sort-of works…

… both branches are always evaluated ☹
(because of call-by-value)

Call-by-name

Warning: depending on the context, “=>” means either
- a lambda definition,
- a function type,
- a delimiter in `match-case`, or
- a by-name parameter

Using by-name parameters in Scala:

```
def log(s: => String) =
  if (loggingEnabled)
    println(s)
```

```
log("A big number: " + ack(5,5))
```

Same effect, using ordinary call-by-value:

```
def log(s: () => String) =
  if (loggingEnabled)
    println(s())
```

```
log(() => "A big number: " + ack(5,5))
```

a “thunk”

With call-by-name, the argument is not evaluated until the parameter is used

How it works: instead of evaluating the argument and binding the value to the parameter, we bind a *closure* to the parameter
– like wrapping the argument into a zero-arguments lambda before entering the function

(Try it in Interpreter.trace in your MiniScala implementation!)

Call-by-name

`ifThenElse` using Scala's by-name parameters:

```
def ifThenElse(cond: Boolean, thenexp: => Int, elseexp: => Int) =
  cond match {
    case True => thenexp
    case False => elseexp
  }
```

```
def gcd(p: Int, q: Int): Int =
  ifThenElse(q == 0, {
    p
  },{
    val t = p % q
    gcd(q, t)
  })
```

Control structures don't have to be hardwired into the programming language!

Call-by-name (+ an extra Scala trick)

`ifThenElse` using Scala's by-name parameters:

(and multiple-parameter curly-bracket arguments, which is a form of currying supported by Scala)

```
def ifThenElse(cond: Boolean)(thenexp: => Int)(elseexp: => Int) =
  cond match {
    case True => thenexp
    case False => elseexp
  }
```

```
def gcd(p: Int, q: Int): Int =
  ifThenElse(q == 0) {
    p
  } {
    val t = p % q
    gcd(q, t)
  }
```

Control structures don't have to be hardwired into the programming language!

Call-by-name

`VarEnv` from week 2:

```
sealed abstract class VarEnv
case class ConsVarEnv(x: Var, v: Int, next: VarEnv) extends VarEnv
case object NilVarEnv extends VarEnv

⋮
def lookup(e: VarEnv, x: Var): Int = e match {
  case ConsVarEnv(y, w, next) => if (x == y) w else lookup(next, x)
  case NilVarEnv => throw new RuntimeException("not found")
}
```

A variant of `lookup` that uses a by-name default value:

```
def getOrElse(e: VarEnv, x: Var, defaultval: => Int): Int = e match {
  case ConsVarEnv(y, w, next) => if (x == y) w else getOrElse(next, x, defaultval)
  case NilVarEnv => defaultval
}
```

Many classes in Scala's collections library have a similar `getOrElse` method

Example use:

```
val e: VarEnv = …
val v1 = getOrElse(e, "x", -1)
val v2 = getOrElse(e, "y", throw new RuntimeException("not found"))
```

important that the third argument is not using call-by-value!

Functional programming

What characterizes functional programming and functional programming languages?

functional

adjective | func·tion·al | \ˈfəŋ(k)-shnəl , -shə-nᵊl\

1. of or relating to a function or functions :
   functional difficulties in the administration.
2. capable of operating or functioning :
   When will the ventilating system be functional again?
3. having or serving a utilitarian purpose; capable of serving the purpose for which it was designed:
   functional architecture; a chair that is functional as well as decorative.

- Functions as first-class values
  - a mathematical function: maps input to output (and does nothing else)
  - lambdas, higher-order functions
  - useful for defining control abstractions (more examples to follow…)
- Avoiding mutable state and assignments
  - often simplifies reasoning about program behavior (equational reasoning)
  - attractive model for exploiting parallelism for multicore and cloud computing

“Immutability changes everything!” ☺

Functional vs. object-oriented programming

Object-oriented: **identity** is everything!

- the most important property of an object is that it has an identity
- two distinct objects (i.e. different identity) may have equal contents
- example: `new Object() != new Object()`

Functional: **equality** is everything!

- mathematical values do not have identity but equality
- example: we cannot have different instances of the number 5

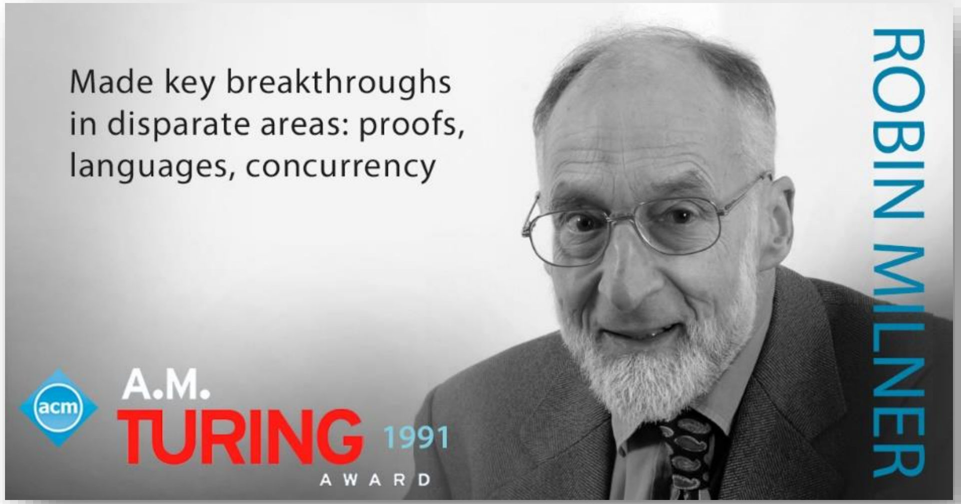
In Java (and Scala): strings are immutable (since Java 1.0)

- Why did the language designers choose that?
- And why does Java also have `StringBuffer` (a mutable variant of `String`)?

Functional programming in Scala

Scala's support for functional programming is heavily inspired by the programming language ML (1973)

Scala lets the programmer choose when to use functional style and when to use imperative/OO style



Essential data types for collections

– both in mathematics and in programming:

- Lists
- Sets
- Maps
- Options

To be mutable,
or not to be mutable,
that is the question

A data type is a set of values, with associated operations

Lists – a cornerstone of functional programming

- The foundation of the Lisp programming language (1958)

- We now focus on immutable lists (mutable lists: later…)
- We earlier defined `IntList`

Immutable integer lists (week 2)

```
sealed abstract class IntList
case class Nil() extends IntList
case class Cons(x: Int, xs: IntList) extends IntList
```

From week 1:
Case class parameters are implicitly declared with ‘val’, meaning that x and xs are immutable fields

```
def length(xs: IntList): Int = xs match {
  case Nil() => 0
  case Cons(_, ys) => 1 + length(ys)
}
```

```
def append(xs: IntList, x: Int): IntList = xs match {
  case Nil() => Cons(x, Nil())
  case Cons(y, ys) => Cons(y, append(ys, x))
}
```

Immutable string lists

Code duplication

Code duplication is an indicator of bad design.

