

Programming Languages

A Journey into Abstraction and Composition

Programming Language Semantics and Design

Prof. Dr. Guido Salvaneschi

p.g Programming
Group

School of Computer Science



University of St.Gallen

WHY STUDYING PROGRAMMING LANGUAGES?

Why Programming Languages?

Programming Languages are a powerful tool to control software complexity

“Better” languages increase our ability to deal with complex problems

- Better languages ‘capture’ a solution more directly.

```
i = 1
TEST: if i < 4
      then goto BODY
      else goto END
BODY: print(i)
      i = i + 1
      goto TEST
END:
```

```
i = 1
while ( i < 4 ) {
    print(i)
    i = i + 1
}
```

The Goal of this Course

Provide insights into the core concepts of PLs

Concepts/abstractions:

- How does it work?
- How can we implement it?
- How do they interact with each other?

Which concepts should we use

- E.g., can we build a full language from a minimal “core”?

Concepts/abstractions are the building blocks of new languages

What do we Study?

Every programming language consists of four elements:

- 1. Syntax:** structure of programs
- 2. Semantics:** meaning associated to syntax
- 3. Libraries:** reusable computations
- 4. Idioms** used by programmers of that language

Can you make examples for each of those?

Which of these elements is the most important for the study of PLs?

What do we Study?

Which of the code fragments is most similar:

$a[25] + 5$ (Java)

$(+ (\text{vector-ref } a\ 25)\ 5)$ (Scheme)

$a[25] + 5$ (C)

But semantics can be different.
E.g., memory writes and array boundaries...

How to Express Semantics?

Informal specs and language surveys

Formal specs: operational, denotational, axiomatic semantics,
... not the focus of this course

Interpreter semantics (cousin of operational semantics)

- Explain a language by writing an interpreter for it
- By telling the computer how to 'execute a concept' we thoroughly understand it ourselves

We'll interleave language surveys and interpreters

- We will also peek at formal semantics
- Inductive versus deductive learning

Interpreter Semantics

An interpreter that defines a language cannot be “wrong”.

- It defines the meaning

Assigning ‘+’ another meaning than addition is not wrong,
at most it is unconventional

- Can you make an example?

Only, when given another specification of a language, one can speak about the correctness of the interpreter relative to the specification

- Do two virtual machines implement the same “meaning”?
- Think of JavaScript in different browsers



Overview

Arithmetic expressions

Naming

First-order functions

High-order and first-class functions

Recursion

State

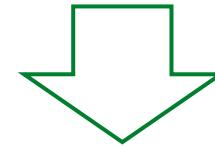
Objects and classes

Memory management

Type systems

...

LANGUAGES WITH
INCREASING COMPLEXITY



HOW TO MODEL THEM

DOES IT MATTER?

The Problem of Semantics

```
i = 5
f(i++, --i)
```

The Problem of Semantics

```
i = 5
f(i++, i--)
```

Which values does the f function receive?

- The ++ operator returns the value and then increments it
- Option 1: left-to-right evaluation 5, 5
- Option 2: right-to-left evaluation 4, 4

The semantics is in the compiler!

More about semantics

```
[5, 12, 9, 2, 18, 1, 25].sort();
```

```
→ [1, 12, 18, 2, 25, 5, 9]
```

```
[5, 12, 9, 2, 18, 1, 25].sort(function(a, b){  
    return a - b;  
});
```

What does this program print?

```
var a = 1;  
  
function four() {  
    if (true) {  
        var a = 4;  
    }  
    alert(a);  
}  
  
four()
```

```
> parseInt(0.5)
<- 0

> parseInt(0.05)
<- 0

> parseInt(0.005)
<- 0

> parseInt(0.0005)
<- 0

> parseInt(0.00005)
<- 0

> parseInt(0.000005)
<- 0

> parseInt(0.0000005)
<- 5
```

SCALA IN A NUTSHELL

Driving Forces for Scala's Design

PLs for component software should be **scalable**

- The same concepts describe small and large parts
- Rather than adding lots of primitives, focus on abstraction, composition, and decomposition

Unification of OO and functional programming can provide scalable support for components

Adoption is key to test the design
→ Interoperability with Java

Scala's Adoption

LinkedIn



foursquare®

NETFLIX

yammer™ SIEMENS



Atlassian

tomtom®

xerox



UBS

the guardian



tumblr.

