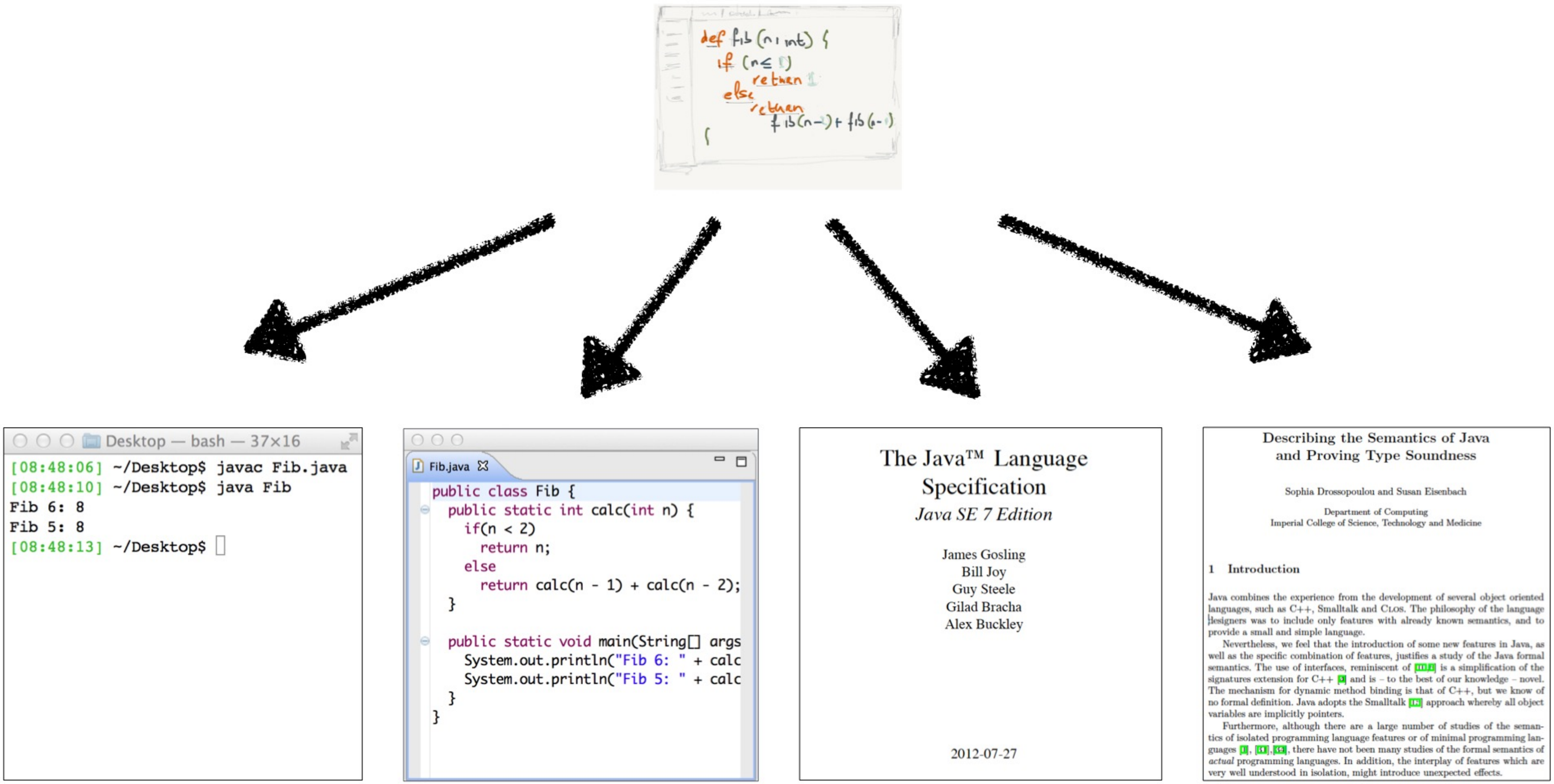
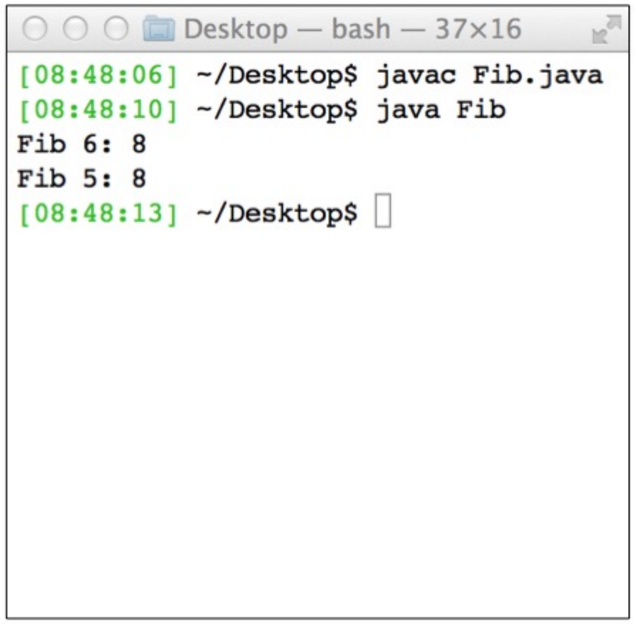
Declare Your Language

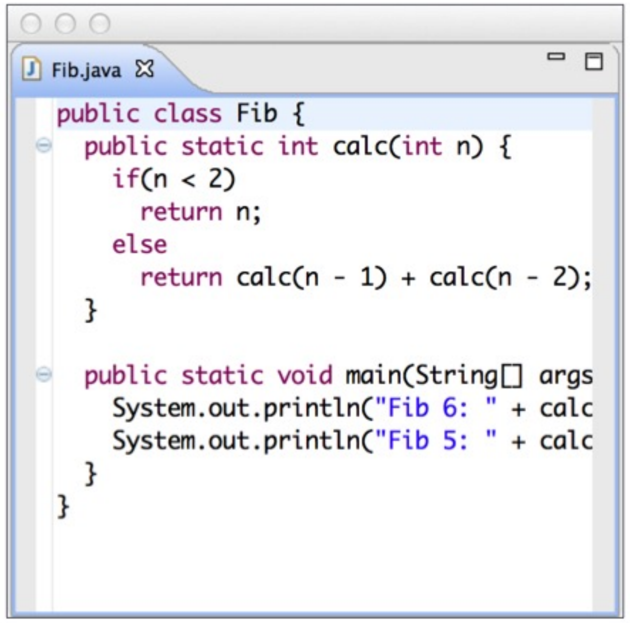
Eelco Visser

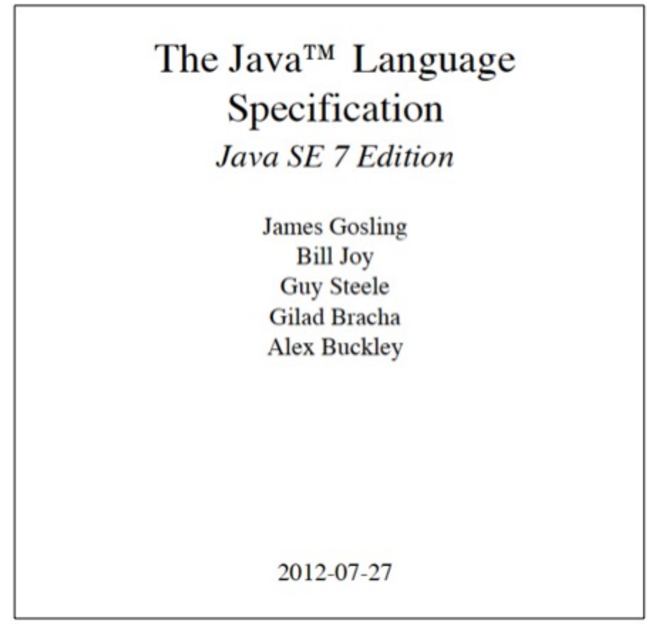
Dynamic Languages Symposium
October 27, 2015

Delft University of Technology


def fib (n, int) {
if (n ≤ 1)
return 1
else
return
fib(n-2) + fib(n-1)
}
Desktop — bash — 37×16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$
Fib.java
public class Fib {
public static int calc(int n) {
if(n < 2)
return n;
else
return calc(n - 1) + calc(n - 2);
}
public static void main(String[] args
System.out.println("Fib 6: " + calc
System.out.println("Fib 5: " + calc
}
}
The Java™ Language Specification
Java SE 7 Edition
James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
2012-07-27
Describing the Semantics of Java and Proving Type Soundness
Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
1 Introduction
Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and Clos. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.
Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,5] is a simplification of the signatures extension for C++ [4] and is – to the best of our knowledge – novel. The mechanism for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk [13] approach whereby all object variables are implicitly pointers.
Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], [31], [33], there have not been many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.