– Objects First with Java, Section 8.4

```
sealed abstract class StringList
case class Nil() extends StringList
case class Cons(x: String, xs: StringList) extends StringList
```

```
def length(xs: StringList): Int = xs match {
  case Nil() => 0
  case Cons(_, ys) => 1 + length(ys)
}
```

```
def append(xs: StringList, x: String): StringList = xs match {
  case Nil() => Cons(x, Nil())
  case Cons(y, ys) => Cons(y, append(ys, x))
}
```

Generic immutable lists

Much like LinkedList<T> in Java, but immutable!

```
sealed abstract class List[T]
case class Nil[T]() extends List[T]
case class Cons[T](x: T, xs: List[T]) extends List[T]
```

```
def length[T](xs: List[T]): Int = xs match {
  case Nil() => 0
  case Cons(_, ys) => 1 + length(ys)
}
```

```
def append[T](xs: List[T], x: T): List[T] = xs match {
  case Nil() => Cons[T](x, Nil[T]())
  case Cons(y, ys) => Cons[T](y, append(ys, x))
}
```

Generic classes and functions

– also called parametric polymorphism
(more common terminology in functional programming)

Abstraction over types!

- `IntList`
- `length(xs: IntList): Int`
- `append(xs: IntList, x: Int): IntList`

- `StringList`
- `length(xs: StringList): Int`
- `append(xs: StringList, x: String): StringList`

⋮

- `List[T]`
- `length[T](xs: List[T]): Int`
- `append[T](xs: List[T], x: T): List[T]`

Generic immutable lists – example uses

```
val a1 = Nil[Int]()
val a2 = Cons[Int](42, a1)
val a3 = Cons(117, a2) // the type argument to Cons is inferred automatically
val b = Cons(117, Cons(42, Nil[Int]()))
println(a3 == b)  // prints true
println(length(b)) // prints 2

val c = Cons("foo", Cons("bar", Nil[String]()))

val d = Cons((1, "foo"), Cons((17, "bar"), Nil[(Int, String)]()))
val e = append(d, (42, "baz"))
println(length(e)) // prints 3
```

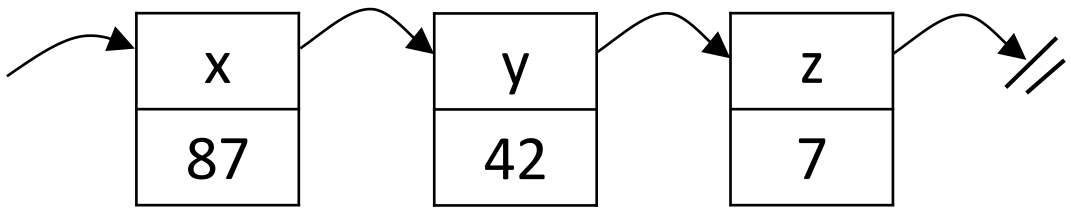
Generic immutable lists – example uses

From week 2:

```
sealed abstract class VarEnv
case class ConsVarEnv(x: Var, v: Int, next: VarEnv) extends VarEnv
case object NilVarEnv extends VarEnv
```

Using our generic `List` definition:

```
type VarEnv = List[(Var, Int)]
```



Option[T] (see also slide 21 from week 3)

A type-safe alternative to `null` values and exceptions (in some situations)

```
sealed abstract class Option[T]
case class None[T]() extends Option[T]
case class Some[T](t: T) extends Option[T]
```

(The definition of `Option[T]` in Scala's standard library is slightly different)

Example:

```
def maxInt(xs: List[Int]): Int = xs match {
  case Nil => throw new IllegalArgumentException("list is empty")
  case Cons(y, Nil) => y
  case Cons(y, ys) => y max maxInt(ys)
}
```

```
def maxInt(xs: List[Int]): Option[Int] = xs match {
  case Nil => None[Int]()
  case Cons(y, ys) =>
    Some[Int](maxInt(ys) match {
      case None[T]() => y
      case Some[T](m) => y max m
    })
}
```

Like `Optional<T>` in Java (as seen in IntProg), but even better with pattern matching!

Option[T] (see also slide 21 from week 3)

`getOrElse` for `Option[T]`, using a by-name parameter:

```
def getOrElse[T](x: Option[T], orelse: => T): T = x match {
  case None[T]() => orelse
  case Some[T](t) => t
}
```

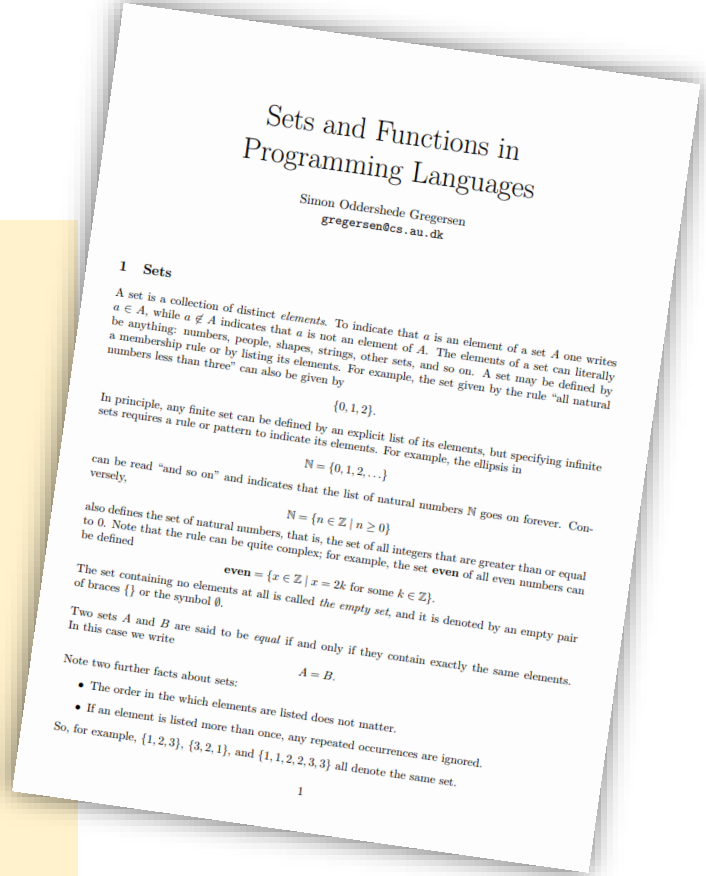
Note: in the definition of `Option[T]` in Scala's standard library,
functions like `getOrElse` are methods in the `Option` class, not separate functions
(like we have seen earlier in exercises 11 and 29)