airbnb



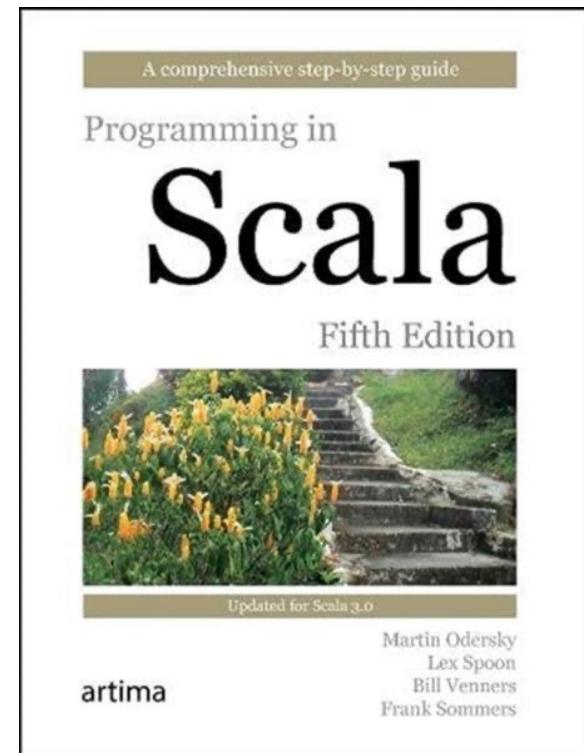
The Scala Language

We will present only as much Scala as needed.

To learn Scala, you can find many books and online courses (not needed for this class).

Note, Scala 3 is changing rapidly

- It is very likely that new features are be added soon



Start Right Away!

You will need both:

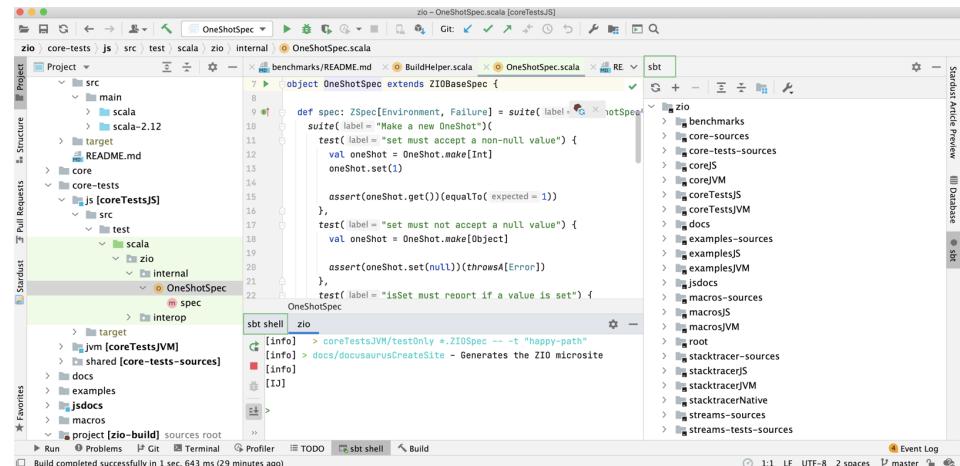
SBT

- www.scala-sbt.org
- Command line tool
- Create projects
- Upload projects to Coursera



Scala IDE

- Eclipse: scala-ide.org (deprecated)
- IntelliJ (recommended)



Scala Example

```
object Timer {  
    def oncePerSecond(callback: () => Unit) = {  
        while (true) { callback(); Thread sleep 1000 }  
    }  
    def timeFlies() = {  
        println("time flies like an arrow...")  
    }  
    def main(args: Array[String]) = {  
        oncePerSecond(timeFlies)  
    }  
}
```

What does this do?

Explain all features not available in Java used in the example.

Classes and Inheritance

```
class Complex(real: Double, imaginary: Double) {  
    def re = real  
    def im = imaginary  
  
    override def toString() =  
        "" + re + (if (im < 0) "" else "+") + im + "i"  
}  
  
object ComplexNumbers {  
    def main(args: Array[String]) = {  
        val c = new Complex(1.2, 3.4)  
        println(c.toString)  
        println("imaginary part: " + c.im)  
    }  
}
```

Explain all features not available in Java used in the example.



Algebraic Data Types (ADTs)

An ADT is a data type whose values are made up of

- a constructor name
- subterm values from other datatypes

```
Typename
= Con1  t_11 ... t_1k1      |
  Con2  t_21 ... t_2k2      |
    ...
  Con_n t_n1 ... t_nk_n
```

Pattern matching to:

- distinguish between values defined with different constructors of an ADT
- extract the subparts of a complex ADT

Case Classes for Algebraic Data Types (ADTs)

```
abstract class Tree  
  
case class Leaf(n: Int) extends Tree  
case class Node(left: Tree, right: Tree) extends Tree
```

Tree values:

```
Node(Leaf(3),Leaf(4))  
Node(Node(Leaf(3),Leaf(4)),Leaf(7))
```

Case Classes vs. “normal” Classes (1)

Factory methods are automatically available for case classes:

Leaf(3) instead of **new Leaf(3)**

Instances of case classes can be **decomposed** into their parts (constructor parameters) through pattern matching

Pattern Matching on Case Classes

Basic idea:

- Attempt to match a value to a series of patterns
- As soon as a pattern matches, extract and name various parts of the value,
- Evaluate code that makes use of these named parts

```
abstract class Tree
case class Leaf(n: Int) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

def sum(t: Tree): Int = t match {
  case Leaf(n) => n
  case Node(left, right) => sum(left) + sum(right)
}
```

Question

```
abstract class Tree
case class Leaf(n: Int) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

def sum(t: Tree): Int = t match {
  case Leaf(n) => n
  case Node(left, right) => sum(left) + sum(right)
}
```

Do we really need case classes?