The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27


Describing the Semantics of Java and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine

1 Introduction

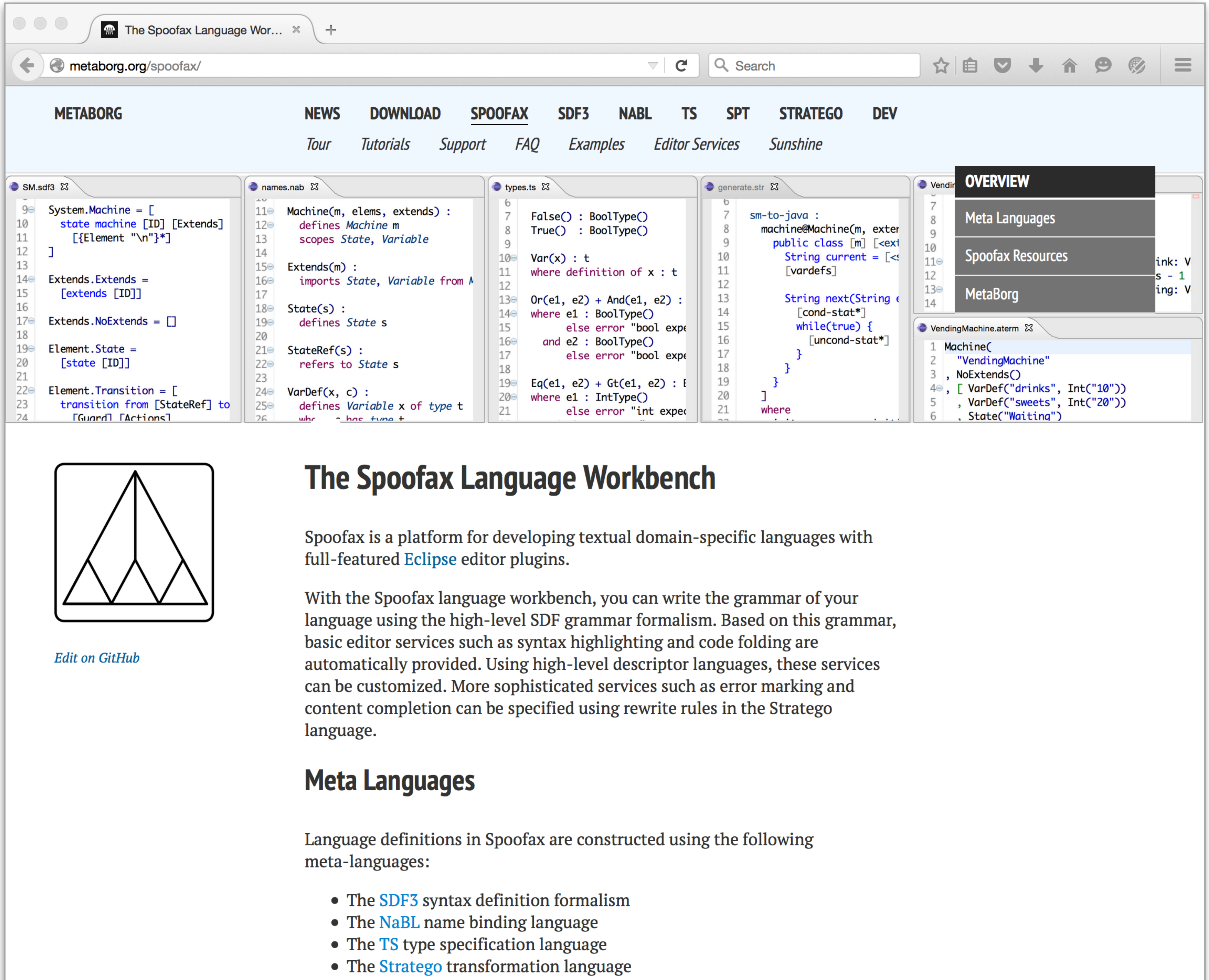
Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and CLOS. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,8] is a simplification of the signatures extension for C++ [9] and is – to the best of our knowledge – novel. The mechanism for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk [13] approach whereby all object variables are implicitly pointers.

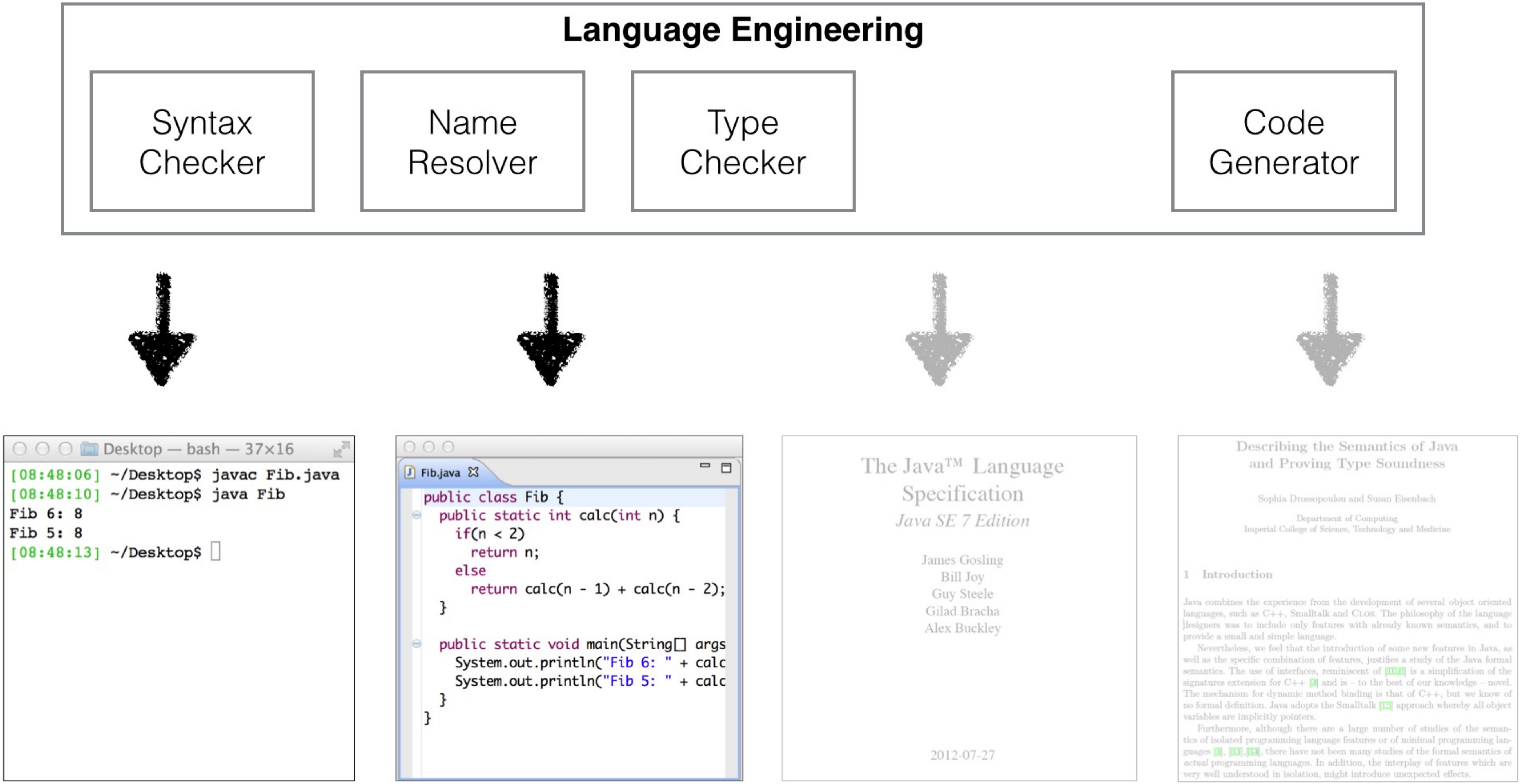
Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], [31],[34], there have not been many studies of the formal semantics of *actual* programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.

parser	parser error recovery syntax highlighting outline	syntax definition	abstract syntax
type checker	code completion navigation type checker	static semantics	type system
code generator interpreter	debugger	dynamic semantics	operational semantics
			type soundness proof

