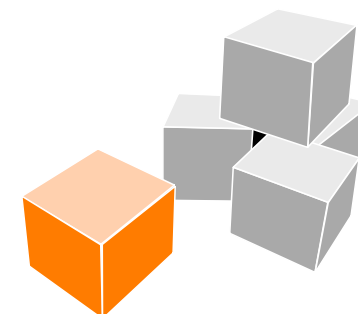


Concepts of Programming Languages

Prof. Dr. Guido Salvaneschi



***Software
Technology
Group***

TU Darmstadt | FB Informatik

WHAT HAPPENED SO FAR

Tentative Overview of the Course Topics

- Arithmetic expressions
- First-order and first-class functions
- Lazy evaluation
- Recursion
- State
- Continuations
- OO concepts
- Garbage collection
- Reactive Programming
- Formal specification of semantics
- Type systems
- ...

A Taxonomy of Functions

- **First-class functions**

- Functions are values/objects with all the rights of other values
 - Can be constructed at runtime
 - Can be passed as arguments to other functions, returned by other functions, stored in data structures etc
- No first-class functions => functions can only be defined in designated portions of the program, where they are given names for use in the rest of the program

- **Higher-order functions**

- Functions that return and/or take other functions as parameters
- Parameterize computations over other computations

- **First-order Functions**

- Functions that neither return nor take other functions as parameters
- Parameterise computations over data

ABSTRACTION!!

A Taxonomy of Functions

Some languages achieve some kind of higher-orderness without first class functions

- Functions pointers (C, C++)
- Objects with an “call” method
 - “A function is an object with a single method”
- Eval

F1WAE

- A language with **first-order** functions
 - Function applications are expressions
 - No first-class/higher-order functions:
 - Function definitions are **not** expressions
- ⇒ separate definitions from expressions
- predefined functions given to interpreter as an argument

Concrete & Abstract Syntax for F1LAE

```
<F1LAE> ::= <num>
          | {+ <F1LAE> <F1LAE>}
          | {- <F1LAE> <F1LAE>}
          | {let {<id> <F1LAE>} <F1LAE>}
          | <id>
          | {<id> <F1LAE>}
```

concrete syntax for
function application

What does this tell us about valid
F1LAE programs?

```
sealed abstract class F1LAE
case class Num(n: Int) extends F1LAE
case class Add(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Sub(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Let(name: Symbol, namedExpr: F1LAE, body: F1LAE) extends F1LAE
case class Id(name: Symbol) extends F1LAE
case class App(funName: Symbol, arg: F1LAE) extends F1LAE
```

abstract syntax for
function application

Function Definitions in F1LAE

- Cannot define functions in F1LAE
 - More strict than necessary for first-orderness
- Predefined functions are passed to the interpreter
- How to represent functions?

```
case class FunDef(argName: Symbol, body: F1LAE)
type FunDefs = Map[Symbol, FunDef]
```

- Example

```
FunDef('n, Add('n, 1))
```

- Class **FunDef** does not extend class **F1LAE** => no syntax for function definitions

F1WAE Interpreter


```
def interp(expr: F1LAE, funDefs: Map[Symbol, FunDef]): Int = expr match {  
  case Num(n) =>  
    n  
  case Add(lhs, rhs) =>  
    interp(lhs, funDefs) + interp(rhs, funDefs)  
  case Sub(lhs, rhs) =>  
    interp(lhs, funDefs) - interp(rhs, funDefs)  
  
  case Let(id, expr, body) =>  
    val body = subst(body, id, Num(interp(expr, funDefs)))  
    interp(body, funDefs)  
  
  case Id(name) => sys.error("found unbound id " + name)  
  
  case App(fun, arg) => funDefs(fun) match {  
    case FunDef(param, body) =>  
      interp(subst(body, param, Num(interp(arg, funDefs))), funDefs)  
  }  
}
```

SUBSTITUTION

Discussion on Scoping

What is the result of interpreting `{f 10}` where `{f n} = {n n}`?

```
interp(  
  App('f, 10),  
  Map('f -> FunDef('n, App('n, 'n)))  
)
```



- Should the interpreter try to substitute the `n` in function position of `App` with `10`? What would happen if this is the case?
- Or should function names and function arguments live in separate “spaces” ==> an identifier in function position within an `App` is not replaced.

Let's see what our interpreter does...