Couldn't we define sum as a method of Tree and its subclasses?

Wouldn't this be more OO conform?

A SIMPLE LANGUAGE: ARITHMETIC EXPRESSIONS

Modeling Syntax

Different notations for the idealized action of adding the idealized numbers (represented) by “3” and “4”:

- | | | |
|------------------|------------------------|--------|
| – 3 + 4 | (infix) | Java |
| – 3 4 + | (postfix) | FORTH |
| – (+ 3 4) | (parenthesized prefix) | Scheme |

Ignoring details of concrete syntax, the essence is a tree (AST) ...

So the first question to answer in modeling languages is how to represent ASTs.

Syntax

Concrete syntax

- What the programmer writes
- Comments, multiple spaces, newlines, ...

Abstract syntax

- Internal representation of the syntax
- Smaller to make automatic processing (e.g., type checking) and reasoning easier.
- Example: Arithmetic Expressions

```
<AE> ::= <num>
         | { + <AE> <AE> }
         | { - <AE> <AE> }
```

Case Classes for ASTs

AST for arithmetic expressions

```
sealed abstract class Expr

case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr
```

Values of this data type:

```
Add(Num(3),Num(4))
Add(Sub(Num(3),Num(4)),Num(7))
```

Template for Our Interpreters

```
def interp(expr: Expr): Int = expr match {  
  case Num(n) => ???  
  case Add(lhs, rhs) => ???  
  case Sub(lhs, rhs) => ???  
}
```

What goes
into “???”

Template for Our Interpreters

```
def interp(expr: Expr): Int = expr match {  
  case Num(n) => ???  
  case Add(lhs, rhs) => ???  
  case Sub(lhs, rhs) => ???  
}
```

What goes
into “???”

```
def interp(expr: Expr): Int = expr match {  
  case Num(n) => n  
  case Add(lhs, rhs) => interp(lhs) + interp(rhs)  
  case Sub(lhs, rhs) => interp(lhs) - interp(rhs)  
}
```

Demo

The AE interpreter

Programming Languages

A Journey into Abstraction and Composition

Let, Substitution and Functions

Prof. Dr. Guido Salvaneschi

p.g Programming
Group

School of Computer Science



University of St.Gallen

Overview

Arithmetic expressions

Naming

First-order functions

High-order and first-class functions

Recursion

State

Objects and classes

Memory management

Type systems

...

A LANGUAGE WITH NAMES

Next: LAE – a Language with Names

Motivation: reduce repetitions by introducing **identifiers** (not yet variables!)

Example program 1:

```
let y = (5 + 10) in
  y + y

= (5 + 10) + (5 + 10)
```

Example program 2:

```
let y = (5 + 10) in
  let x = 20 in
    (x + y)

= 20 + (5 + 10)
```

LAE: Abstract Syntax

```
<AE> ::= <num>
        | {+ <AE> <AE>}
        | {- <AE> <AE>}
```

Extend
with “let”

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

LAE: Concrete Syntax

Which implementation steps are needed?

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr

???
```

Extend
with “let”

LAE: Concrete syntax

Which implementation steps are needed?

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr
???
```

Extend
with “let”

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr

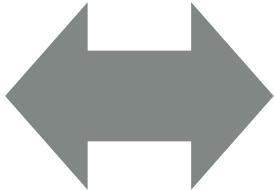
case class Let(name: Symbol, namedExpr: Expr, body: Expr) extends Expr
case class Id(name: Symbol) extends Expr
```

Semantics

We need to give a semantics to let expressions

We do so via the concept of substitution

Semantics
of let



Substitution

Defining Substitution

Wanted: A definition of the process of substitution

Here is one:

Definition (Substitution):

*To substitute identifier i in e with expression v ,
replace all identifier sub-expressions of e
named i with v .*

Try it out with the following LAE expressions:

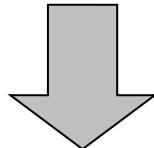
1. `let x = 5 in x + x`

2. `let x = 5 in x + (let x = 3 in x)`

Defining Substitution

```
1. let x = 5 in x + x
```

```
2. let x = 5 in x + (let x = 3 in x)
```



```
1. 5 + 5
```

```
2. 5 + (let 5 = 3 in 5)
```

This is not even syntactically legal!
-> it does not respect the BNF and
it would be rejected by a parser

Defining Substitution

Definition (Binding Instance):

A binding instance of an identifier is the instance of the identifier that gives it its value. In LAE, the <id> position of a 'let' is the only binding instance.

Definition (Scope)

The scope of a binding instance is the region of program in which instances of the identifier refer to the value bound by the binding instance.

Definition (Bound Instance)

An identifier is bound if it is contained within the scope of a binding instance of its name.

Definition (Free Instance)

An identifier not contained in the scope of any binding instance of its name is said to be free.

Defining Substitution