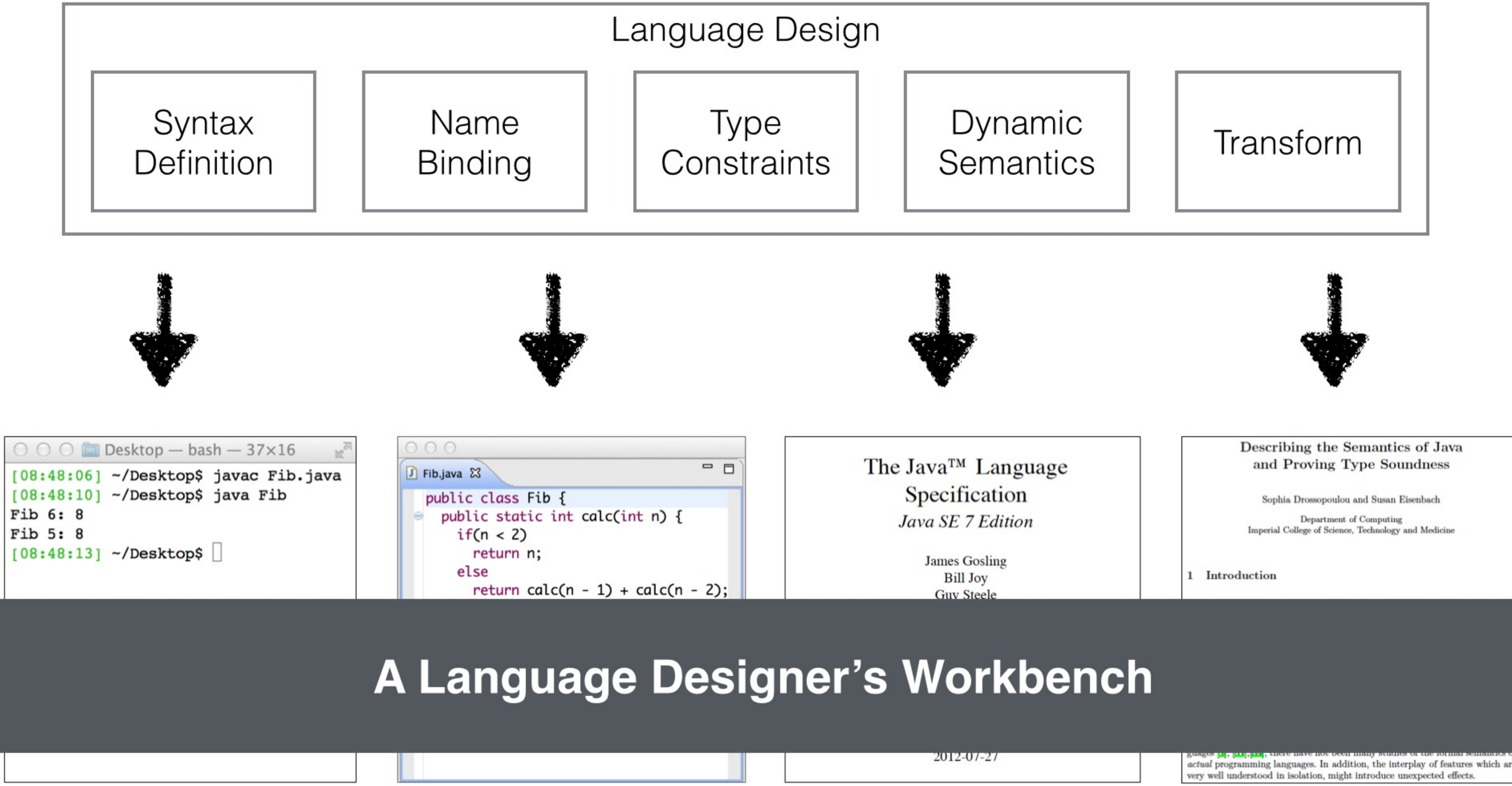
Spoofax Language Workbench



[Kats, Visser. OOPSLA 2010]

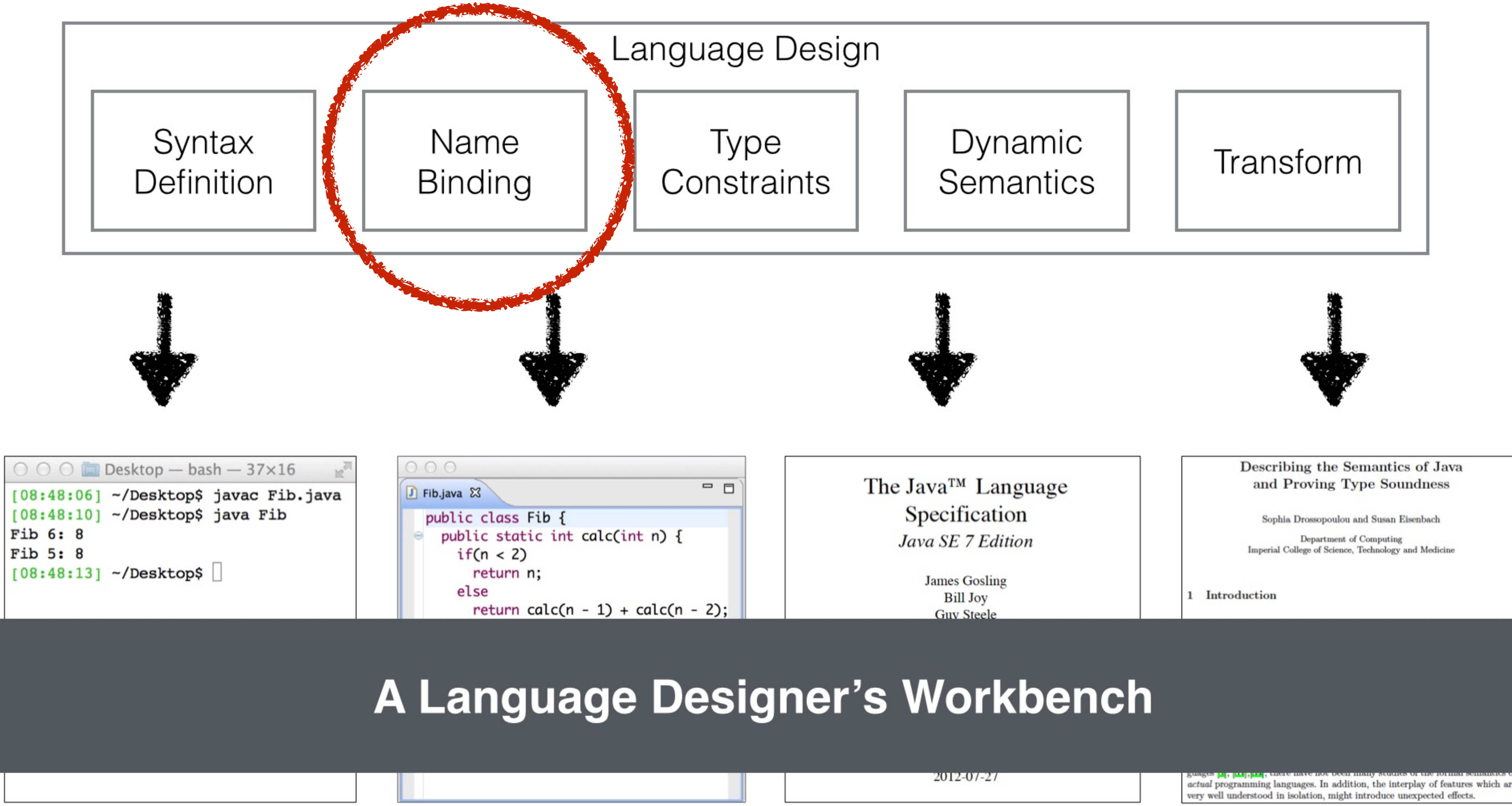

Language Engineering
Syntax Checker
Name Resolver
Type Checker
Code Generator
Desktop — bash — 37×16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$
Fib.java
public class Fib {
public static int calc(int n) {
if(n < 2)
return n;
else
return calc(n - 1) + calc(n - 2);
}
public static void main(String[] args
System.out.println("Fib 6: " + calc
System.out.println("Fib 5: " + calc
}
}
The Java™ Language Specification
Java SE 7 Edition
James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
2012-07-27
Describing the Semantics of Java and Proving Type Soundness
Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
1 Introduction