Discussion on Scoping: Namespaces

- Should the interpreter try to substitute the `n` in the function position of the application with the number 10, then complain that no such function can be found (or even that the lookup fails because the names of functions must be identifiers, not numbers)?
- `(f 10) -> (interp (subst (app 'n (id 'n)) n 10)) -> (interp (app 10 10)) -> either “10: no such function definition” or “10 is invalid function name”`
- **Single namespace** (e.g. Scheme). The name of a function can be bound to a value in a local scope, thereby rendering the function inaccessible through that name.
- **Multiple namespaces** (e.g., Common Lisp). The interpreter decide that function names and function arguments live in two separate “spaces”, and context determines in which space to look up a name.
 - An identifier in a function position within an app-expr is not replaced.
 - F1WAE employs namespaces.
 - Running F1WAE, the result is “`n: : no such function definition`”. Identifiers (incl. function arguments) and function names are “looked-up” differently (identifiers are not looked-up at all but immediately substituted as soon as a corresponding binding instance is found for them).

let and Environments

- The interpreter receives a store called **environment**, which maps identifiers to values
- `{let {x e} t}` simply stores in the **environment** a mapping from `x` to the value `e` evaluates to
- When the interpreter encounters an `id` expression, it looks up the corresponding value in the environment
- Free variables not in environment → **error**
- We represent environments as values of the type **Env**:

```
type Env = Map[Symbol, Int]
```

Is this our F1LAE with Environments?

```
def interp(expr: F1LAE, funDefs: FunDefs, env: Env): Int = expr match {  
  case Num(n) =>  
    n  
  case Add(lhs, rhs) =>  
    interp(lhs, funDefs, env) + interp(rhs, funDefs, env)  
  case Sub(lhs, rhs) =>  
    interp(lhs, funDefs, env) - interp(rhs, funDefs, env)  
  
  case Let(id, expr, body) =>  
    val newEnv = env + (id -> interp(expr, funDefs, env))  
    interp(body, funDefs, newEnv)  
  
  case Id(name) =>  
    env(name)  
  
  case App(fun, arg) => funDefs(fun) match {  
    case FunDef(param, body) =>  
      val funEnv = env + (param -> interp(arg, funDefs, env))  
      interp(body, funDefs, funEnv)  
  }  
}
```

What's the result of evaluating
`{let {n 5} {f 10}}`
where `{f x} = {n}`?

What is the answer when using
F1WAE with substitution?

Static Versus Dynamic Scoping

Definition Scope (of a name binding):

The scope of a name binding is the part of the program where the binding is in effect.

Definition Static/Lexical Scoping:

The scope of a name binding is determined syntactically (at compile-time).

Definition Dynamic Scoping:

The scope of a name binding is determined by the execution context (at runtime).

F1LAE with Environments

```
def interp(expr: F1LAE, funDefs: FunDefs, env: Env): Int = expr match {  
  case Num(n) =>  
    n  
  case Add(lhs, rhs) =>  
    interp(lhs, funDefs, env) + interp(rhs, funDefs, env)  
  case Sub(lhs, rhs) =>  
    interp(lhs, funDefs, env) - interp(rhs, funDefs, env)  
  
  case Let(id, expr, body) =>  
    val newEnv = env + (id -> interp(expr, funDefs, env))  
    interp(body, funDefs, newEnv)  
  
  case Id(name) =>  
    env(name)  
  
  case App(fun, arg) => funDefs(fun) match {  
    case FunDef(param, body) =>  
      val funEnv = env + Map(param -> interp(arg, funDefs, env))  
      interp(body, funDefs, funEnv)  
  }  
}
```

Static
Scoping!

FIRST CLASS FUNCTIONS

Outline

- Implementing first-order functions
- Environments (static vs. dynamic scoping)
- **About first-class functions**
- Functional decomposition and recursion patterns
- Implementing first-class functions

Abstracting over Computations

```
def filter[A, B] (relOp: (A, B) => Boolean, b: B, list: List[A]): List[A] =  
  list match {  
    case Nil => Nil  
    case x :: xs =>  
      val filteredRest = filter(relOp, b, xs)  
      if (relOp(x, b)) x :: filteredRest  
      else filteredRest  
  }
```

relOp parameter
stands for any
relational operation
to apply

```
def <(a: Int, b: Int) = a < b  
def >(a: Int, b: Int) = a > b  
  
def below(thres: Int, l: List[Int]) = filter(<, thres, l)  
def above(thres: Int, l: List[Int]) = filter(>, thres, l)  
  
println(below(4, List(1, 2, 3, 4, 5)))  
println(above(4, List(1, 2, 3, 4, 5)))  
  
def squaredGt(x: Int, c: Int) = x * x > c  
println(filter(squaredGt, 10, List(1, 2, 3, 4, 5)))
```

Functions that Return Functions

Expressions in functional languages (Scheme, Haskell, Scala) can evaluate to functions.

The body of a function is an expression \rightarrow a function can return a function.