Implementing sets using lists

Exercise

```scala
type Set[A] =

def makeEmpty[A](): Set[A] = …
def isEmpty[A](set: Set[A]): Boolean = …
def size[A](set: Set[A]): Int = …
def add[A](set: Set[A], x: A): Set[A] = …
def contains[A](set: Set[A], x: A): Boolean =
def remove[A](set: Set[A], x: A): Set[A] = …
def union[A](set1: Set[A], set2: Set[A]): Set[A] = …
def intersection[A](set1: Set[A], set2: Set[A]): Set[A] = …
def difference[A](set1: Set[A], set2: Set[A]): Set[A] = …
```



Sets and Functions in Programming Languages

Simon Oddershede Gregersen
gregersen@cs.au.dk

1 Sets

A set is a collection of distinct *elements*. To indicate that a is an element of a set A one writes $a \in A$, while $a \notin A$ indicates that a is not an element of A. The elements of a set can literally be anything: numbers, people, shapes, strings, other sets, and so on. A set may be defined by a membership rule or by listing its elements. For example, the set given by the rule "all natural numbers less than three" can also be given by

$$\{0, 1, 2\}.$$

In principle, any finite set can be defined by an explicit list of its elements, but specifying infinite sets requires a rule or pattern to indicate its elements. For example, the ellipsis in

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

can be read "and so on" and indicates that the list of natural numbers $\mathbb{N}$ goes on forever. Conversely,

$$\mathbb{N} = \{n \in \mathbb{Z} \mid n \geq 0\}$$

also defines the set of natural numbers, that is, the set of all integers that are greater than or equal to 0. Note that the rule can be quite complex; for example, the set **even** of all even numbers can be defined

$$\mathbf{even} = \{x \in \mathbb{Z} \mid x = 2k \text{ for some } k \in \mathbb{Z}\}.$$

The set containing no elements at all is called *the empty set*, and it is denoted by an empty pair of braces $\{\}$ or the symbol $\emptyset$.

Two sets A and B are said to be *equal* if and only if they contain exactly the same elements. In this case we write

$$A = B.$$

Note two further facts about sets:

- The order in the which elements are listed does not matter.
- If an element is listed more than once, any repeated occurrences are ignored.

So, for example, $\{1, 2, 3\}$, $\{3, 2, 1\}$, and $\{1, 1, 2, 2, 3, 3\}$ all denote the same set.

1

Homework exercises

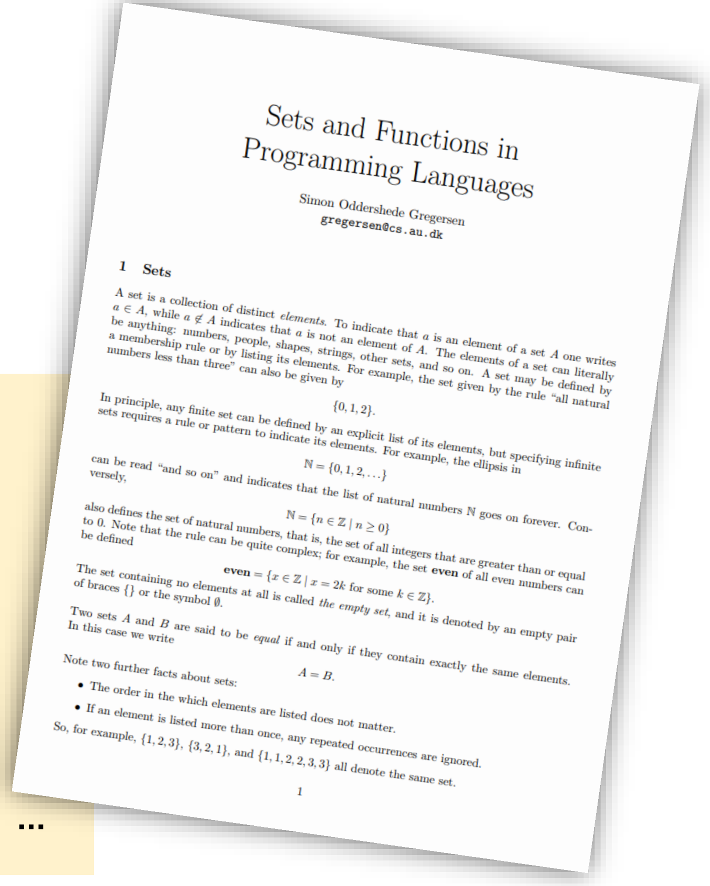
- implement (some of) these functions
- use them instead of Scala's standard library to represent `Set[Id]` in the MiniScala interpreter

Implementing maps using lists

```
type Map[K, V] =
```

Exercise

```
def makeEmpty[K, V](): Map[K, V] = …
def extend[K, V](e: Map[K, V], x: K, v: V): Map[K, V] = …
def lookup[K, V](e: Map[K, V], x: K): V = …
def getOrElse[K, V](e: Map[K, V], x: K, orelse: => V): V = …
```



Sets and Functions in Programming Languages

Simon Oddershede Gregersen
gregersen@cs.au.dk

1 Sets

A set is a collection of distinct *elements*. To indicate that a is an element of a set A one writes $a \in A$, while $a \notin A$ indicates that a is not an element of A. The elements of a set can literally be anything: numbers, people, shapes, strings, other sets, and so on. A set may be defined by a membership rule or by listing its elements. For example, the set given by the rule "all natural numbers less than three" can also be given by

$$\{0, 1, 2\}.$$

In principle, any finite set can be defined by an explicit list of its elements, but specifying infinite sets requires a rule or pattern to indicate its elements. For example, the ellipsis in

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

can be read "and so on" and indicates that the list of natural numbers $\mathbb{N}$ goes on forever. Conversely,

$$\mathbb{N} = \{n \in \mathbb{Z} \mid n \geq 0\}$$

also defines the set of natural numbers, that is, the set of all integers that are greater than or equal to 0. Note that the rule can be quite complex; for example, the set **even** of all even numbers can be defined

$$\textbf{even} = \{x \in \mathbb{Z} \mid x = 2k \text{ for some } k \in \mathbb{Z}\}.$$

The set containing no elements at all is called *the empty set*, and it is denoted by an empty pair of braces {} or the symbol $\emptyset$.

Two sets A and B are said to be *equal* if and only if they contain exactly the same elements. In this case we write

$$A = B.$$

Note two further facts about sets:

- The order in the which elements are listed does not matter.
- If an element is listed more than once, any repeated occurrences are ignored.

So, for example, $\{1, 2, 3\}$, $\{3, 2, 1\}$, and $\{1, 1, 2, 2, 3, 3\}$ all denote the same set.