```
let x = 5 in
  x + (let x = 3 in
        x + x)
```

```
5 + (let x = 3 in
      5 + 5)
```

```
5 + (5 + 5)
```

```
15
```

What can go wrong here?

-> We do not
respect scoping

Defining Substitution

Definition (Binding Instance):

A binding instance of an identifier is the instance of the identifier that gives it its value.
In LAE, the `<id>` position of a 'let' is the only binding instance.

Definition (Scope)

The scope of a binding instance is the region of program in which instances of the identifier refer to the value bound by the binding instance.

Definition (Bound Instance)

An identifier is bound if it is contained within the scope of a binding instance of its name.

Definition (Free Instance)

An identifier not contained in the scope of any binding instance of its name is said to be free.

```
let x = 5 in
  x + (let x = 3 in
        x + y)
```



Defining Substitution

Definition (Substitution):

To substitute identifier i in e with expression v , replace all **free** instances of i in e with v .

This definition is implicitly using a notion of **scope**

Substitute only in the scope of the identifier

An inner binding for the same name introduces a **new scope**.

The scope of the outer binding is **shadowed** or **masked** by the inner binding.

Substituting the inner x is wrong.

The previous definition
respected binding instances,
but not their scope.



Calculating LAE Expressions

```
def interp(expr: Expr): Int = expr match {  
  case Num(n) => n  
  case Add(lhs, rhs) => calc(lhs) + calc(rhs)  
  case Sub(lhs, rhs) => calc(lhs) - calc(rhs)  
  case Let(boundId, namedExpr, boundExpr) => ???  
  case Id(name) => ???  
}
```

Use a “subst” function

```
def subst(expr: LAE, substId: Symbol, value: LAE)
```

```
def interp(expr: Expr): Int = expr match {  
  case Num(n) => n  
  case Add(lhs, rhs) => interp(lhs) + interp(rhs)  
  case Sub(lhs, rhs) => interp(lhs) - interp(rhs)  
  case Let(boundId, namedExpr, boundExpr) => {  
    interp(subst(boundExpr, boundId, Num(interp(namedExpr))))  
  }  
  case Id(name) => sys.error("found unbound id " + name)  
}
```

Calculating LAE Expressions

Any identifier in the scope of a let-expr is replaced with a value when the calculator encounters that identifier's binding instance.

- There are no free instances of the identifier given as an argument left in the result.
- Subst replaces identifiers with values **before** the calculator ever “sees” them.

The calculator can't assign a value to a free identifier

- If a free identifier is found, the calculator halts with an error

Calculating LAE Expressions: The Substitution Function

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {  
  case Num(n) => ???  
  case Add(lhs, rhs) => ???  
  case Sub(lhs, rhs) => ???  
  
  case Let(boundId, namedExpr, boundExpr) => ???  
  
  case Id(name) => ...  
}
```

Calculating LAE Expressions: The Substitution Function

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {  
  case Num(n) => expr  
  case Add(lhs, rhs) => Add(subst(lhs, substId, value), subst(rhs, substId, value))  
  case Sub(lhs, rhs) => Sub(subst(lhs, substId, value), subst(rhs, substId, value))  
  
  case Let(boundId, namedExpr, boundExpr) => ???  
  
  case Id(name) => ...  
}
```

Calculating LAE Expressions: The Substitution Function

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {  
  case Num(n) => expr  
  case Add(lhs, rhs) => Add(subst(lhs, substId, value), subst(rhs, substId, value))  
  case Sub(lhs, rhs) => Sub(subst(lhs, substId, value), subst(rhs, substId, value))  
  
  case Let(boundId, namedExpr, boundExpr) => {  
    val substNamedExpr = subst(namedExpr, substId, value)  
    Let(boundId, substNamedExpr, subst(boundExpr, substId, value))  
  }  
  
  case Id(name) => ...  
}
```

What is wrong with this one?