Especially useful when produced function “remembers” arguments ...

```
def add(x: Int) = {  
  def xAdder(y: Int) = x + y  
  xAdder _  
}
```

Functions that Return Functions

```
def filter2[A, B](relOp: (A, B) => Boolean) = {
```

```
  def innerFilter(b: B, list: List[A]): List[A] = list match {  
    case Nil => Nil  
    case x :: xs => {  
      val filteredRest = innerFilter(b, xs)  
      if (relOp(x, b)) x :: filteredRest  
      else filteredRest  
    }  
  }
```

```
  innerFilter _  
}
```

filter2 consumes **relOp**, defines **innerFilter**, and returns it.
innerFilter remembers **relOp**

Anonymous Functions

- If auxiliary functions are only used as arguments to some abstract function **f**, we use an inner **def**.

```
case class Person(name: Symbol)

def findAnon(list: List[Person], name: Symbol) = {
  def hasName(p: Person, name: Symbol) = p.name == name
  filter(hasName, name, list)
}
```

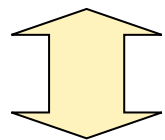
- Anonymous functions provide a short-hand for this frequent use of locally defined functions → **lambda**-expressions

Syntax of lambdas in Scala

```
(<var> ... <var>) => <exp>
```

- the sequence of variables in parentheses are the function's parameters
- component on the right-hand side is the function's body

```
(x-1 ... x-n) => exp
```



```
def plus4(x: Int) = x + 4  
def plus4 = (x: Int) => x + 4
```

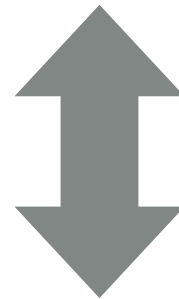
```
def aNewName (x-1 ... x-n) exp  
aNewName
```

aNewName may
not occur in **exp**

Syntax of lambdas in Scala

```
case class Person(name: Symbol)

def findAnon(list: List[Person], name: Symbol) = {
  def hasName(p: Person, name: Symbol) = p.name == name
  filter(hasName, name, list)
}
```



```
def findAnon(list: List[Person], name: Symbol) = {
  filter((p: Person, name: Symbol) => { p.name == name }, name, list)
}
```

Functions that Return Functions

```
def filter2[A, B](relOp: (A, B) => Boolean) = {
```

```
  def innerFilter(b: B, list: List[A]): List[A] = list match {  
    case Nil => Nil  
    case x :: xs =>  
      val filteredRest = innerFilter(b, xs)  
      if (relOp(x, b)) x :: filteredRest  
      else filteredRest  
  }
```

```
  innerFilter _  
}
```

Can we use an anonymous function here?

Outline

- Implementing first-order functions
- Environments (static vs. dynamic scoping)
- About first-class functions
- **Functional decomposition and recursion patterns**
- Implementing first-class functions

Recursion Operators

- Many recursive programs share a **common pattern of recursion**.
- Repeating the same patterns again and again is tedious, time consuming, error prone.
- Such repetition can be avoided by introducing special *recursion operators*.
- Recursion operators encapsulate common patterns allowing to concentrate on parts that are different for each application.

Any Problems with this `list-of-squares`?

```
def listOfSquares(list: List[Int]): List[Int] = list match {  
  case Nil => Nil  
  case x :: xs => square(x) :: listOfSquares(xs)  
}
```

- This definition draws attention to the **element-by-element processing** of the input list.
- The operation applied to elements (`square`) is hard-coded.
- How the result is constructed from the input is hard-coded (coupled to a particular implementation of the list).

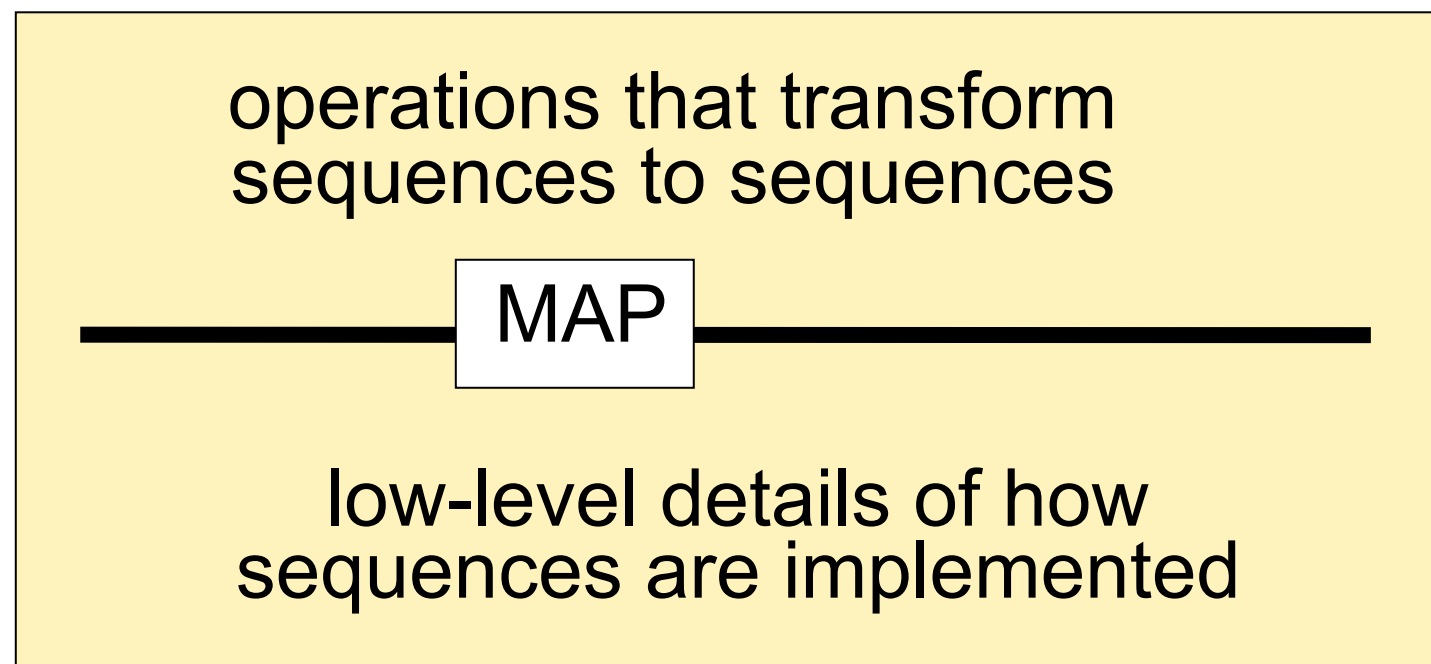
What about this Definition?

```
def map[A, B](f: A => B, list: List[A]): List[B] = list match {  
  case Nil => Nil  
  case x :: xs => f(x) :: map(f, xs)  
}  
  
def listOfSquares2(list: List[Int]) = map(square, list)
```