A Language Designer's Workbench



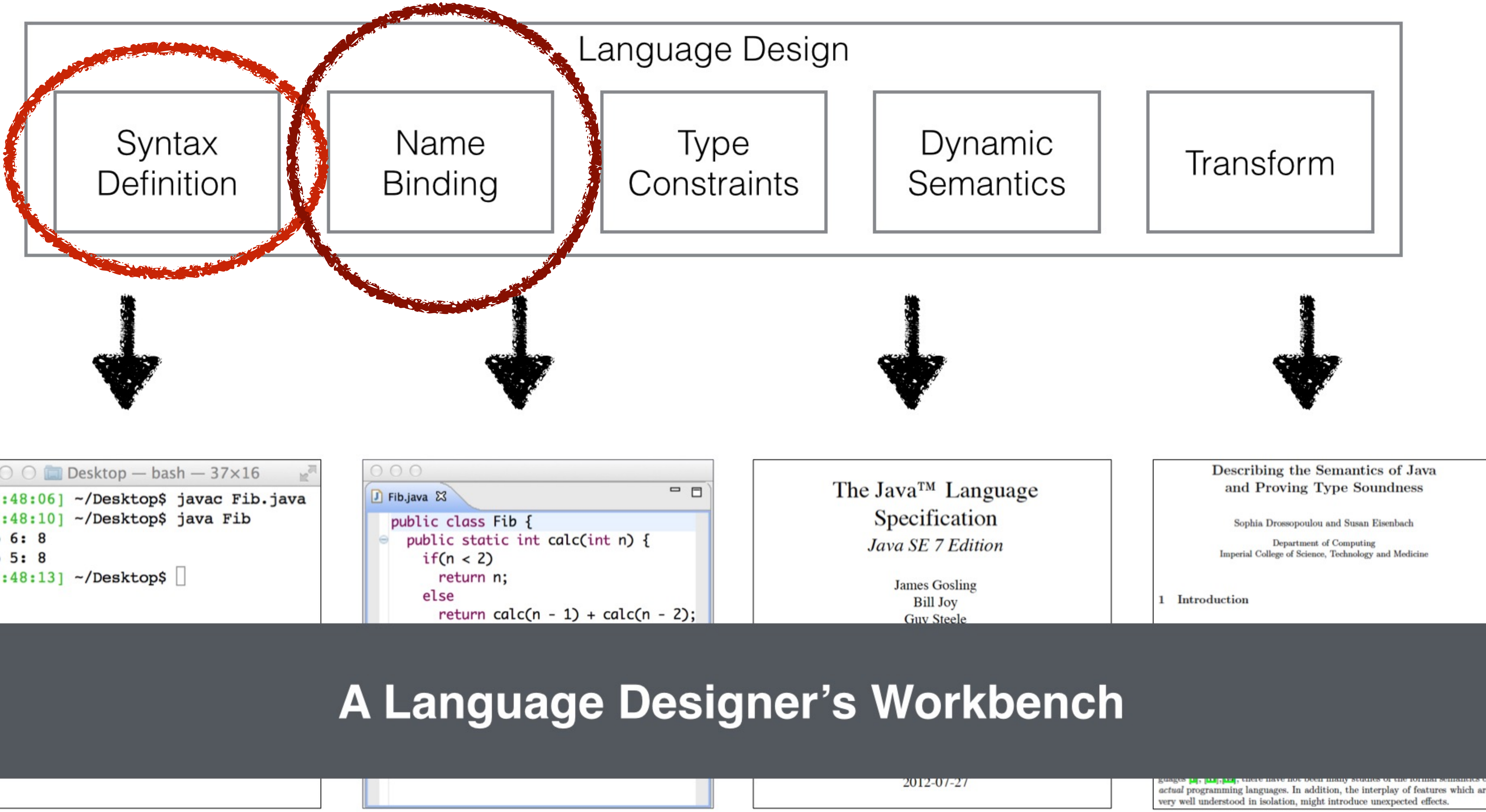
[Visser, Tolmach, Wachsmuth et al.; Onward 2014]

A Theory of Name Resolution



[Neron, Tolmach, Visser, Wachsmuth et al.; ESOP 2015]

Declarative Syntax Definition

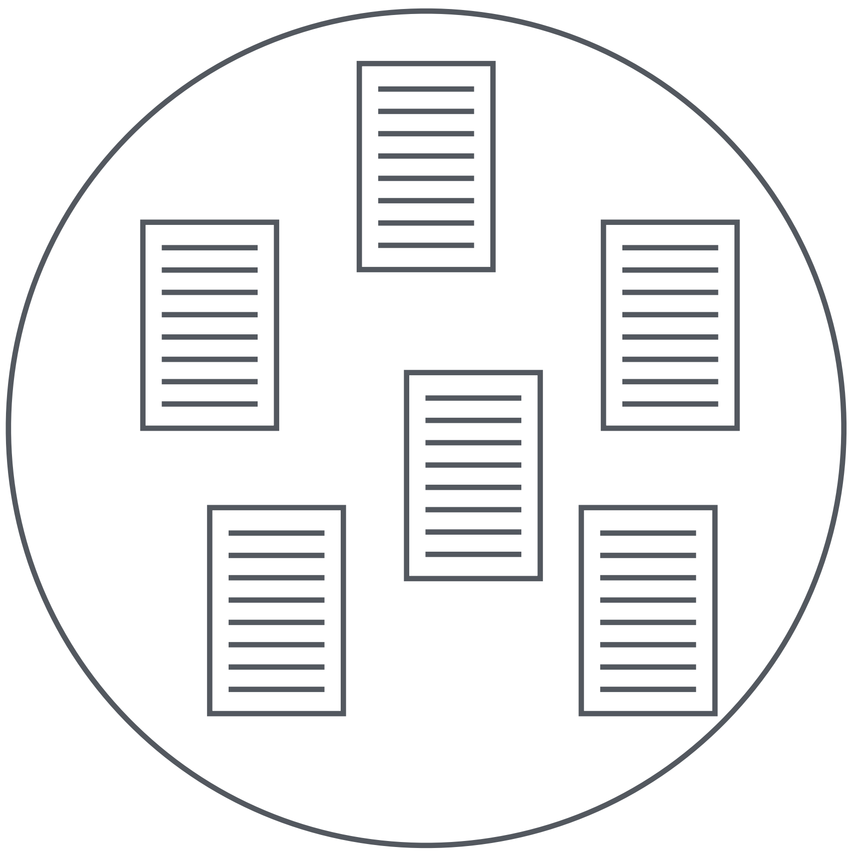


SDF2, SDF3 [Visser and many others 1994-2015]

Declare Your Syntax

[Kats, Visser, Wachsmuth; Onward 2010]