1

<u>Bonus homework exercises</u>

- implement (some of) these functions
- use them instead of Scala's standard library to represent `Map[Id, Val]` in the MiniScala interpreter

Implementing lists using sets or maps

We could equally well implement lists using sets or maps
(lists, sets, and maps are mathematically all equally expressive, ignoring performance)

```
type List[A] = Set[(Int, A)]
```

the integer represents the position of the element in the list

```
type List[A] = Map[Int, A]
```

Programming exercise: concatenating IntLists

```
sealed abstract class IntList
case object Nil extends IntList
case class Cons(x: Int, xs: IntList) extends IntList

def concat(xs: IntList, ys: IntList): IntList = ???
```

Exercise: Implement the function concat that concatenates two IntLists

(If you have done your homework, this exercise is easy ☺)

Remember the key principle from week 2:
follow the inductive definition of the data

Example:

```
concat(Cons(1, Cons(2, Nil)), Cons(3, Cons(4, Cons(5, Nil))))
= Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

Reversing lists (week 2)

```
sealed abstract class IntList
case object Nil extends IntList
case class Cons(x: Int, xs: IntList) extends IntList

def append(xs: IntList, x: Int): IntList = xs match {
  case Nil => Cons(x, Nil)
  case Cons(y, ys) => Cons(y, append(ys, x))
}

def reverse(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(x, ys) => append(reverse(ys), x)
}
```

As discussed earlier, it takes in the order of N^2 computation steps to reverse a list with N elements using this implementation

Reversing lists – mix of imperative and functional (1)

We want a linear time implementation of `reverse`

```
def reverse(xs: IntList): IntList = {
  var r = Nil
  var it = xs
  while (!it.isEmpty()) {
    r = Cons(it.head(), r)
    it = it.tail()
  }
  r
}
```

ensuring absence of exceptions requires extra reasoning

Assume `IntList` elements have some extra methods

```
sealed abstract class IntList {
  def isEmpty(): Boolean
  def head(): Int
  def tail(): IntList
}

case object Nil extends IntList {
  def isEmpty() = true
  def head() = throw new IllegalArgumentE
  def tail() = throw new IllegalArgumentException("empty!")
}

case class Cons(x: Int, xs: IntList) extends IntList {
  def isEmpty() = false
  def head() = x
  def tail() = xs
}
```

Reversing lists – mix of imperative and functional (2)

```
def reverse(xs: IntList): IntList = {
  var r = Nil
  def rev(xs: IntList) = xs match {
    case Nil => r
    case Cons(x, ys) =>
      r = Cons(x, r)
      rev(ys)
  }
  rev(xs)
}
```

Now using recursion and pattern matching (and `IntList` is `sealed`, so no risk of exceptions), but still using mutable state

Reversing lists – functional, with accumulator

```
def reverse(xs: IntList): IntList = {
  def rev(xs: IntList, acc: IntList) = xs match {
    case Nil => acc
    case Cons(x, ys) => rev(ys, Cons(x, acc))
  }
  rev(xs, Nil)
}
```

How it works, by example:

```
reverse(Cons(1, Cons(2, Cons(3, Nil))))
= rev(Cons(1, Cons(2, Cons(3, Nil))), Nil)
= rev(Cons(2, Cons(3, Nil)), Cons(1, Nil))
= rev(Cons(3, Nil), Cons(2, Cons(1, Nil)))
= rev(Nil, Cons(3, Cons(2, Cons(1, Nil))))
= Cons(3, Cons(2, Cons(1, Nil)))
```

Structural induction on IntLists, cont.

Two purely functional implementations of `reverse`:

`reverse1`: easy to understand, but inefficient (quadratic time)

`reverse2`: more complicated, but efficient (linear time)

```
sealed abstract class IntList
case object Nil extends IntList
case class Cons(x: Int, xs: IntList) extends IntList


def append(xs: IntList, x: Int): IntList = xs match {
  case Nil => Cons(x, Nil)
  case Cons(y, ys) => Cons(y, append(ys, x))
}


def reverse1(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(x, ys) => append(reverse1(ys), x)
}


def reverse2(xs: IntList): IntList = {
  def rev(xs: IntList, acc: IntList): IntList = xs match {
    case Nil => acc
    case Cons(x, ys) => rev(ys, Cons(x, acc))
  }
  rev(xs, Nil)
}
```

<u>Theorem:</u> `reverse1(xs) = reverse2(xs)` for any IntList xs

Structural induction on IntLists, cont.

Let's try to prove that `reverse1(xs)` = `reverse2(xs)` for any IntList `xs` by induction in xs

- Base case, `xs` = `Nil`: trivial…
- Inductive step, `xs = Cons(y, ys)`:

using the induction hypothesis

Induction hypothesis: `reverse1(ys) = reverse2(ys)`

`reverse1(xs)` = `reverse1(Cons(y, ys))` = `append(reverse1(ys), y)` = `append(reverse2(ys), y)` = `append(rev(ys, Nil), y)`

`reverse2(xs)` = `reverse2(Cons(y, ys))` = `rev(Cons(y, ys), Nil)` = `rev(ys, Cons(y, Nil))`

If we can prove the following lemma, we're done:

Lemma 1 : `append(rev(ys, Nil), y)` = `rev(ys, Cons(y, Nil))` for any IntList `ys` and any int `y`

Structural induction on IntLists, cont.

Lemma 1: `append(rev(ys, Nil), y) = rev(ys, Cons(y, Nil))` for any IntList `ys` and any int `y`

Proof (attempt) by induction in `ys`:

- Base case, `ys = Nil`: trivial...
- Inductive step, `ys = Cons(z, zs)`:

  Induction hypothesis: `append(rev(zs, Nil), t) = rev(zs, Cons(t, Nil))` for any int `t`

  `append(rev(ys, Nil), y) = append(rev(Cons(z, zs), Nil), y) = append(rev(zs, Cons(z, Nil)), y) =` **?**

  `rev(ys, Cons(y, Nil)) = rev(Cons(z, zs), Cons(y, Nil)) = rev(zs, Cons(z, Cons(y, Nil)) =` **?**

  We seem to be stuck, can't apply the induction hypothesis ☹

Structural induction on IntLists, cont.

Lemma ①: `append(rev(ys, Nil), y)` = `rev(ys, Cons(y, Nil))` for any IntList `ys` and any int `y`

Lemma ②: `append(rev(ys, acc), y)` = `rev(ys, append(acc, y))` for any IntLists `ys` and `acc` and any int `y`

Proof that lemma ② implies lemma ①: Homework exercise ☺

Proof of lemma ②: Homework exercise ☺

This result provides the missing piece for the proof from slide 31

Sometimes it pays off to prove a stronger property than what is needed

Iteration with immutable accumulators

A general idea:

Instead of using a mutable variable declared outside the loop,
pass an extra “accumulator” argument that eventually becomes the result

Sorting IntLists with QuickSort

```
/** Partitions xs into two lists: one containing the elements that satisfy p,
   and one containing the elements that do not satisfy p */
def partition(xs: IntList, p: Int => Boolean): (IntList, IntList) = ???