This version emphasizes
squaring as a
transformation of a list to
another list

map : an Abstraction Barrier

- **Abstraction barrier**: **map** supports a **higher-level of abstraction** by isolating the implementation of procedures that transform lists from the details of how list elements are extracted and combined.



- One can vary the implementation of the list independent of the mapping function applied to each element.
- **map** encapsulates a **recursion pattern**.

Quiz

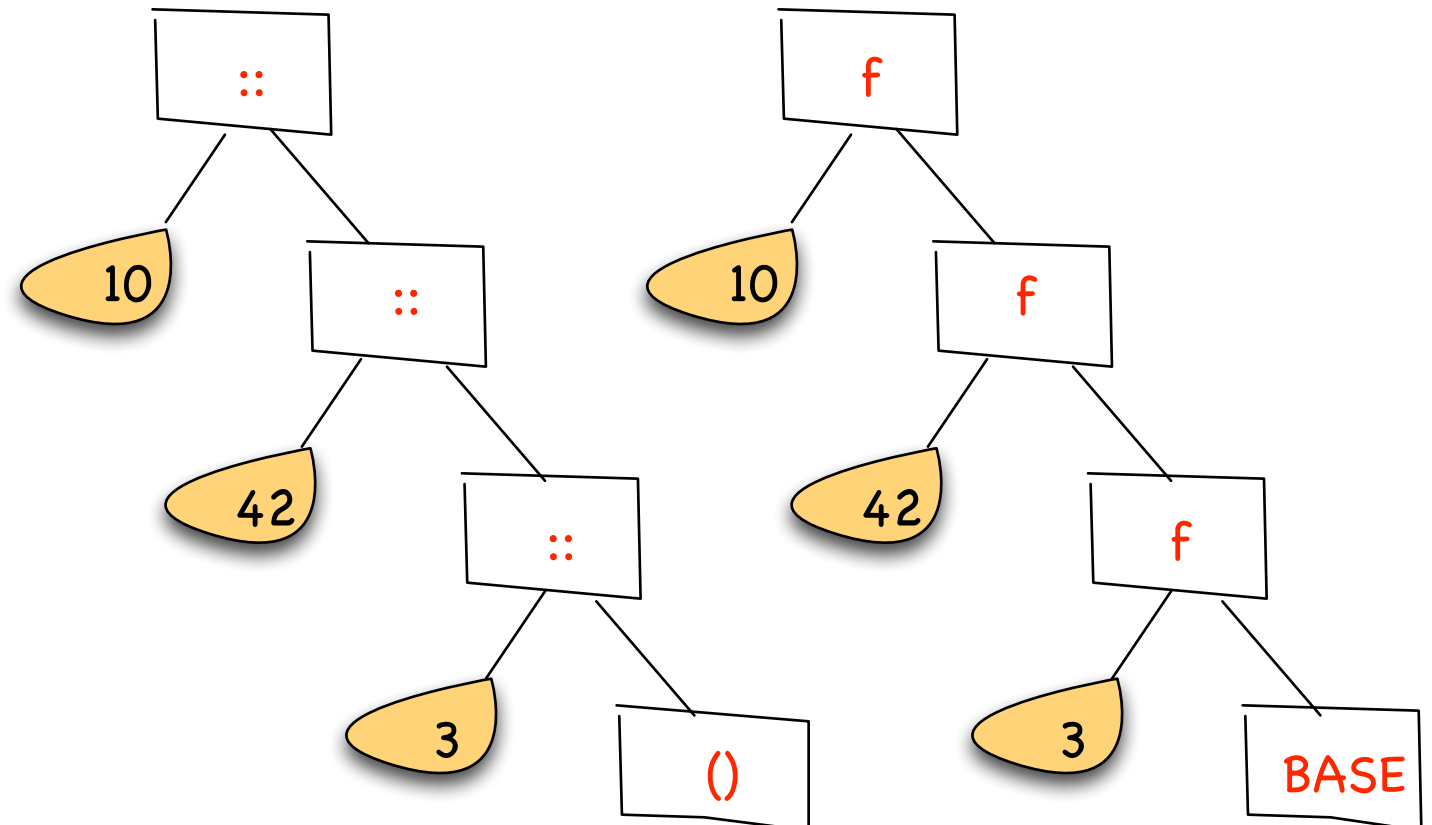
Can you think of map as an instance of another recursion operator?