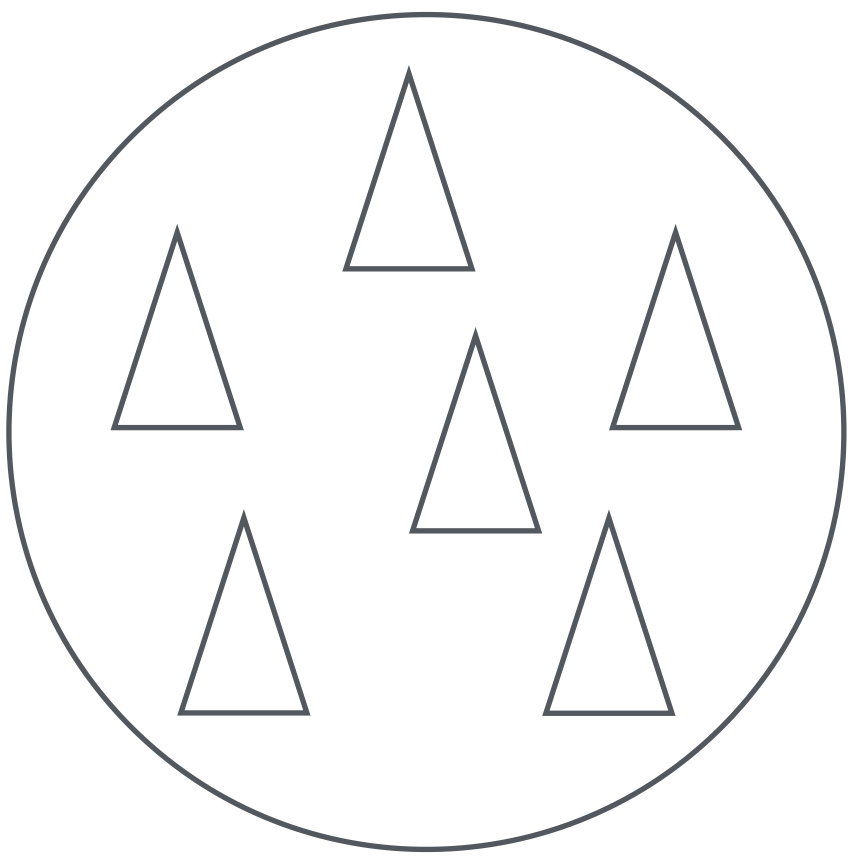
Language = Set of Sentences

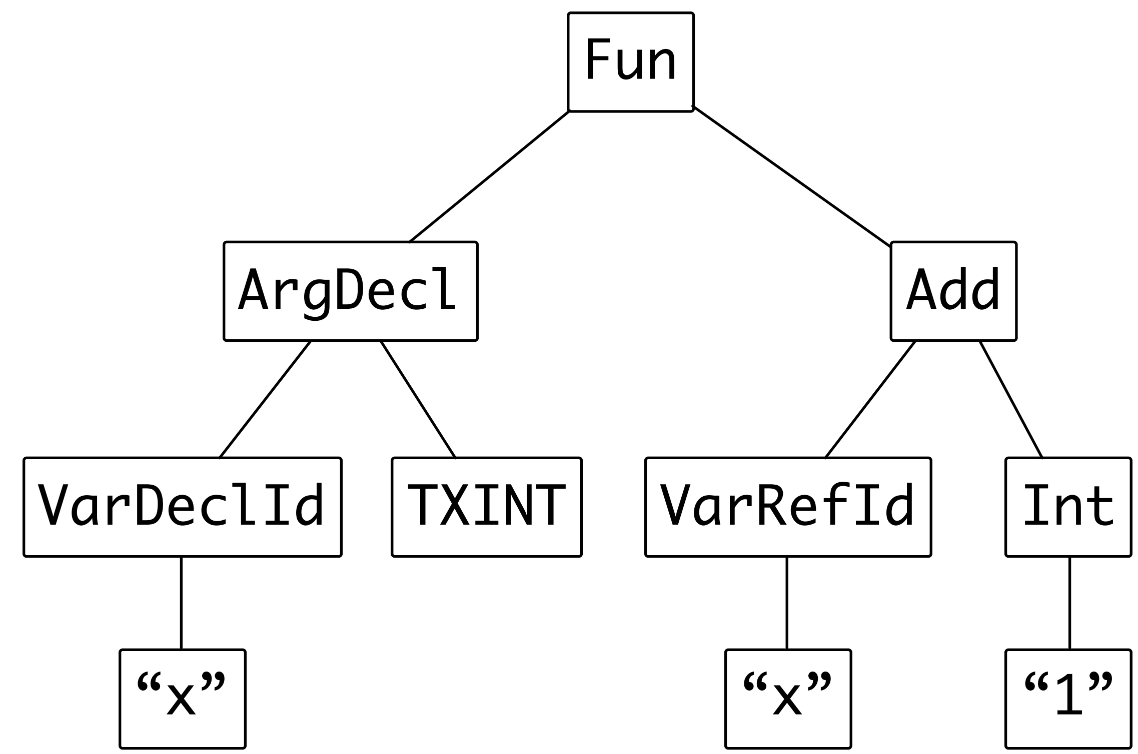


```
fun (x : Int) { x + 1 }
```

text is a convenient interface for writing and reading programs

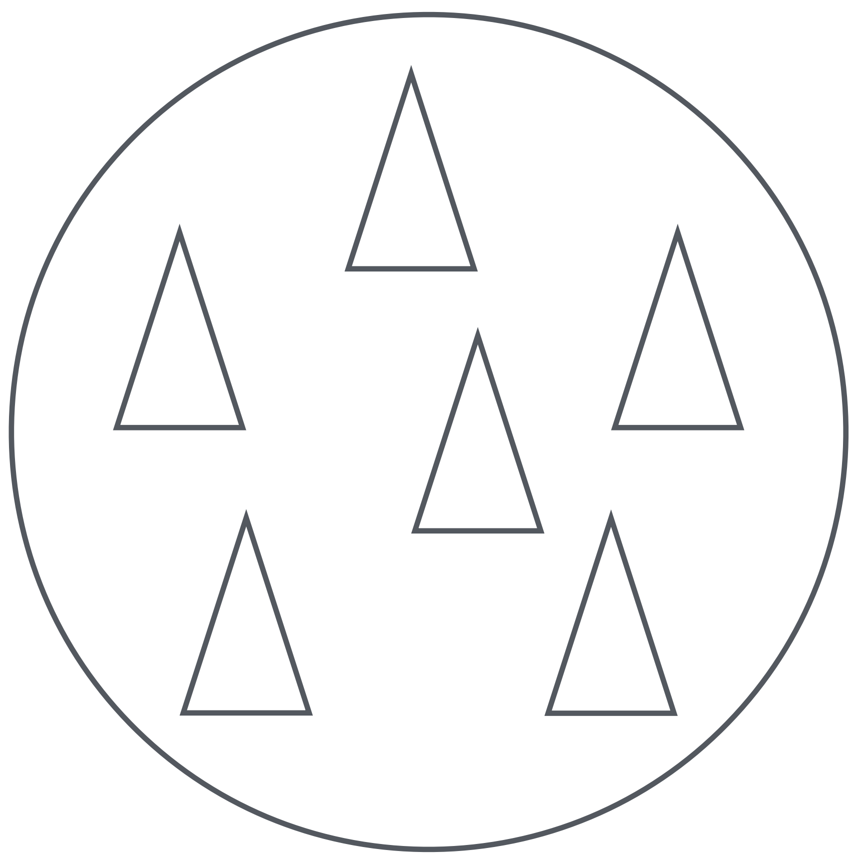
Language = Set of Trees





tree is a convenient interface for transforming programs

Tree Transformation

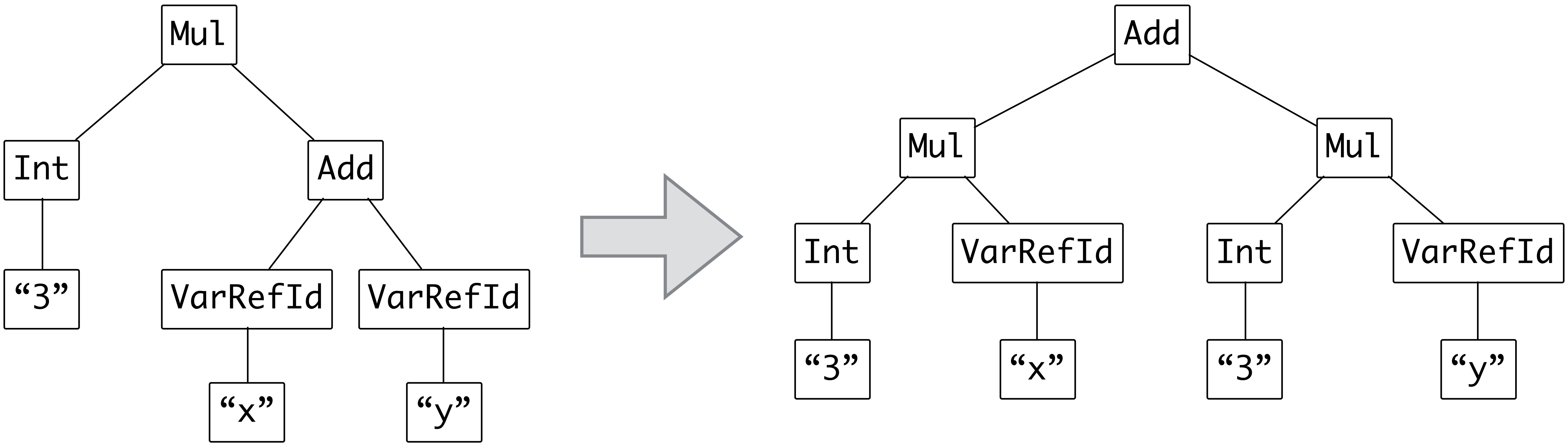


Syntactic
coloring
outline view
completion

Semantic
transform
translate
eval
analyze
refactor
type check

tree is a convenient interface for transforming programs

Tree Transformation