def qsort(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) =>
    val (smaller, rest) = partition(ys, t => t < y)
    concat(qsort(smaller), Cons(y, qsort(rest)))
}
```

"There are two ways of constructing a software design:
One way is to make it so simple
that there are obviously no deficiencies,
and the other way is to make it so complicated
that there are no obvious deficiencies."
– C.A.R. (Tony) Hoare

Partitioning IntLists – using mutable state

```
def partition(xs: IntList, p: Int => Boolean): (IntList, IntList) = {
  var p1, p2: IntList = Nil
  def part(xs: IntList): Unit = xs match {
    case Nil => // do nothing
    case Cons(y, ys) =>
      part(ys)
      if (p(y)) p1 = Cons(y, p1) else p2 = Cons(y, p2)
  }
  part(xs)
  (p1, p2)
}
```

Partitioning IntLists – using accumulator

```
def partition(xs: IntList, p: Int => Boolean): (IntList, IntList) = {
  def part(xs: IntList, acc: (IntList, IntList)): (IntList, IntList) = xs match {
    case Nil => acc
    case Cons(y, ys) =>
```

Exercise

```
  }
  part(xs, (Nil, Nil))
}
```

Using accumulators in tree operations

Recall IntTrees from week 5 (exercise 54c):

```
sealed abstract class IntTree
case object Leaf extends IntTree
case class Branch(left: IntTree, x: Int, right: IntTree) extends IntTree

def flatten(t: IntTree): IntList = … // convert to IntList using left-to-right inorder

def flatten2(t: IntTree): IntList = {
  def f(t: IntTree, acc: IntList): IntList = t match {
    case Leaf => acc
    case Branch(left, x, right) =>
  }
  f(t, Nil)
}
```

Exercise

How can avoiding mutable state be an advantage?

- As discussed week 5, immutability simplifies reasoning about program behavior
  - Case in point: try to prove that the `reverse` function from slide 28 and the one from slide 26 are functionally equivalent
    (it is of course possible, but more complicated than for the fully functional variant)

- Also, immutability makes sharing and aliasing irrelevant!
  - Example: Does append create an entirely new IntList, or does it reuse elements from the input list?
  - IntLists are immutable, so it doesn't matter!

```
def append(xs: IntList, x: Int): IntList
```

  - Next, an example from Java where mutability matters…

How can avoiding mutable state be an advantage?

A typical real-world Java example:

```
class ProtectedResource {
  private Resource theResource = ...;
  private String[] allowedUsers = ...;
  public String[] getAllowedUsers() {
    return allowedUsers;
  }
  public String currentUser() { ... }
  public void useTheResource() {
    for (int i=0; i < allowedUsers.length; i++) {
      if (currentUser().equals(allowedUsers[i])) {
        ... // access allowed: use it
        return;
      }
    }
    throw new IllegalAccessException();
  }
}
```

What's the problem here?

```
⋮
p.getAllowedUsers()[0] = p.currentUser()
⋮
```

Defensive copying necessary (which is easy to forget):

```
public String[] getAllowedUsers() {
  String[] copy = new String[allowedUsers.length];
  for (int i=0; i < allowedUsers.length; i++)
    copy[i] = allowedUsers[i];
    return copy;
  }
}
```

Defensive copying is unnecessary if state is immutable

Control abstractions using higher-order functions

Code duplication

Code duplication is an indicator of bad design.

– Objects First with Java, Section 8.4

```
sealed abstract class IntList
case object Nil extends IntList
case class Cons(x: Int, xs: IntList) extends IntList

def length(xs: IntList): Int = xs match {
  case Nil => 0
  case Cons(y, ys) => 1 + length(ys)
}

def sum(xs: IntList): Int = xs match {
  case Nil => 0
  case Cons(y, ys) => y + sum(ys)
}

def prod(xs: IntList): Int = xs match {
  case Nil => 1
  case Cons(y, ys) => y * prod(ys)
}
```

Remember the `filter` example from last week?

Control abstractions using higher-order functions

Code duplication

Code duplication is an indicator of bad design.

– Objects First with Java, Section 8.4

```
sealed abstract class IntList
case object Nil extends IntList
case class Cons(x: Int, xs: IntList) extends IntList

def length(xs: IntList): Int = xs match {
```

```
}
```

```
def fold(xs: IntList, z: Int, f: (Int, Int) => Int): Int = xs match {
  case Nil => z
  case Cons(y, ys) => f(y, fold(ys, z, f))
}
```

first parameter is the next element in the list

second parameter is like an accumulator

```
def length(xs: IntList): Int = fold(xs, 0, (x, y) => 1 + y)

def sum(xs: IntList): Int = fold(xs, 0, (x, y) => x + y)

def prod(xs: IntList): Int = fold(xs, 1, (x, y) => x * y)
```

Example: `fold(Cons(1, Cons(2, Cons(3, Nil))))`
`= f(1, f(2, f(3, z)))`

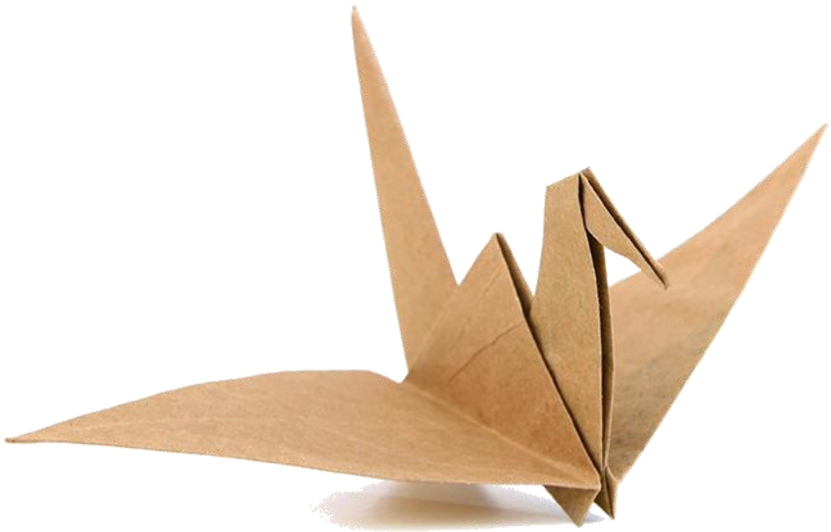
The traversal is now separated from the operations – and is reusable!

Mutable state vs. folding

A general pattern (informally):

```
{ var r = init
  for (x <- xs)
    r = f(x,r)
  r }
```

```
fold(xs, init, f)
```



Polymorphic fold

```
def fold(xs: IntList, z: Int, f: (Int, Int) => Int): Int = xs match {
  case Nil => z
  case Cons(y, ys) => f(y, fold(ys, z, f))
}
```