Calculating LAE Expressions

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {
  case Num(n) => expr
  case Add(lhs, rhs) => Add(subst(lhs, substId, value), subst(rhs, substId, value))
  case Sub(lhs, rhs) => Sub(subst(lhs, substId, value), subst(rhs, substId, value))

  case Let(boundId, namedExpr, boundExpr) => {
    val substNamedExpr = subst(namedExpr, substId, value)
    if (boundId == substId)
      Let(boundId, substNamedExpr, boundExpr)
    else
      Let(boundId, substNamedExpr, subst(boundExpr, substId, value))
  }

  case Id(name) => {
    if (substId == name) value
    else expr
  }
}
```



Two Substitution Regimes

Eager substitution:

avoids re-computing
the same value
several times.

1

```
{let {x {+ 5 5}} {let {y {- x 3}} {+ y y}}}  
= {let {x 10} {let {y {- x 3}} {+ y y}}}  
= {let {y {- 10 3}} {+ y y}}  
= {let {y 7} {+ y y}}  
= {+ 7 7}  
= 14
```

Lazy substitution:

the expression may be
evaluated multiple times.

2

```
{let {x {+ 5 5}} {let {y {- x 3}} {+ y y}}}  
= {let {y {- {+ 5 5} 3}} {+ y y}}  
= {+ {- {+ 5 5} 3} {- {+ 5 5} 3}}  
= {+ {- 10 3} {- {+ 5 5} 3}}  
= {+ {- 10 3} {- 10 3}}  
= {+ 7 {- 10 3}}  
= {+ 7 7}  
= 14
```



Two Substitution Regimes: Questions

1. Which one have we implemented?

```
def interp(expr: Expr): Int = expr match {  
  ...  
  case Let(boundId, namedExpr, boundExpr) => {  
    interp(subst(boundExpr, boundId, Num(interp(namedExpr))))  
  }  
  ...  
}
```

2. Our example suggests that the eager regime generates an answer in fewer steps. Is this always true?

```
{let {x (+ 5 5)}  
  {let {y 4} {+ y y}}}
```

3. Do the two regimes always produce the same result for LAE?

```
{let {x (+ z 4)}  
  {let {y 4} {+ y y}}}
```

Demo

The LAE interpreter



Programming Languages

A Journey into Abstraction and Composition

Basic Functions

Prof. Dr. Guido Salvaneschi

p.g Programming
Group

School of Computer Science



University of St.Gallen

A Note on Abstraction

A powerful concept in Computer Science

- “Hide details”
- “Make something generic w.r.t. something else”
- “Parametrize over something”

Let expressions

- Parametrize an expression based on another expression

Functions

- Parametrize over data (only?)

Generics

- Parametrize over types

The F1LAE Language

A language with very simple functions

- Function **applications** (function calls) are expressions
- Function **definitions** are not expressions

Separate definitions from expressions

- Similar to the C language
- Predefined functions given to interpreter as an argument

(No first-class/higher-order functions – we will discuss when this means exactly in the next lecture)

Concrete & Abstract Syntax for F1LAE

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

how does the concrete syntax change?

```
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
```

???

how does the abstract syntax change?

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

The F1LAE Language

Function applications (calls) can be nested.

Function applications can be bound to names in let expressions.

Function Id can be bound to names in let expressions.

We cannot define new functions in F1LAE.

- Function definitions are not expressions.
- There is no expression of the kind {fun ...}

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

Function Definitions in F1LAE

Cannot define functions in F1LAE

- More strict than necessary, functions could be declared at top level, more similar to C

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 fun definitions**

Example Interpreter Calls

scala.Symbol: tick (‘) followed by identifier

```
interp(
  App('f, 10),
  Map(
    'f -> FunDef('x, App('g, Add('x, 3))),
    'g -> FunDef('y, Sub('y, 1)))
```

Note: Scala 3 has removed symbols

- We use String, instead, in the interpreter implementations

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

```
interp(
  App("f", 10),
  Map(
    "f" -> FunDef("x", App("g", Add("x", 3))),
    "g" -> FunDef("y", Sub("y", 1)))
```

Discussion

What is needed to interpret F1LAE expressions?

<Interpreter>

F1LAE 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)
  }
}
```

Demo

The F1LAEImmediateSubstInterp interpreter

Discussion on Scoping

What is the result of $\{f\ 10\}$ where $\{f\ x\} = \{g\ \{+ x\ 3\}\}$ and $\{g\ y\} = \{- y\ 1\}$

```
val funDefs = Map(  
  'f -> FunDef('x, App('g, Add('x, 3))),  
  'g -> FunDef('y, Sub('y, 1))  
)  
  
interp(App('f, 10), funDefs)
```

Some open issues:

- Should **f** be able to invoke **g** or should the invocation fail because **g** is defined after **f**?
- What if there are multiple bindings for the same name?
- If a function can invoke every defined function, it can also invoke itself. Do we have recursion in F1LAE?

ENVIRONMENTS: STATIC VS. DYNAMIC SCOPING

Let and Substitution

In `{let {x e} t}` we immediately replace free identifiers `x` in expression `t` with the value expression `e` evaluates to

`id` expressions left in `t` after substitution denote free identifiers

If the interpreter encounters an `id` expression → error

Quiz

Do you see any problems
with this strategy?

Let and Substitution

The current solution is **slow**

Consider the following sequence of evaluation steps:

```
{let {x 3}  
{let {y 4}  
{let {z 5}  
{+ x {+ y z}}}}}  
  
= {let {y 4}  
  {let {z 5}  
    {+ 3 {+ y z}}}}  
  
= {let {z 5}  
  {+ 3 {+ 4 z}}}  
  = {+ 3 {+ 4 5}}}
```

Substitution is applied three times: once for each let

The program has size n , measured in abstract syntax tree nodes.

