Concepts of Programming Languages

This Course in a Nutshell

Dr. Sebastian Erdweg
Dr. Guido Salvaneschi
Prof. Dr. Mira Mezini

Software Technology Group
TU Darmstadt | FB Informatik

Outline

- Why study programming languages?
- What exactly will we study and how?

- Excursus: Basics of Scala for Java Programmers

- Modeling rudimentary languages to start with
  - A language for arithmetic
  - A language with names

- Tentative preview of language concepts we will study
- Few organizational data

Programming Languages (PLs) Are ...

- ... **a powerful instrument to control complexity in design**
  - A means for instructing a computer to perform tasks
  - Frameworks within which we organize ideas about problem domains

- “Better” languages increase our ability to deal with complex problems by providing ways to describe things 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 insight into the core concepts of PLs

- What do the concepts mean
  - how do they work
  - how can we implement them
  - how do they interact with each other
- Which concepts should we use
- Concepts are the building blocks of new languages

Why Study PL Design?

Many reasons

- Better PLs enable more productive software development
- Insights into PL design applicable to API design
- Many computational domains can be considered PLs
  - Example: solving polynomial equations
  - Specify computation in some notation
  - Implementation embodies computation rules for polynomials
  - Report result in some notation
- Modeling techniques for computer systems and technologies for development of new programming languages overlap each other.

What Language Aspects 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

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

How to Study Semantics?

- Informal specs and language surveys

- Formal specs: operational, denotational, axiomatic semantics,
  … if at all, will only take a brief look in this course

- Interpreter semantics (cousin of operational semantics)
  - Explain a language by writing an interpreter for it
  - By telling the computer what it means to execute a concept we thoroughly understand it ourselves

- We'll interleave language surveys and interpreters
  - 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

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

Some Interpreter Terminology

- **Host language** (or **base language**):

the language, in which the interpreter is implemented.

In our case: Scala

- **Interpreted language** (or **object language**):

the language that the interpreter evaluates

  - We assume that we already understand the host language

- **meta-interpreter** or **meta-circular interpreter**:

  host language == interpreted language

Note…

Although we will write and study interpreters for particular languages that we define …

they have the fundamental structure of interpreters for any expression-oriented language that can be used to write programs on a sequential machine.

Administrative

- http://www.erdweg.org/teaching/14-copl/
- slash at the end is significant
- user: copl
- pass: smallstep

- http://weblab.tudelft.nl/edition/2dea221a-0ceb-4a5b-ad5b-99be4d7f3003
- Thursday 2pm until Thursday 10am

- Group exercises on Friday (see Tucan)

Outline

- Why study programming languages, what exactly we will study, and how?

- Excursus: Basics of Scala for Java Programmers

- Modeling rudimentary languages to start with
  – A language for arithmetic
  – A language with names

- Preview of the language concepts we will study

Driving Forces for Scala's Design

Goal: Better PL support for component software

Two hypotheses:

1. 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

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

Adoption is key for testing the hypotheses ==>
Scala interoperates with Java and .NET

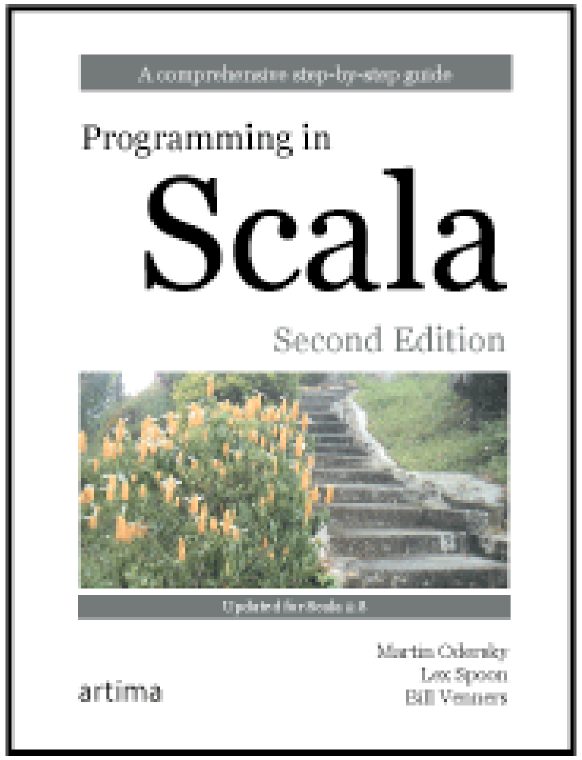
Scala’s Adoption

LinkedIn
foursquare
twitter
yammer
SIEMENS
NASA
Atlassian
TOMTOM
xerox
UBS
theguardian
SONY PICTURES imageworks