```
def fold[B](xs: IntList, z: B, f: (Int, B) => B): B = xs match {
  case Nil => z
  case Cons(y, ys) => f(y, fold(ys, z, f))
}
```

```
def fold[A, B](xs: List[A], z: B, f: (A, B) => B): B = xs match {
  case Nil() => z
  case Cons(y, ys) => f(y, fold(ys, z, f))
}
```

foldRight vs. foldLeft

```
def foldRight[A,B](xs: List[A], z: B, f: (A, B) => B): B = xs match {
  case Nil() => z
  case Cons(y, ys) => f(y, foldRight(ys, z, f))
}
```

previously called `fold`

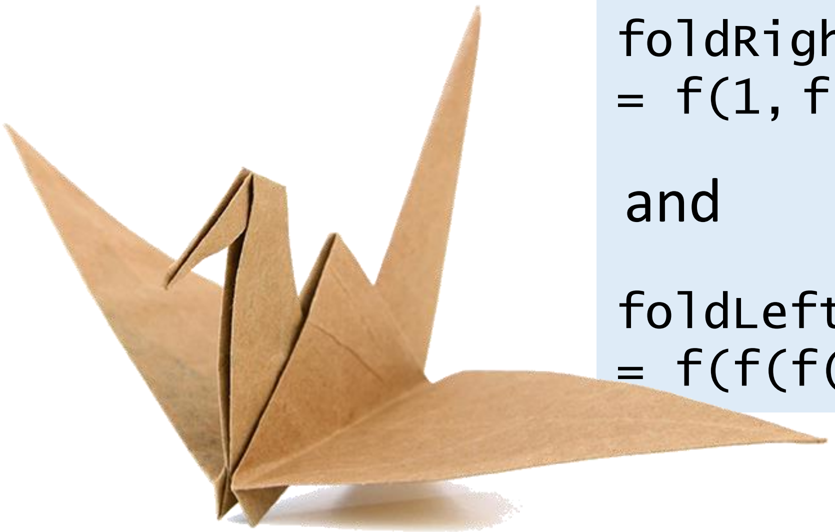
```
def foldLeft[A,B](xs: List[A], z: B, f: (B, A) => B): B = xs match {
  case Nil() => z
  case Cons(y, ys) => foldLeft(ys, f(z, y), f)
}
```

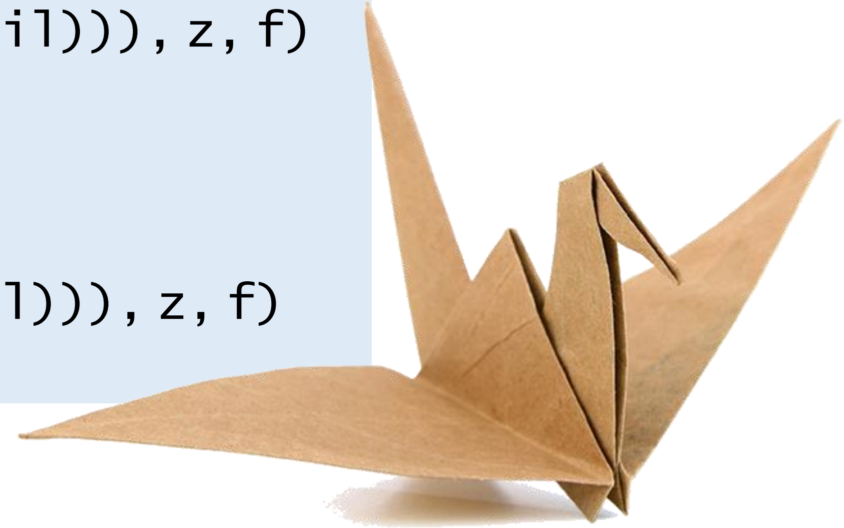
Exercise: Check that

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), z, f)
= f(1, f(2, f(3, z)))
```

and

```
foldLeft(Cons(1, Cons(2, Cons(3, Nil))), z, f)
= f(f(f(z, 1), 2), 3)
```





Reversing lists with foldRight/foldLeft (non-polymorphic)

Original `reverse`:

```
def reverse1(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(x, ys) => append(reverse1(ys), x)
}
```

Using accumulator trick:

```
def reverse2(xs: IntList): IntList = {
  def rev(xs: IntList, acc: IntList) = xs match {
    case Nil => acc
    case Cons(x, ys) => rev(ys, Cons(x, acc))
  }
  rev(xs, Nil)
}
```

Using `foldRight`:

```
def reverse3(xs: IntList): IntList = foldRight(xs, Nil, (a: Int, b: IntList) => append(b, a))
```

Using `foldLeft`:

```
def reverse4(xs: IntList): IntList = foldLeft(xs, Nil, (b: IntList, a: Int) => Cons(a, b))
```

Which one do you prefer, and why?

Partitioning IntLists with fold

From slide 37:

From slide 36:

```
def partiti
  var p1, p
  def part(
    case Ni
    case Co
      part(
      if (p
  }
  part(xs)
  (p1, p2)
}
```

```
def partition(xs: IntList, p: Int => Boolean): (IntList, IntList) = {
  def part(xs: IntList, acc: (IntList, IntList)): (IntList, IntList) = xs match {
    case Nil => acc
    case Cons(y, ys) =>
      val (acc1, acc2) = part(ys, acc)
      if (p(y)) (Cons(y, acc1), acc2) else (acc1, Cons(y, acc2))
  }
  part(xs, (Nil, Nil))
}
```

```
def partition(xs: IntList, p: Int => Boolean): (IntList, IntList) =
  foldRight(xs, (Nil, Nil), (y, acc: (IntList, IntList)) =>
    if (p(y)) (Cons(y, acc._1), acc._2) else (acc._1, Cons(y, acc._2)))
```

Which style do you prefer?

Fold on other data types

The “fold” idea works on any inductive data type!

Example: `fold` on `Option[T]`

```
sealed abstract class Option[T]
case class None[T]() extends Option[T]
case class Some[T](t: T) extends Option[T]
```

by-name parameter useful here,
because the argument may not be used

```
def fold[T,B](opt: Option[T], ifEmpty: => B, f: T => B) = opt match {
  case None() => ifEmpty
  case Some(t) => f(t)
}
```

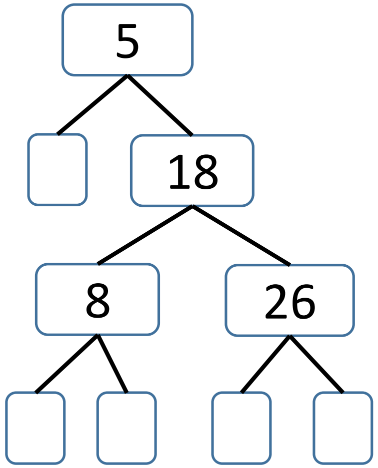
Homework exercises: implement and use fold on `Nat` and `Exp`

https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Fold on trees

Recall IntTree from week 5 (exercise 54c):

```
sealed abstract class IntTree
case object Leaf extends IntTree
case class Branch(left: IntTree, x: Int, right: IntTree) extends IntTree
```