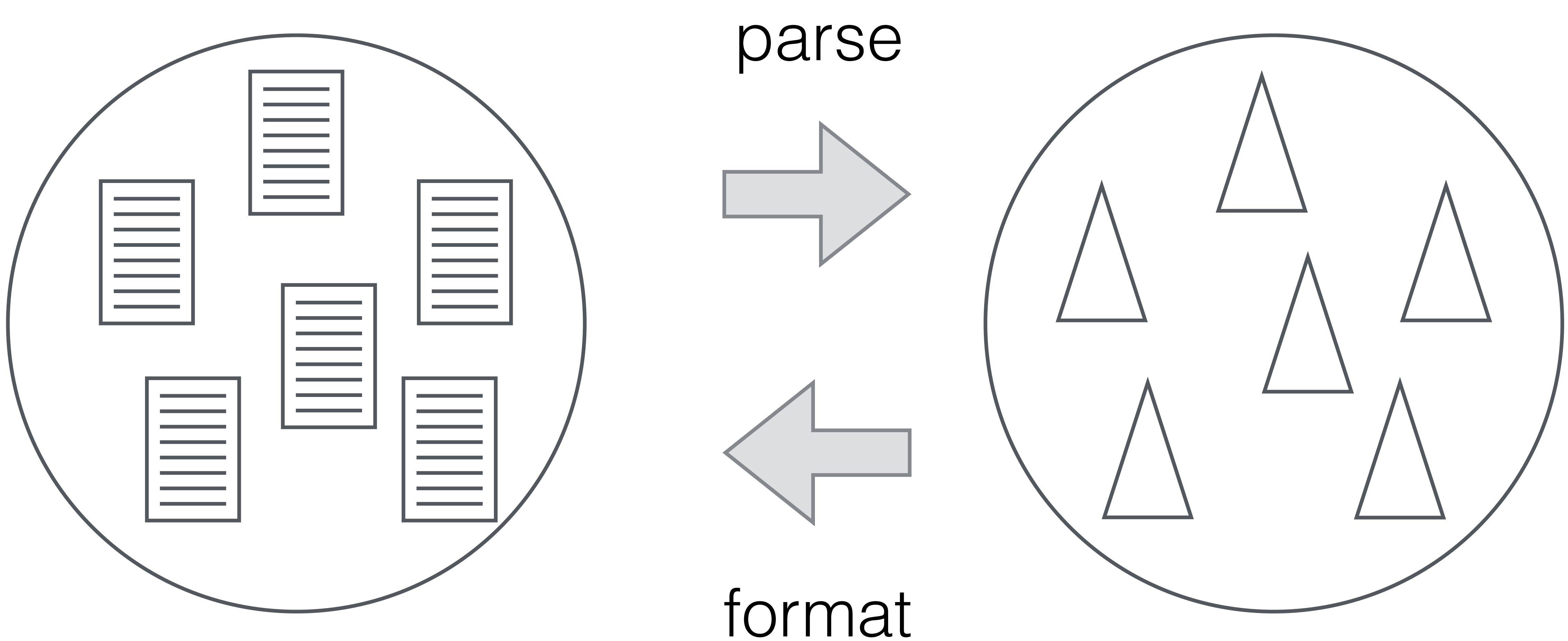



```
Mul(Int("3"),
    Add(VarRefId("x"),
        VarRefId("y")))
```

```
Add(Mul(Int("3"),
        VarRefId("x")),
    Mul(Int("3"),
        VarRefId("y")))
```

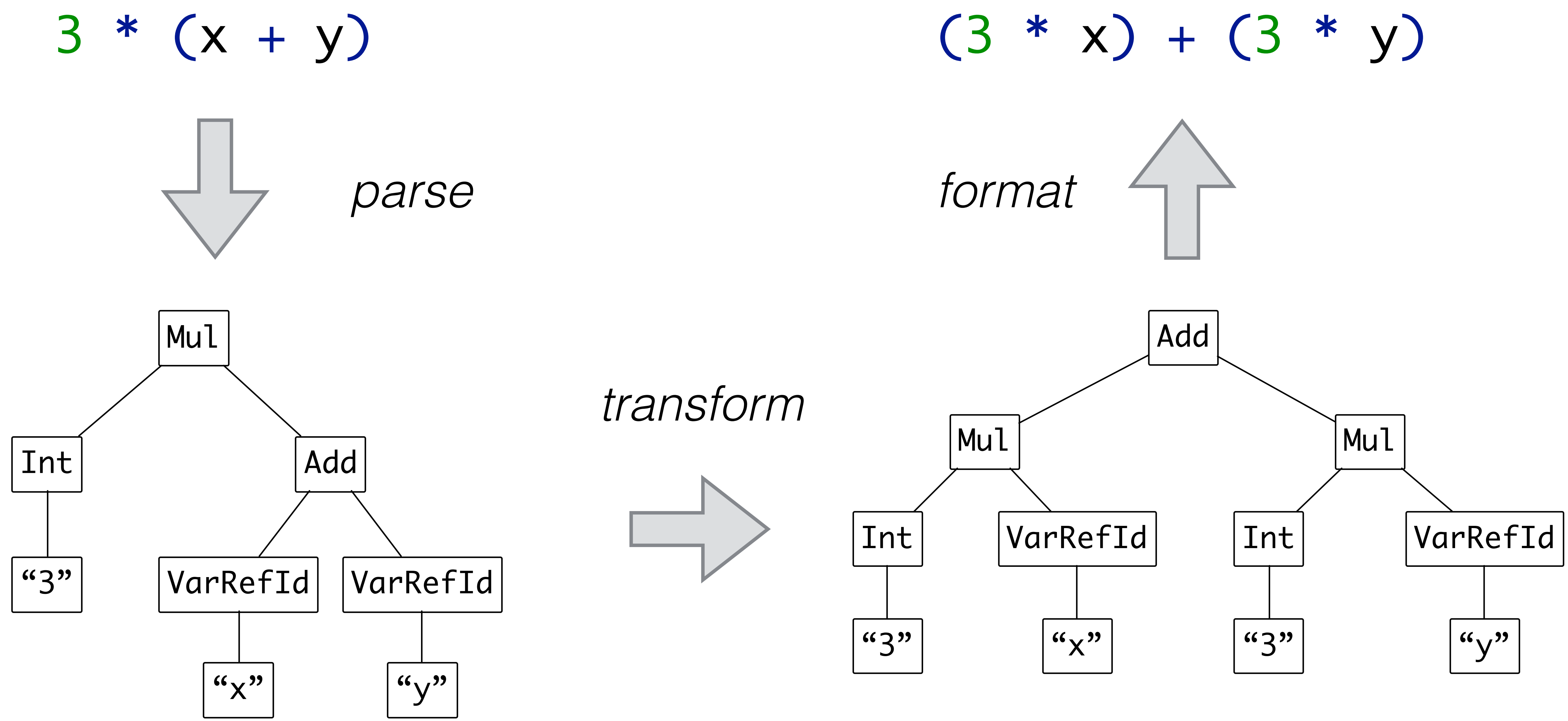
```
Mul(e1, Add(e2, e3)) -> Add(Mul(e1, e2), Mul(e1, e3))
```

Language = Sentences *and* Trees



different representations convenient for different purposes

From Text to Tree and Back



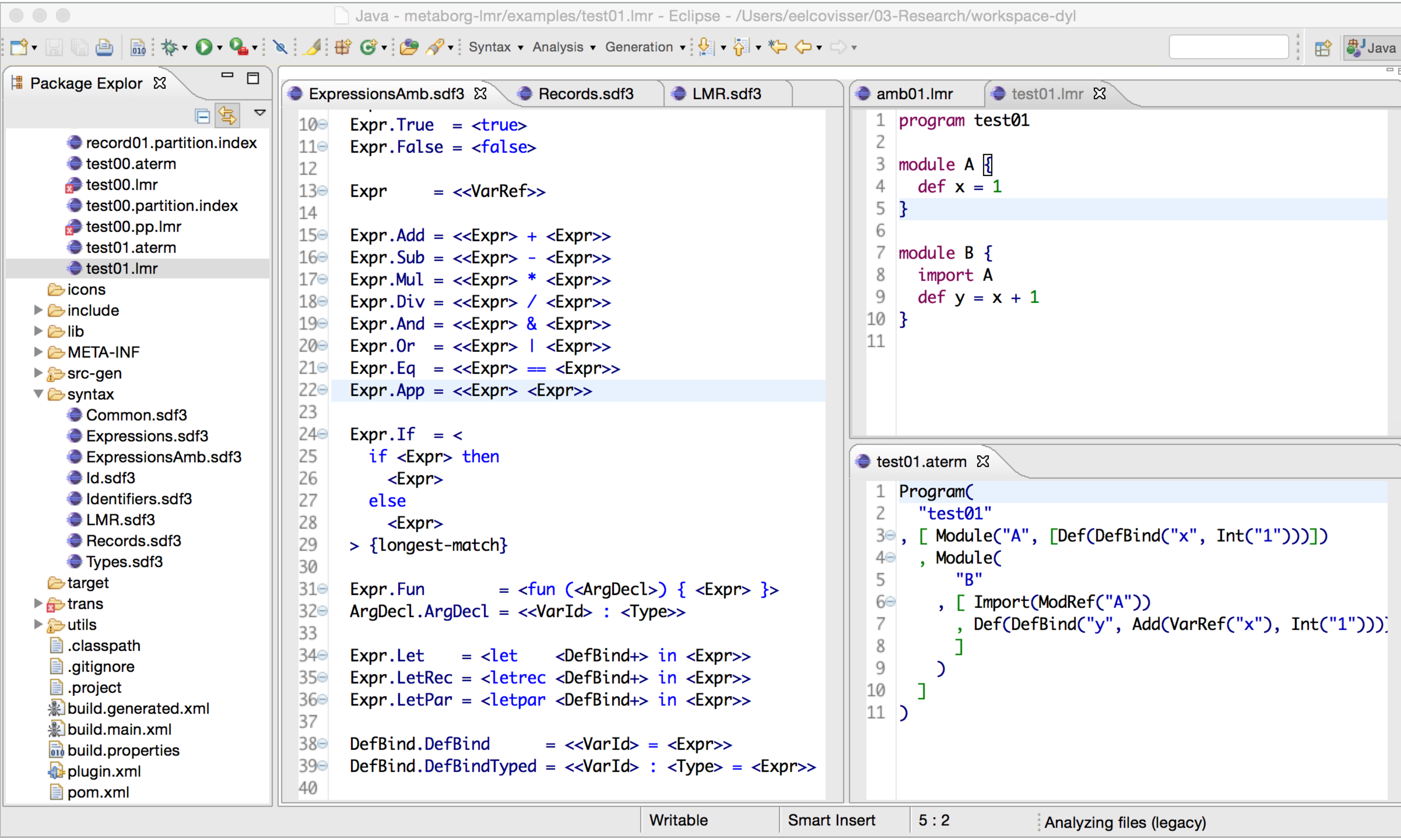
SDF3 defines Trees *and* Sentences

```
Expr.Int = INT
Expr.Add = <<Expr> + <Expr>>
Expr.Mul = <<Expr> * <Expr>>
```