Each substitution sweeps the whole rest of the program.

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]
```

Here is F1LAE. How Does it Change?

```
def interp(expr: F1LAE, funDefs: FunDefs): 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)
  }
}
```

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)  
  }  
}
```

Scoping

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

Using environments, the result is 5.

- The result of applying a function may change depending on the calling context?
- This contradicts our mathematical understanding of a function.

What is the answer when using
F1LAE with substitution?

For F1WAE with substitution n would not be substituted and it would be an error

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!

Programming Languages

A Journey into Abstraction and Composition

First-class Functions

Prof. Dr. Guido Salvaneschi

p.g Programming
Group

School of Computer Science



University of St.Gallen

Functions: Terminology

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
 - Can be returned by other functions, stored in data structures etc.
- No first-class functions => functions can only be defined in designated regions 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
- Parameterize computations over data

First-class Functions

Some languages achieve some kind of higher-orderless without first class funs.

Functions pointers (C, C++)

- Once can pass around the pointer

Objects with a “call” method

- “A function is an object with a single method”
- One can pass around the callable object

Eval: interprets a string on the fly

- Slow, not secure, not safe, makes code unreadable and hard to maintain.
- One can do pretty much everything.

```
var result;
function Sum(val1, val2)
{
    return val1 + val2;
}
eval("result = Sum(5, 5);");
alert(result);
```

```
var str = '({ "firstName": "Bill",
              "lastName": "Gates" })';
var obj = eval(str);
obj.firstName; // Bill
```

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.

Also: The body of a function is an expression
→ a function can return a function

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

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

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)
}
```

FUNCTIONAL DECOMPOSITION AND RECURSION PATTERNS

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
- They allow one to concentrate on parts that are different for each application

Any Problems with this list-of-squares?

```
def list0fSquares(list: List[Int]): List[Int] = list match {  
  case Nil => Nil  
  case x :: xs => square(x) :: list0fSquares(xs)  
}
```

This definition includes the **element-by-element processing** of the input list.

The operation applied to elements (**square**) is hard-coded.

The construction of the result from the input is hard-coded

- Coupled to a particular implementation of the list, using the :: operator

What about this Definition?

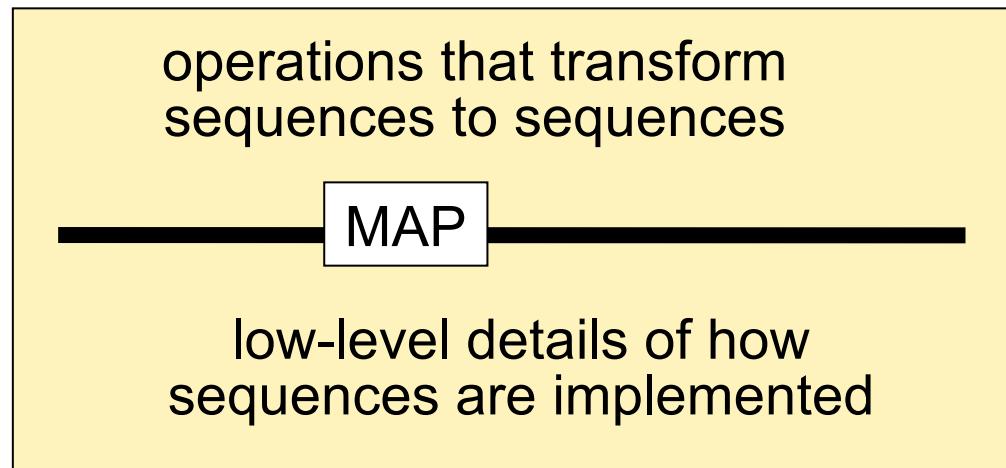
```
def map[A, B](f: A => B, list: List[A]): List[B] = list match {  
  case Nil => Nil  
  case x :: xs => f(x) :: map(f, xs)  
}  
  