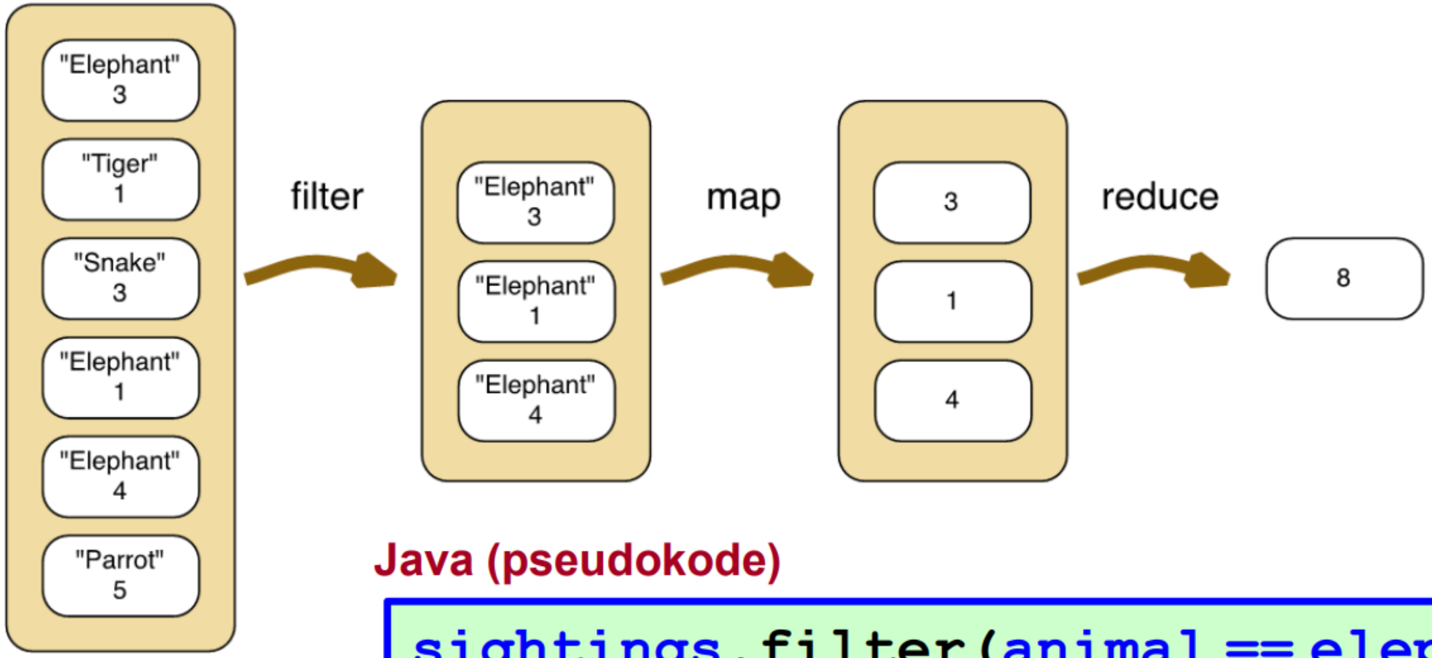

```
def fold[B](t: IntTree, z: B, f: (B, Int, B) => B): B = t match {
  case Leaf => z
  case Branch(left, x, right) => f(fold(left, z, f), x, fold(right, z, f))
}
```

(Other variants of fold on trees are possible)

Homework exercises: implement `flatten` using `fold`

Working with collections using higher-order functions

From the IntProg course:



Java (pseudokode)

```
sightings.filter(animal == elephant)
         .map(count)
         .reduce(sum);
```

Now let's see how `filter`, `map`, and `reduce` can be implemented (in Scala)...

filter

Code duplication

Code duplication is an indicator of bad design.

– Objects First with Java, Section 8.4

From last week:

```
def getPositiveNumbers(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) => {
    val rest = getPositiveNumbers(ys)
    if (y > 0) Cons(y, rest) else rest
  }
}

def getEvenNumbers(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) => {
    val rest = getEvenNumbers(ys)
    if (y % 2 == 0) Cons(y, rest) else rest
  }
}
```

filter

```
def filter(xs: IntList, p: Int => Boolean): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) =>
    val r = filter(ys, p)
    if (p(y)) Cons(y, r) else r
}


def getPositiveNumbers(xs: IntList): IntList = filter(xs, x => x > 0)

def getEvenNumbers(xs: IntList): IntList = filter(xs, x => x % 2 == 0)
```

Polymorphic:

```
def filter[T](xs: List[T], p: T => Boolean): List[T] = xs match {
  case Nil() => Nil[T]()
  case Cons(y, ys) =>
    val r = filter(ys, p)
    if (p(y)) Cons(y, r) else r
}
```

map

Code duplication

Code duplication is an indicator of bad design.

– Objects First with Java, Section 8.4

```
def increment(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) => Cons(y + 1, increment(ys))
}

def double(xs: IntList): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) => Cons(y * 2, double(ys))
}
```

```
def map(xs: IntList, f: Int => Int): IntList = xs match {
  case Nil => Nil
  case Cons(y, ys) => Cons(f(y), map(ys, f))
}


def increment(xs: IntList): IntList = map(xs, x => x + 1))

def double(xs: IntList): IntList = map(xs, x => x * 2)
```

Polymorphic map? Exercise…

reduce

Variants of `foldRight` / `foldLeft` for use on nonempty collections:

- `reduceRight(Cons(1, Cons(2, Cons(3, Nil)), f)`
  `= f(1, f(2, 3))`
- `reduceLeft(Cons(1, Cons(2, Cons(3, Nil)), f)`
  `= f(f(1, 2), 3)`
- Other variants:
  - `reduce` uses unspecified order (left or right) – only use with associative operations!
  - `reduceOption` returns `Option[T]`, instead of throwing exception if empty

```
def reduceRight[T](xs: List[T], f: (T, T) => T): T = xs match {
  case Nil() => throw new IllegalArgumentException("empty!")
  case Cons(y, Nil()) => y
  case Cons(y, ys) => f(y, reduceRight(ys, f))
}
```

Collections in Scala’s standard library

- Scala’s standard library classes `List`, `Map`, `Option`, etc. provide 100s of useful methods like `fold`, `filter`, `map`, etc.
- They are defined as methods inside the classes, not as separate functions outside the classes
- Many use currying (see slide 7)
  and other Scala features that we haven’t discussed (yet), in particular variance and traits