The Scala Language

- We will present only as much Scala as needed.
- For learning Scala, refer to books and online courses:

https:// www.coursera.org/course/progfun



[More books:](http://www.scala-lang.org/node/959)

http://www.scala-lang.org/node/959

[Quick entrance for Java Programmers:](http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html)

http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

Some Features of Scala

- Everything is an object
- Classes
- Case classes and pattern matching
- Implicit conversions

Everything is an Object

- Numbers are objects
- Functions are objects

Scala Numbers are Objects

```
1 + 2 * 3 / x
```

Infix operator notation for method calls

+, *, ... are identifiers referring to methods of number objects.

```
(1).+(((2).*(3))./(x))
```

Operator Notation

Operator notation is not limited to methods like + that look like operators in other languages.

You can use any method in infix operator notation.

```
val str = “hello”
str indexOf ‘o’
```

Prefix and postfix operator notations are supported too.

```
scala> -7
res12: Int = -7

scala> (7).unary_-
res13: Int = -7
```

```
scala> val s = "Hello World!"
s: String = Hello World!

scala> s toLowerCase
res16: String = hello world!

scala> s.toLowerCase
res17: String = hello world!
```

Scala Functions are Objects

```
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.

Is timeFlies really needed?

Some Features of Scala

- Everything is an object
- Classes and Inheritance
- Case classes and pattern matching
- Implicit conversions

Classes and Inheritance

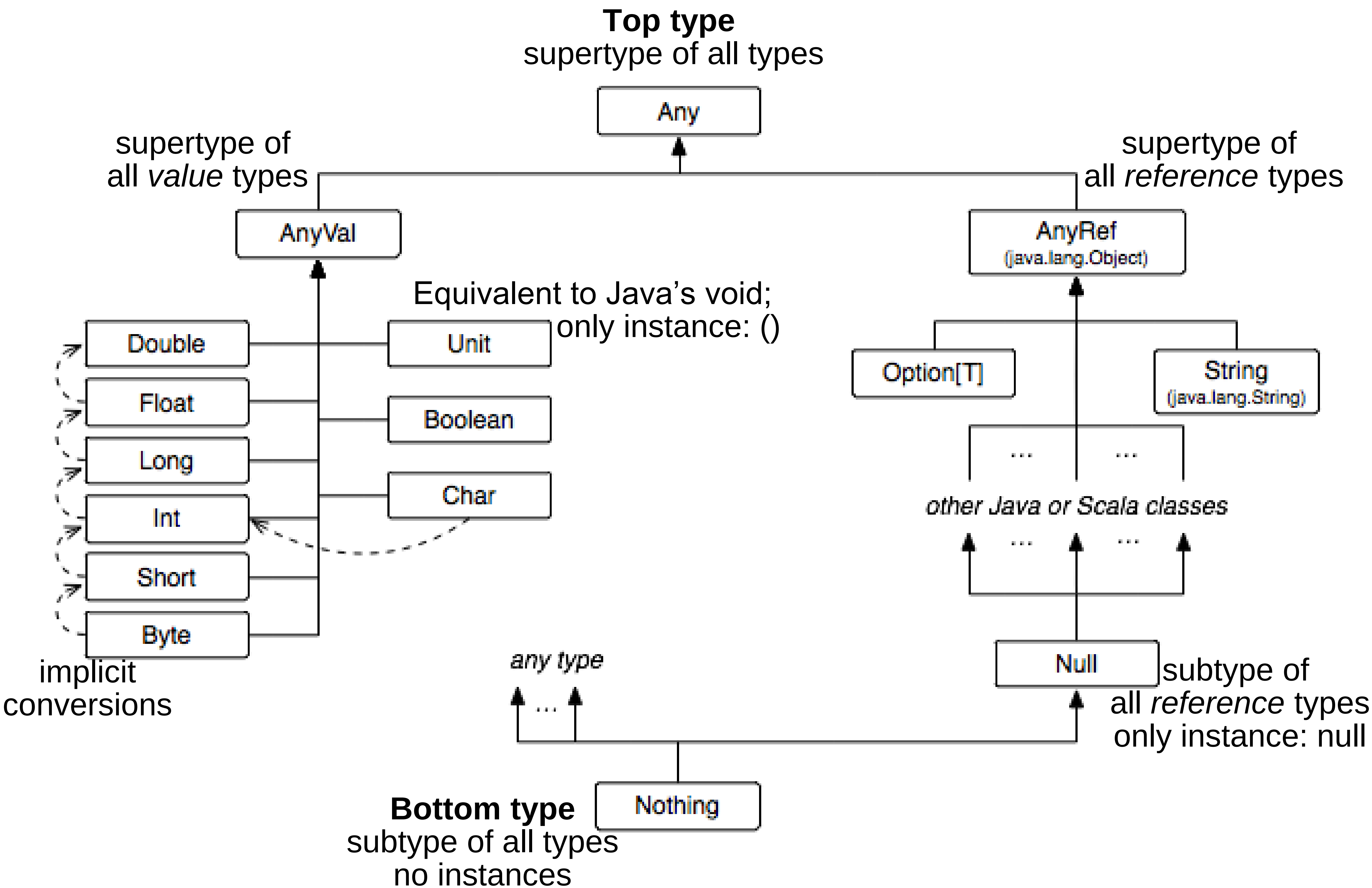
```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary