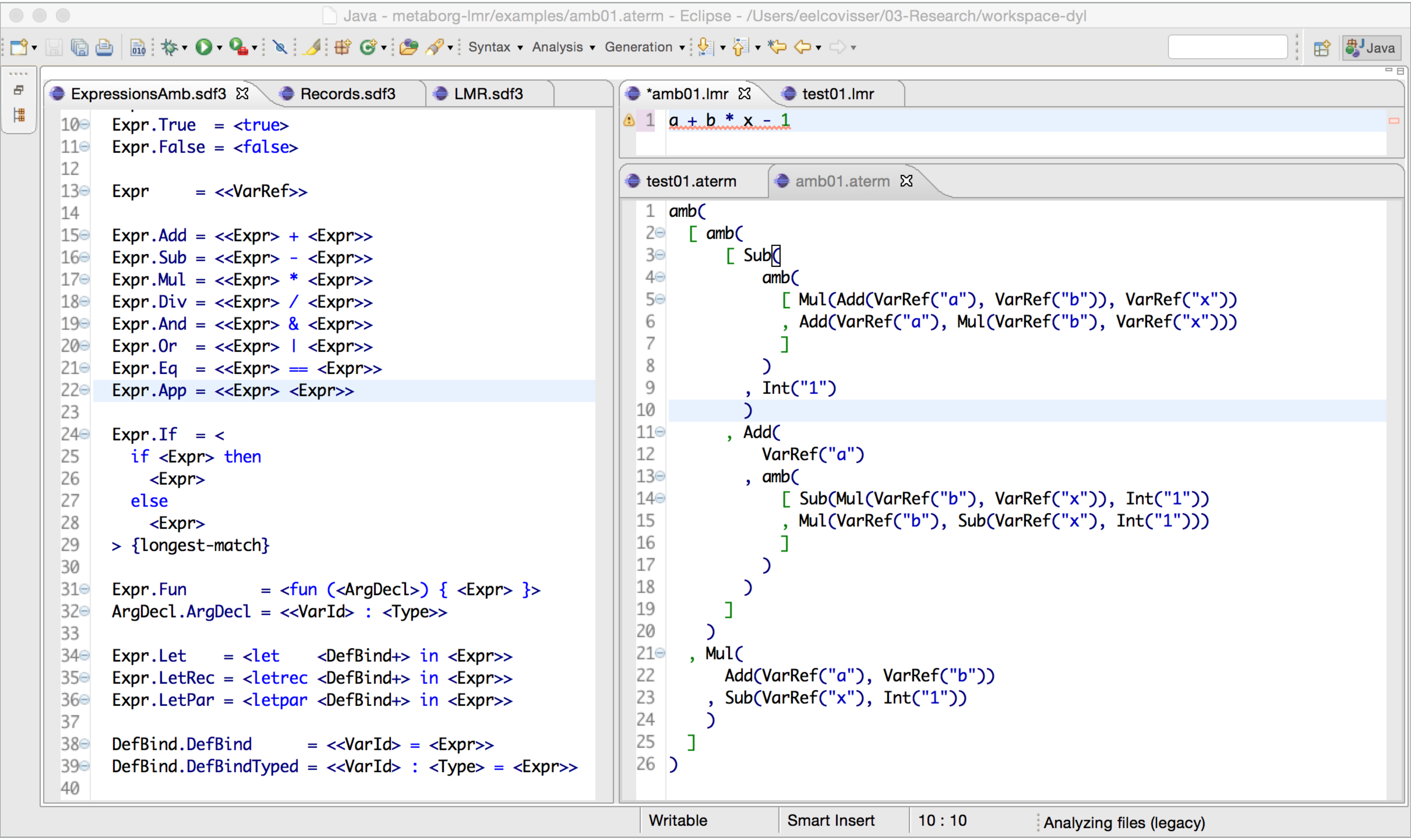
format (tree to text)	+	trees (structure)	=>	parse (text to tree)

```
parse(s) = t  where format(t) == s (modulo layout)
```

Grammar Engineering in Spoofax



Ambiguity



Java - metaborg-lmr/examples/amb01.aterm - Eclipse - /Users/eelcovisser/03-Research/workspace-dyl
Syntax Analysis Generation
Java
ExpressionsAmb.sdf3
Records.sdf3
LMR.sdf3
Expr.True = <true>
Expr.False = <false>

Expr = <<VarRef>>

Expr.Add = <<Expr> + <Expr>>
Expr.Sub = <<Expr> - <Expr>>
Expr.Mul = <<Expr> * <Expr>>
Expr.Div = <<Expr> / <Expr>>
Expr.And = <<Expr> & <Expr>>
Expr.Or = <<Expr> | <Expr>>
Expr.Eq = <<Expr> == <Expr>>
Expr.App = <<Expr> <Expr>>

Expr.If = <
if <Expr> then
<Expr>
else
<Expr>
> {longest-match}

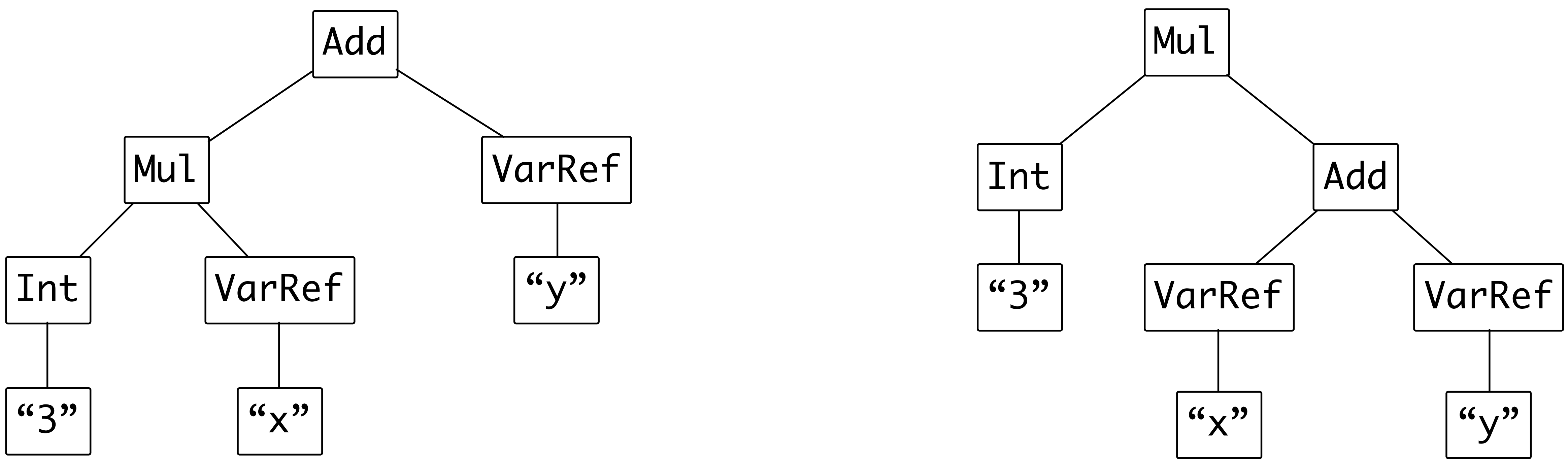
Expr.Fun = <fun (<ArgDecl>) { <Expr> }>
ArgDecl.ArgDecl = <<VarId> : <Type>>

Expr.Let = <let <DefBind+> in <Expr>>
Expr.LetRec = <letrec <DefBind+> in <Expr>>
Expr.LetPar = <letpar <DefBind+> in <Expr>>