The `fold`^{*)} Recursion Operator

^{*)} aka **reduce**, **accumulate**, **compress** or **inject**

In Scala:

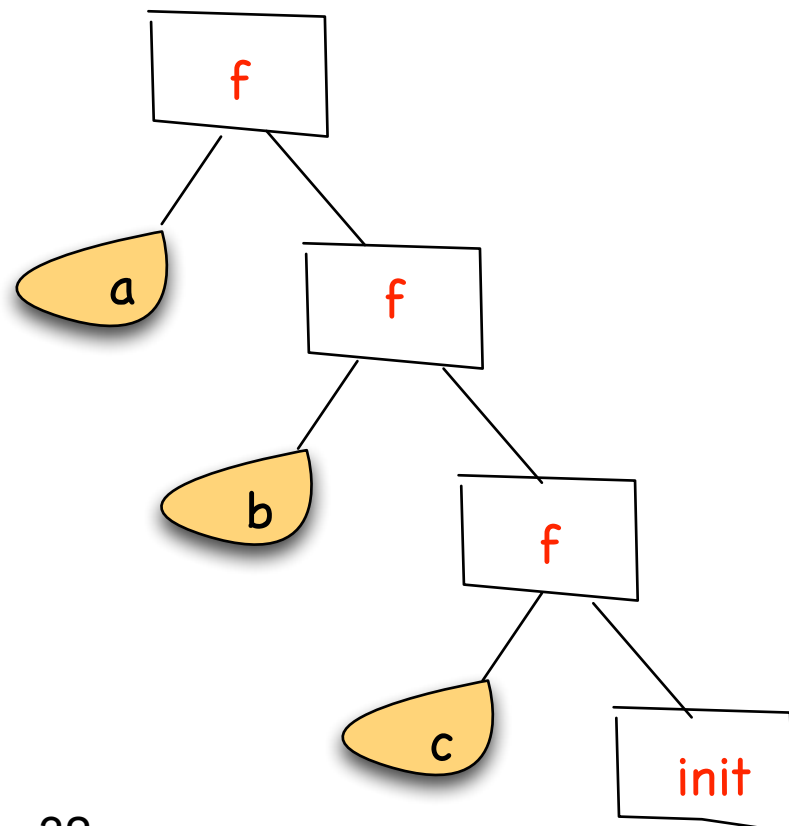
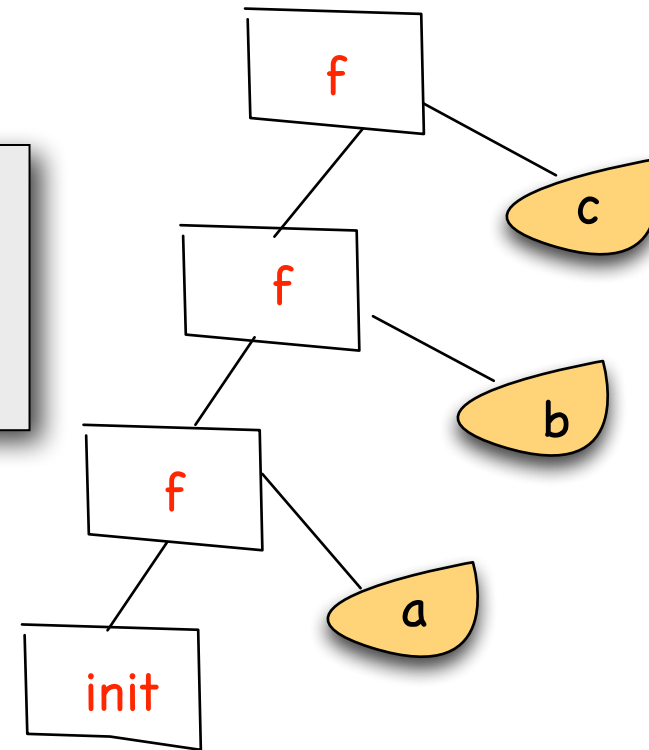
```
def fold[A, B](init: B, combine: (A, B) => B, l: List[A]): B =  
  l match {  
    case Nil => init  
    case x :: xs => combine(x, fold(init, combine, xs))  
  }
```