  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.

Scala's Type Hierarchy



Some Features of Scala

- Everything is an object
- Classes
- Case classes and pattern matching
- Implicit conversions

Algebraic Data Types (ADTs)

- An ADT is a data type whose values are data 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_nkn
```

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)

- Scala's case classes model regular, non-encapsulated ADT as objects and enable pattern matching on such objects

- Especially valuable when processing recursive (e.g., tree) structures

Case Classes for Arithmetic Expressions

```
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?

Case Classes vs. “normal” Classes (2)

- Getter functions automatically defined for constructor parameters

- Default definitions for methods equals and hashCode that <u>work on the structure of the instances and not on their identities</u>

Some Features of Scala

- Everything is an object
- Classes
- Case classes and pattern matching
- Implicit conversions

Implicit Conversions to Expected Types

- **Definitions**: Adapters of values of a certain available type to values of another required type.
- **Implicit**: Inserted automatically by the compiler into the program whenever this is needed to fix any type error

```
object Test {

  implicit def intToTree(n: Int): Tree = Leaf(n)

  def main(args: Array[String]) {
    val tree = Node(Node(3, 3), 2)
 }
}
```

intToTree converts an Int to a value of type Num

When we pass integers where a tree is expected, they are automatically converted by intToTree

Outline

- Why study programming languages, what exactly we will study, and how?
- Excursus: Basics of Scala for Java Programmers
- Modeling rudimentary languages to start with
  - A language for arithmetic
  - A language with names
- Preview of the language concepts we will study

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.

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) => ???
}
```

Next: WAE – a Language with Names

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

Example programs:

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

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

Substitution or „Name and Conquer“

Quiz: What implementation steps are needed?

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

```
def parse(prog: String): Expr = ...
```

```
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
```

The interpreter …

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 WAE expressions:

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

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

Defining Substitution

Definition (Binding Instance):

A binding instance of an identifier is the instance of the identifier that gives it its value. In WAE, the <id> position of a with 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

Definition (Substitution):

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

Calculating WAE Expressions

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

```
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(calc(namedExpr))))
 }
 case Id(name) => sys.error("found unbound id " + name)
}
```

Calculating WAE Expressions

```
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) => ...
 }
```

Two Substitution Regimes

Eager substitution (static and dynamic reduction): 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 (Static reduction): 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?
2. Our example suggests that the eager regime generates an answer in fewer steps. Is this always true?
3. Do the two regimes always produce the same result for WAE?

Outline

- Why study programming languages, what exactly we will study, and how?
- Excursus: Basics of Scala for Java Programmers
- Modeling rudimentary languages to start with
  - A language for arithmetic
  - A language with names
- Preview of the language concepts we will study

Tentative Overview of the Course Topics

- First-order and first-class functions
- Lazy evaluation
- Recursion
- State
- Continuations
- OO concepts
- Garbage collection
- Formal specification of semantics
- ...