def listOfSquares2(list: List[Int]) = map(square, list)
```

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**

- isolating the implementation of procedures that transform lists from the details of how list elements are extracted and combined.
- One can vary the implem. of the list independent of the function applied to each element.



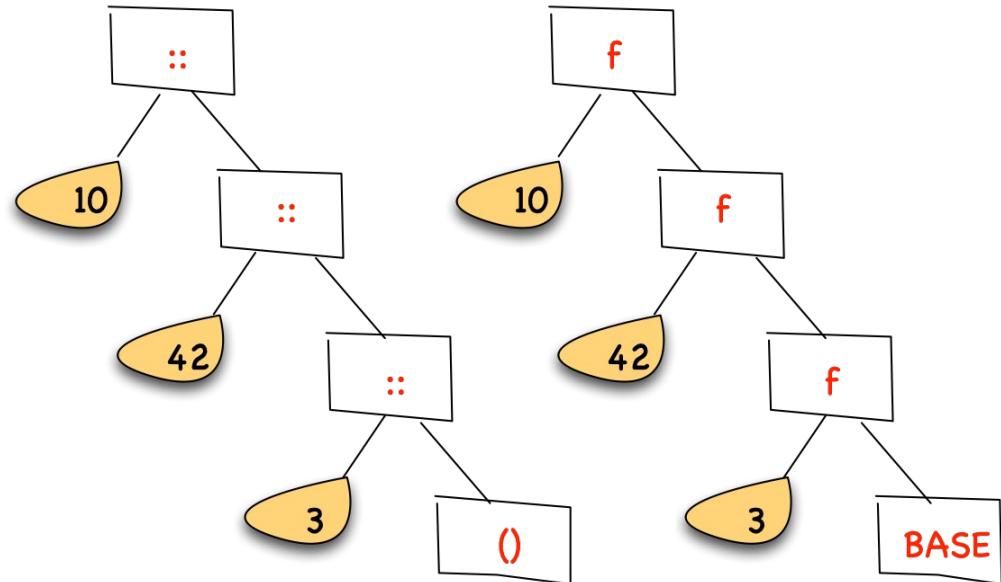
map encapsulates a **recursion pattern**.

The fold* Recursion Operator

*aka **reduce**, **accumulate**, **compress** or **inject**

In Scala:

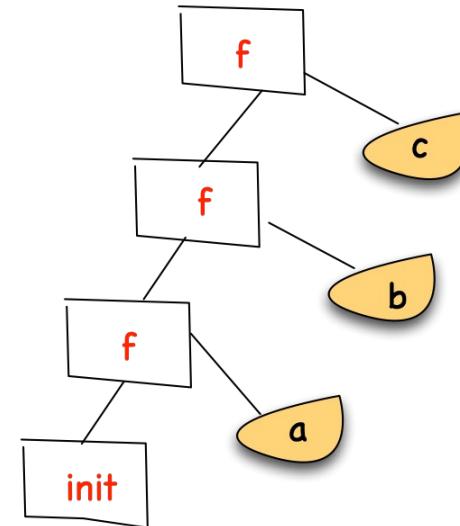
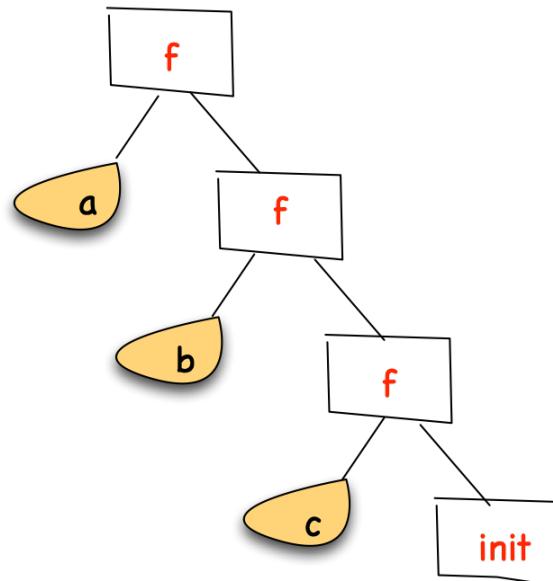
```
def fold[A, B](init: B, combine: (A, B) => B, l: List[A]): B =  
  l match {  
    case Nil => init  
    case x :: xs => combine(x, fold(init, combine, xs))  
  }
```



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) ) )
}
```

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?

What needs to be done to turn LAE into FLAE?



FLAE with Substitution: Interpreter Implementation Steps

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

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(funExp: FLAE, arg: FLAE) extends FLAE
```

Static
scoping!

```
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
```

Interpreter: FLAEImmediateSubstInterp

FLAE with Substitution

```
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")  
  }  
}
```

Example: Static vs Dynamic Scoping

```
let d = 2 in  
  let f = fun x { x + d } in  
    let d = 1 in  
      f 2
```

The result is different:

- Static (lexical) scoping: evaluates to 4
- Dynamic scoping: evaluates to 3

Early versions of LISP, APL, 1 PostScript, TeX, Perl, early versions of Python had dynamic scoping

FLAE with Environments

Replace substitution with environment lookup

Preserve static scoping

Question

To avoid dynamic scoping, in F1LAE with environments, we used an empty environment for evaluation function applications.

Can we apply the same trick here?

FLAE 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())
```