Folding in Scala

Two predefined operations: `foldLeft`, `foldRight`:

```
List(a, b, c).foldLeft(init)(f)
==
f(f(f(init a), b), c)
```



```
List(a, b, c).foldRight(init)(f)
==
f(a, f(b, f(c, init)))
```


Some Instantiations of `fold`

```
package templates

object FoldingTemp extends App {
  def summing(list: List[Int]): Int = ???
  def product(list: List[Int]): Int = ???
  def length[A](list: List[A]): Int = ???
  def reverse[A](list: List[A]): List[A] = ???
  def myMap[A, B] (f: A => B, list: List[A]): List[B] = ???
  def myFilter[A](p: A => Boolean, list: List[A]): List[A] = ???

  println(summing(List(1, 4, 5)))
  println(product(List(1, 4, 5)))
  println(length(List(1, 4, 5)))
  println(reverse(List(1, 4, 5)))
  println(myMap( (x:Int) => 2*x, List(1, 4, 5) ))
  println(myFilter( (x: Int) => x > 2, List(1, 4, 5) ) )
}
```

When Process Structure Follows Data ...

```
case class Tree(value: Int, branches: List[Tree])

def sumOfOddSquares(forest: List[Tree]): Int = forest match {
  case Nil => 0
  case x :: xs => x match {
    case Tree(value, branches) =>
      val restSum = sumOfOddSquares(branches) + sumOfOddSquares(xs)
      if (isOdd(value)) square(value) + restSum
      else restSum
  }
}
```

```
def evenFibs(n: Int): List[Int] = {
  def process(k: Int): List[Int] = {
    if (k > n) Nil
    else {
      val f = fib(k)
      if (isEven(f)) f :: process(k + 1)
      else process(k + 1)
    }
  }
  process(0)
}
```

Do you recognize any similarity between these two processes?

Focusing on the Process Structure...

- sumOfOddSquares:

- **enumerates** the leaves of a tree
- **filters** them to select the odd ones
- **process** each of the selected ones by squaring
- **folds** the result list using +, starting with 0

generic filter

generic map

generic fold

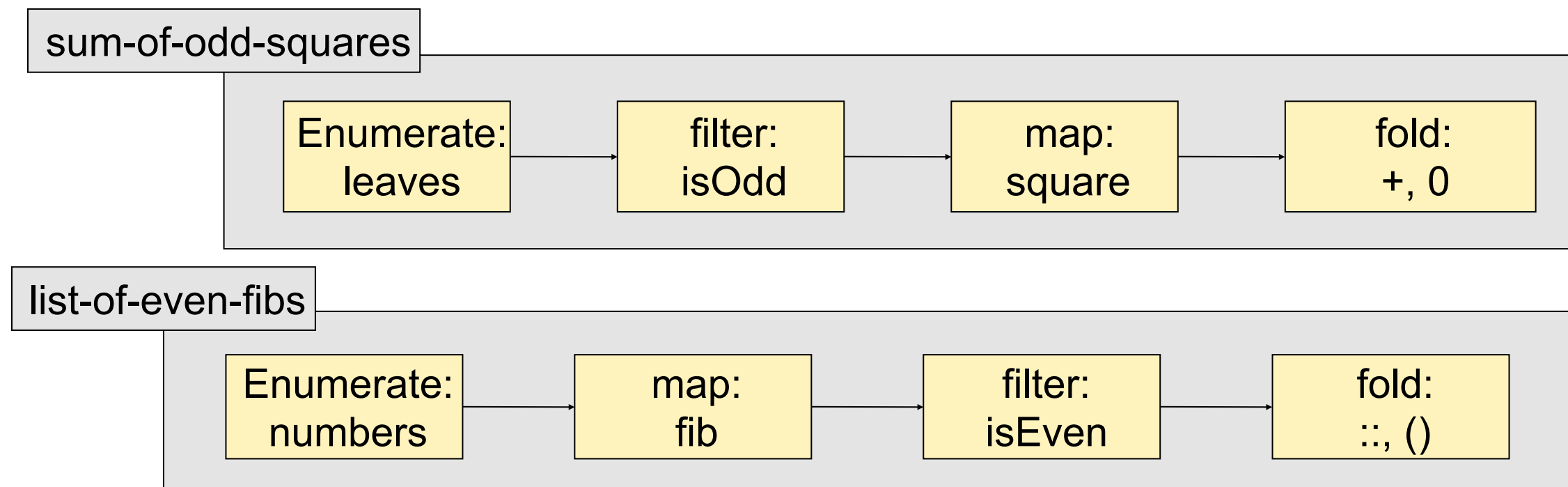
- evenFibs:

- **enumerates** the integers from 0 to n
- **process** each integer by computing its fibonacci
- **filters** them to select the even ones
- **folds** the result list using ::, starting with empty list

generic map

generic filter

generic fold



Advantages of Functional Decomposition

Functional decomposition supports **program designs that are modular** by combining independent components, instantiated from recursion operators that can be provided in a library of standard components.