- So they are sometimes invoked slightly differently compared to the functions we have defined
- Example:
  - `fold` on Scala’s `List[T]`:

```
sealed abstract class List[A] { // simplified
  ⋮
  def foldRight[B](z: B)(f: (A, B) => B): B = …
}
```

  - `fold` from slides 16+45:

```
sealed abstract class List[A] { … }
def foldRight[A,B](xs: List[A], z: B, f: (A, B) => B): B = …
```

Initial environment – imperative style

(but using immutable maps)

Create initial environment with value from user for each occurring variable

```
def makeInitialEnv(program: Exp): VarEnv = {
  var env = Map[Var, Int]()
  for (x <- freeVars(program)) {
    print(s"Please provide a value for the variable $x: ")
    env = env + (x -> StdIn.readInt())
  }
  env
}
```

Initial environment – functional style

Create initial environment with value from user for each occurring variable

```
def makeInitialEnv(program: Exp): VarEnv =
  freeVars(program).foldLeft(Map[Var, Int]())((env, x) => {
    print(s"Please provide an value for the variable $x: ")
    env + (x -> IntVal(StdIn.readInt()))
  })
```

Which style do you prefer?

Imperative vs. functional style

Homework exercise:

Look through your MiniScala implementation,
rewrite use of mutable state (`var`) and iteration (`for`-loops)
to use higher-order functions instead (in particular `foldLeft` or `foldRight`)
– or vice versa if your implementation is already fully "functional"

Obvious places to look:

- function definitions with lists of parameters, and calls with lists of arguments
- blocks with lists of declarations and expressions

Implementing environments in the interpreter

We have seen three approaches:

1. Using `Map` from the standard library

```
type Env = Map[Id, Val]
```

2. Using our own inductive data type (linked lists)

```
sealed abstract class Env
private case class ConsEnv(x: Id, v: Val, next: Env) extends Env
private case object NilEnv extends Env
```

3. Using higher-order functions

```
type Env = Id => Val
```

Implementing environments using higher-order functions

```scala
type Env = Id => Val

def makeEmpty(): Env =
  (x: Id) => throw new RuntimeException("not found")

def extend(e: Env, x: Id, v: Val): Env =
  (y: Id) => if (x == y) v else e(y)

def lookup(e: Env, x: Id): Val = e(x)
```

We can use this representation of environments to simplify the way recursion is handled in our MiniScala interpreter!

First step, using Option[T] instead of exceptions

```
type Env = Id => Option[Val]

def makeEmpty(): Env =
  (x: Id) => None

def extend(e: Env, x: Id, v: Val): Env =
  (y: Id) => if (x == y) Some(v) else e(y)

def getOrElse(env: Env, id: Id, default: => Val): Val =
  env(id).getOrElse(default)
```

using by-name parameter

the `getOrElse` method in `Option`

Recursion in MiniScala

Recall how we handle recursion in MiniScala v5 (slide 17 from week 6):

all functions definitions in the current block

$$\frac{\rho' = \rho[f^1 \mapsto (x^1, e^1, \rho, D), \ldots, f^n \mapsto (x^n, e^n, \rho, D)] \qquad D = \{\texttt{def}\ f^1(x^1) = e^1, \ldots, \texttt{def}\ f^n(x^n) = e^n\}}{\rho \vdash \texttt{def}\ f^1(x^1) = e^1;\ \ldots;\ \texttt{def}\ f^n(x^n) = e^n \Rightarrow \rho'}$$

rebinds all the functions from the block

$$\frac{\begin{array}{l}\rho \vdash e_0 \Rightarrow (x, e_2, \rho_2, D) \\ \rho \vdash e_1 \Rightarrow v_1\end{array} \qquad \begin{array}{r} D = \{\texttt{def}\ f^1(x^1) = e^1, \ldots, \texttt{def}\ f^n(x^n) = e^n\} \\ \rho_2[x \mapsto v_1, \boxed{f^1 \mapsto (x^1, e^1, \rho_2, D), \ldots, f^n \mapsto (x^n, e^n, \rho_2, D)}] \vdash e_2 \Rightarrow v_2\end{array}}{\rho \vdash e_0\ (e_1) \Rightarrow v_2}$$

Recursive environments

ρ’ is now defined **recursively**

$$\frac{\rho' = \rho[f \mapsto (x, e, \rho')]}{\rho \vdash \texttt{def}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e \Rightarrow \rho'}$$

$$\frac{\rho \vdash e_0 \Rightarrow (x, e_2, \rho_2) \qquad \rho \vdash e_1 \Rightarrow v_1 \qquad \rho_2[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\rho \vdash e_0\texttt{(}e_1\texttt{)} \Rightarrow v_2}$$

The rule for function calls is the same as slide 14 from week 6, but now recursive functions work!

Recursive environments

Also works smoothly for mutual recursion:

$$\frac{\rho' = \rho[f^1 \mapsto (x^1, e^1, \rho'), \ldots, f^n \mapsto (x^n, e^n, \rho')]}{\rho \vdash \texttt{def}\ f^1\texttt{(}x^1\texttt{)} \texttt{=} e^1 \texttt{;} \ldots \texttt{;} \texttt{def}\ f^n\texttt{(}x^n\texttt{)} \texttt{=} e^n \Rightarrow \rho'}$$

No need for the “rebinding” of functions at the call sites

(compare with the rule for `def` on slide 64)

Implementing recursive environments

recursive

$$\rho' = \rho[f \mapsto (x, e, \rho')]$$

Cannot be implemented with `Map` or with our immutable linked-list data type

– but it is possible using the higher-order function approach!

```
def extendWithClosure(env: Env, d: DefDecl): Env = {
  def env2(y: Id): Option[Val] =
    extend(env, d.fun, ClosureVal(d.param, d.body, env2))(y)
  env2
}
```

exploiting the fact that `def`s (in the meta-language) can be recursive

Implementing recursive environments

$$\rho' = \rho[\overbrace{f^1 \mapsto (x^1, e^1, \rho'), \ldots, f^n \mapsto (x^n, e^n, \rho')}^{\text{recursive}}]$$

Extending to multiple defs, to support mutual recursion:

```
def extendWithClosures(env: Env, defs: List[DefDecl]): Env = {
  def env2(y: Id): Option[Val] =
    defs.foldLeft(env)((e, d) => extend(e, d.fun, ClosureVal(d.param, d.body, env2)))(y)
  env2
}
```

We no longer need the “rebinding” at `CallExp` or the `defs` field in `ClosureVal` ☺

Summary

- Call-by-name: evaluate arguments when used inside the function
  - Call-by-value: evaluate arguments before the function body
- Essential collections: immutable lists, sets, maps, and options
- Parametric polymorphism: abstracting over types
- Iteration with accumulators (vs. using mutable state)
- Control abstractions and data processing with higher-order functions
  - fold, filter, map, etc.
- Functional programming in the MiniScala implementation
  - Avoiding mutable state
  - Implementing recursive environments using higher-order functions