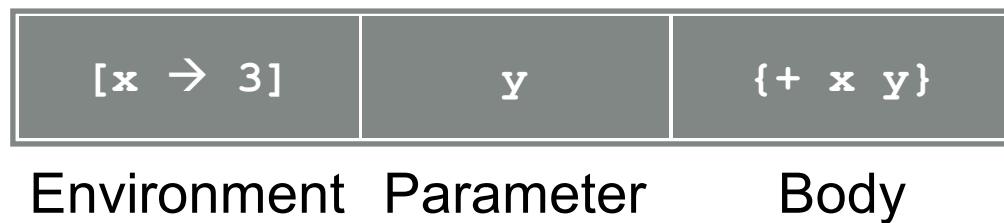
FLAE with Environments

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]

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



FLAE with Environments

1. Define a data type for representing FLAE 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

FLAE with Environments

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

Interpreter: FLAESTaticInterpreter

Quiz

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

Lambda Calculus

Originally developed by Church in the 1930s, to study, with others, the foundations of mathematics via universal models of computation:

- Lambda calculus (Church)
- Turing Machines (Turing)
- Primitive recursive functions (Kleene, Gödel)

It is considered the foundation of modern functional programming and we look at it from this perspective

Outdated version with different precedence

Lambda Calculus

(abstraction over application)

Pleas refer to the alternative on the next slide where Application has precedence over abstraction

x	<i>variable</i>
$\lambda x. M$	abstraction – i.e. a function
$M N$	application – i.e. a function call

Very simple model

- All functions are anonymous (i.e. they are Lambdas)
- Lexical coping
- Semantics by substitution
- In the original model, only functions and variables (numbers, etc, can be encoded)

$$(\lambda x. x) \ 3 \\ \rightarrow 3$$

$$(\lambda x. x) \ y \\ \rightarrow y$$

$$(\lambda x. x) \ \lambda y. y \\ \rightarrow \lambda y. y$$

$$(\lambda x. x+1) \ 7 \\ \rightarrow 7+1 \\ \rightarrow 8$$

$$(\lambda x. (\lambda y. x+y)) \ 7 \ 8 \\ \rightarrow (\lambda y. (7+y)) \ 8 \\ \rightarrow 7+8 \\ \rightarrow 15$$

$$(\lambda x. x \ x) \ \lambda y. y \\ \rightarrow \lambda y. y \ \lambda y. y \\ \rightarrow \lambda y. y$$



Lambda Calculus

x	<i>variable</i>
$\lambda x.M$	<i>abstraction – i.e. a function</i>
$M N$	<i>application – i.e. a function call</i>

Very simple model

- All functions are anonymous (i.e. they are lambdas)
- Lexical coping
- Semantics by substitution
- In the original model, only functions and variables (numbers, etc, can be encoded)

$$(\lambda x.x) \ 3 \\ \rightarrow 3$$

$$(\lambda x.x) \ \lambda y.y \\ \rightarrow \lambda y.y$$

$$(\lambda x. \lambda y.x+y) \ 7 \ 8 \\ \rightarrow (\lambda y.7+y) \ 8 \\ \rightarrow 7+8$$

$$(\lambda x.x \ x) \ \lambda y.y \\ \rightarrow (\lambda y.y) \ \lambda y.y \\ \rightarrow \lambda y.y$$

$$(\lambda x.x) \ y \\ \rightarrow y$$

$$(\lambda x.x+1) \ 7 \\ \rightarrow 7+1 \\ \rightarrow 8$$

$$\rightarrow 15$$

Example: Python is Statically Scoped

```
def foo():
    a = 10
    print(a)

foo()
> 10
```

```
a = 10

def foo():
    print(a)

foo()
```

```
def foo():
    print(a)
```

```
def bar():
    a = 10
    foo()
```

```
bar()
```

1. a is in (the function) scope
2. a is in the global scope, hence it is in scope
3. a is not in scope

```
> NameError: name
  'a' is not defined
```

Example: Python is Statically Scoped

In Python there are two scopes: global and local (to a function)

- Note: indentation per se does not create a new scope
- There is no *block* scoping

```
def foo():
    if True:
        a = 10
    print(a)
```

```
foo()
```

```
> 10
```

Example: Closures in Python

Closures are often used to implement some form of delayed evaluation

- All function objects have a `__closure__` attribute
- It returns a tuple of cell objects if it is a closure function
- The cell object has the attribute `cell_contents` which stores the closed value.

```
from urllib.request import urlopen
def page(url):
    def get():
        return urlopen(url).read()
    return get

>>> url1 = page("http://www.google.com")
>>> url2 = page("http://www.bing.com") >>> url1
<function page.<locals>.get at 0x10a6054d0>
>>> url2
<function page.<locals>.get at 0x10a6055f0>

>>> gdata = url1() # Fetches http://www.google.com
>>> bdata = url2() # Fetches http://www.bing.com
```

```
>>> page.__closure__ # Returns None since not a closure
>>> url1.__closure__
(<cell at 0x10a5f1250: str object at 0x10a5f3120>,)
```

```
>>> url1.__closure__[0].cell_contents
'http://www.google.com'
>>> url2.__closure__[0].cell_contents
'http://www.bing.com'
```