Modular construction is a powerful strategy for **controlling complexity** in engineering design

Outline

- Implementing first-order functions
- Environments (static vs. dynamic scoping)
- About first-class functions
- Functional decomposition and recursion patterns
- **Implementing first-class functions**

FLAE: A Language with First-Class Functions

- **Function definitions are expressions in FLAE:**
 - They can appear everywhere within other compound expressions
 - Functions are values just like numbers
 - Function values can be passed to and returned by other functions
- Examples of FLAE programs:

```
{{fun {x} {+ x x}} 3}
```

```
{let {inc {fun {x} {+ x 1}}}  
  {+ {inc 4} {inc 5}}}
```

```
{let {x 3} {fun {y} {+ x y}}}
```

Concrete and Abstract Syntax of FLAE

```
<F1LAE> ::= <num>
          | {+ <F1LAE> <F1LAE>}
          | {- <F1LAE> <F1LAE>}
          | {let {<id> <F1LAE>} <F1LAE>}
          | <id>
          | {<id> <F1LAE>}
```

How will the concrete and abstract syntax of FLAE differ?

```
sealed abstract class F1LAE
case class Num(n: Int) extends F1LAE
case class Add(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Sub(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Let(name: Symbol, namedExpr: F1LAE, body: F1LAE) extends F1LAE
case class Id(name: Symbol) extends F1LAE
case class App(funName: Symbol, arg: F1LAE) extends F1LAE
```

Concrete and Abstract Syntax of FWAE

- Concrete syntax

```
<FWAE> ::= <num>
        | {+ <FLAE> <FLAE>}
        | {- <FLAE> <FLAE>}
        | {let {<id> <FLAE>} <FLAE>}
        | <id>
        | {fun {<id>} <FLAE>}
        | {<FLAE> <FLAE>}
```

- Abstract syntax

```
sealed abstract class FLAE
  case class Num(n: Int) extends FLAE
  case class Add(lhs: FLAE, rhs: FLAE) extends FLAE
  case class Sub(lhs: FLAE, rhs: FLAE) extends FLAE
  case class Let(name: Symbol, namedExpr: FLAE, body: FLAE) extends FLAE
  case class Id(name: Symbol) extends FLAE
  case class Fun(param: Symbol, body: FLAE) extends FLAE
  case class App(funExpr: FLAE, arg: FLAE) extends FLAE
```


Interpreting FLAE

- We first implement an interpreter for FLAE that employs substitution
 - To facilitate the comparison to LAE and F1LAE
 - To define a „reference“ specification of the semantics
- Next, we will replace substitutions with environments

Interpreting FLAE

- What does the interpreter produce, i.e., what are the **values** of FLAE?
- How do we represent functions inside the interpreter?
- What needs to be done to turn LAE into FLAE?

FLAE with Substitution

1. Extend the class of values

- Interpreters so far produced Scala integers.
- The interpretation of FLAE expressions can also produce functions
- First try: Return values of FLAE are **Num** or **Fun**

2. Add clauses for function **definition** expressions to the substitution procedure and the interpreter

3. Modify substitution and interpretation of **application** expressions

<fill in holes in the templates ...>

FLAE with Substitution

Substitution function

```
...  
case class Let(name: Symbol, namedExpr: FLAE, body: FLAE) extends FLAE  
case class Id(name: Symbol) extends FLAE  
case class Fun(param: Symbol, body: FLAE) extends FLAE  
case class App(funExpr: FLAE, arg: FLAE) extends FLAE
```

```
case Fun(param, body) =>  
  if (param == substId)  
    Fun(param, body)  
  else  
    Fun(param, subst(body, substId, value))  
  
case App(funExpr, argExpr) =>  
  App(subst(funExpr, substId, value), subst(argExpr, substId, value))
```

```
abstract class Value  
case class VNum(n: Int) extends Value  
case class VFun(param: Symbol, body: FLAE) extends Value
```

FLAE with Substitution

Interpreter function

```
def interp(expr: FLAE): Value = expr match {  
    ...  
    case Fun(param, body) => VFun(param, body)  
    case App(funExpr, argExpr) => interp(funExpr) match {  
        case VFun(param, body) => interp(subst(body, param, argExpr))  
        case v1 => error(s"Expected function value but got $v1")  
    }  
}
```

Interpreter

- This was the FLAEImmediateSubstInterp

FWAE with Environments

- Replace substitution with environment lookup
- Preserve static scoping

Question

To avoid dynamic scoping, in F1WAE with environments we used an empty environment for evaluating function applications.

Can we apply the same trick here?

FWAE with Environments

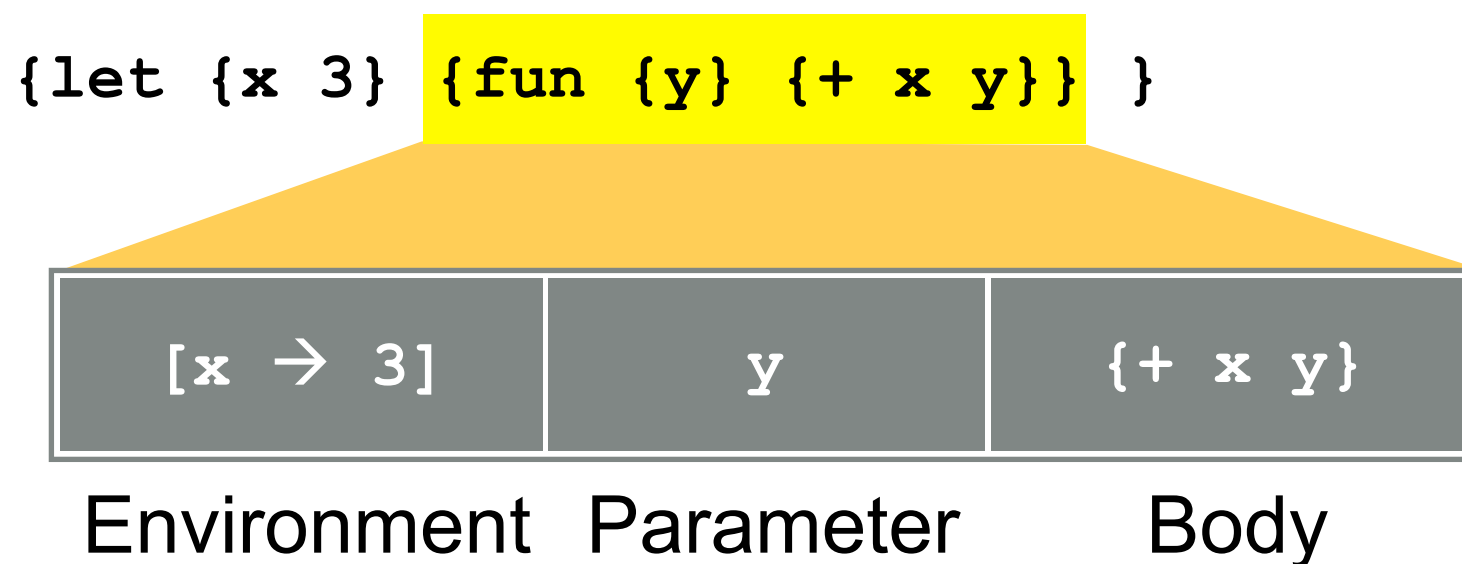
- If uncertain, compare manual evaluations of the following...

```
interp(App('f', Num(4)),  
        Map('f -> FunDef('y, Add('x 'y))))
```

```
interp(Let('x, Num(3),  
           App(Fun('y, Add('x 'y)) Num(4))),  
        Map())
```


FWAE with Environments

- Remember environment at function-definition time, so that it can be used for binding free identifiers in function bodies when applying the function.
- Function + environment = **closure**



`type Env = Map[Symbol, Value]`

FWAE with Environments

1. Define a data type for representing FWAE values (closures)
2. Modify the definition of environments to use the new values
3. Modify the interpreter to:
 - use the environment for deferring substitutions
 - return closures as the result of evaluating function definitions
 - use the environment of the closure returned by evaluating the function sub-expression when evaluating function applications

FWAE with Environments

```
def interp(expr: FWAE, env: Env = Map()): Value = expr match {  
  
  ...  
  
}
```

<fill in holes in the template ...>

Interpreter

FLAStaticInterpr.scala

Quiz

Do we really need **let** expressions in a language with first-class functions?

Quiz

Do we really need **let** expressions in a language with first-class functions?

No. Why?

Hint: we have two binding constructs, but we only need one. Which one is more powerful?

“Fun” is more powerful than “let” (because of parameters, which can’t be expressed in “let”) so:

Substitution function

encode {let {x 4} {+ x 3}} as {{fun {x} {+ x 3}} 4}

=> Lambda Calculus

“Closures” for Java

Closures are one of the features added in Java 8.

Main driving force: concurrent, parallel programming on multi-cores.

```
public interface Callable<V> {  
    V call();  
}
```

Are Java closures real closures?

- The compiler provides convenient syntax
- They are objects that implement the Callable interface
- Local variables referenced from a lambda expression must be **final** or **effectively final***

*A variable or parameter whose value is never changed after it is initialized is effectively final [up to what the compiler can infer]