DefBind.DefBind = <<VarId> = <Expr>>
DefBind.DefBindTyped = <<VarId> : <Type> = <Expr>>
*amb01.lmr
test01.lmr
a + b * x - 1
test01.aterm
amb01.aterm
amb(
[amb(
[Sub(
amb(
[Mul(Add(VarRef("a"), VarRef("b")), VarRef("x"))
, Add(VarRef("a"), Mul(VarRef("b"), VarRef("x")))
]
)
, Int("1")
)
, Add(
VarRef("a")
, amb(
[Sub(Mul(VarRef("b"), VarRef("x")), Int("1"))
, Mul(VarRef("b"), Sub(VarRef("x"), Int("1")))
]
)
)
]
)
, Mul(
Add(VarRef("a"), VarRef("b"))
, Sub(VarRef("x"), Int("1"))
)
]
)
Writable
Smart Insert
10 : 10
Analyzing files (legacy)

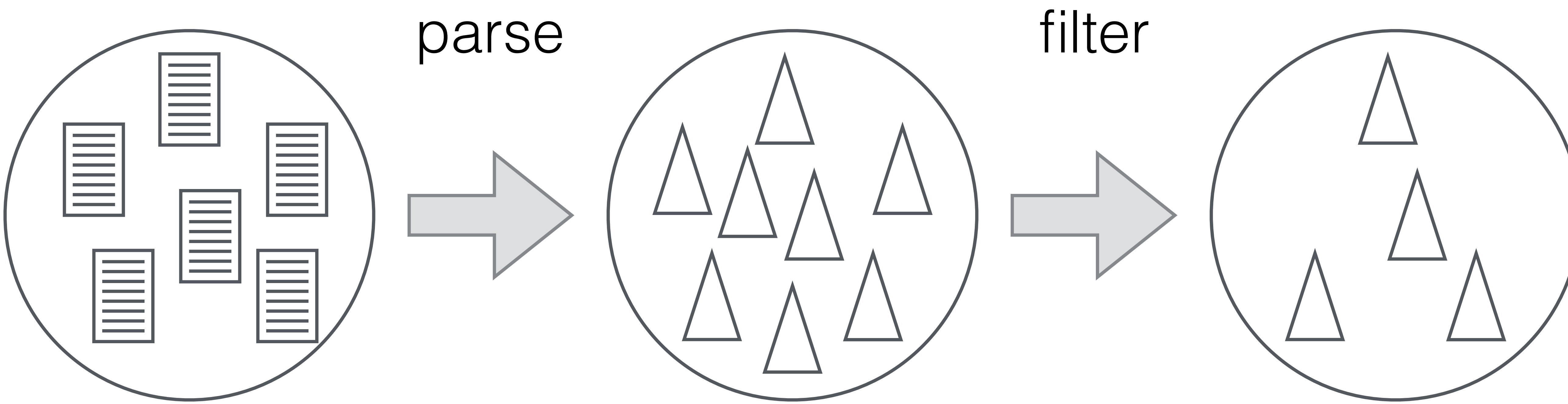
Ambiguity

```
3 * x + y
```



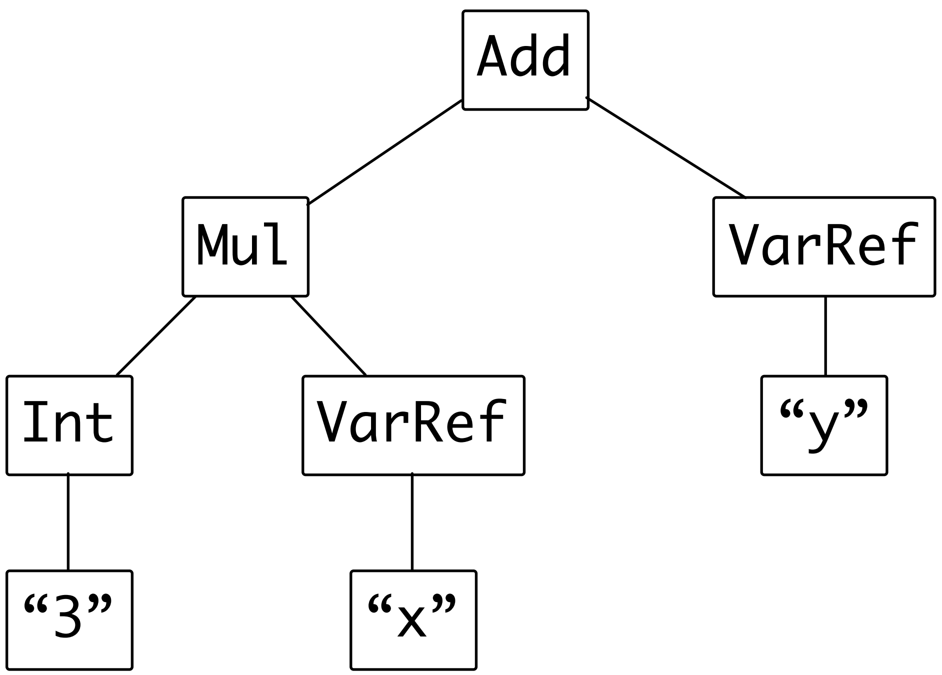

```
t1 != t2  /\ format(t1) = format(t2)
```

Declarative Disambiguation

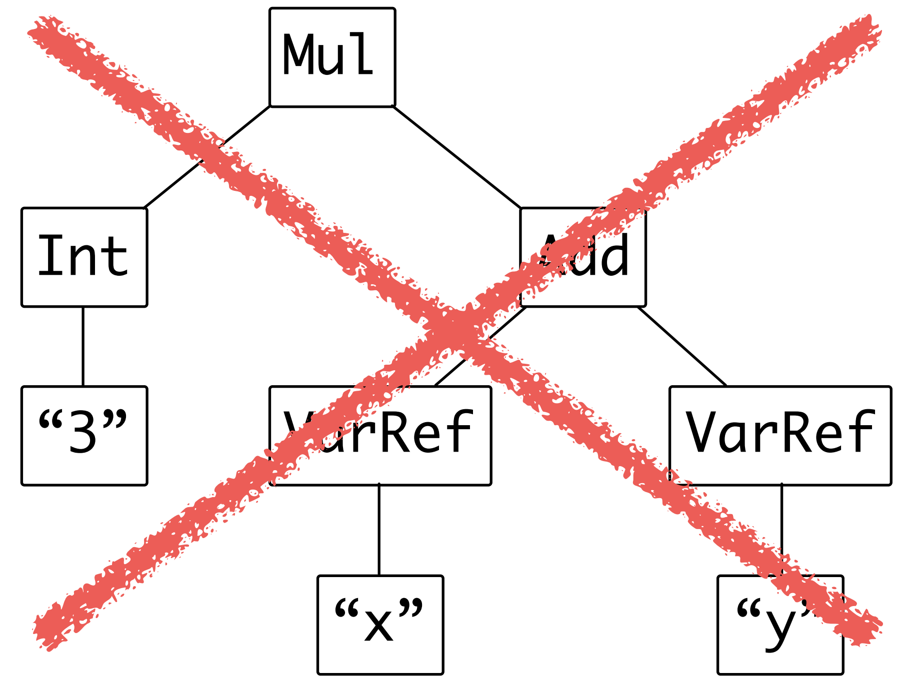


Disambiguation Filters [Klint & Visser; 1994], [Van den Brand, Scheerder, Vinju, Visser; CC 2002]

Priority and Associativity




```
3 * x + y
```




```
context-free syntax
  Expr.Int = INT
  Expr.Add = <<Expr> + <Expr>> {left}
  Expr.Mul = <<Expr> * <Expr>> {left}
context-free priorities
  Expr.Mul > Expr.Add
```

Recent improvement: safe disambiguation of operator precedence [Afroozeh et al. SLE13, Onward15]

Multi-Purpose Declarative Syntax Definition

```
Exp.Ifz = <
 ifz <Exp> then
  <Exp>
 else
  <Exp>
>
```

Syntax Definition

Parser

Error recovery rules

Pretty-Printer

Abstract syntax tree

Syntactic coloring

Syntactic completion

Folding rules

Outline rules

Declare Your Syntax : Summary

(1) language-specific grammar + disambiguation rules

(2) language-independent spec of well-formed trees for grammar

(3) formatting based on layout hints in grammar

(4) parser generated automatically

(4’) no need to understand parsing algorithm

(4’’) debugging in terms of representation

(5) syntactic and semantic operations abstract from parsing

Declare Your Names

A Theory of Name Resolution




Pierre Neron[1]

Andrew Tolmach[2]

Eelco Visser[1]

Guido Wachsmuth[1]

1 TU Delft Delft University of Technology

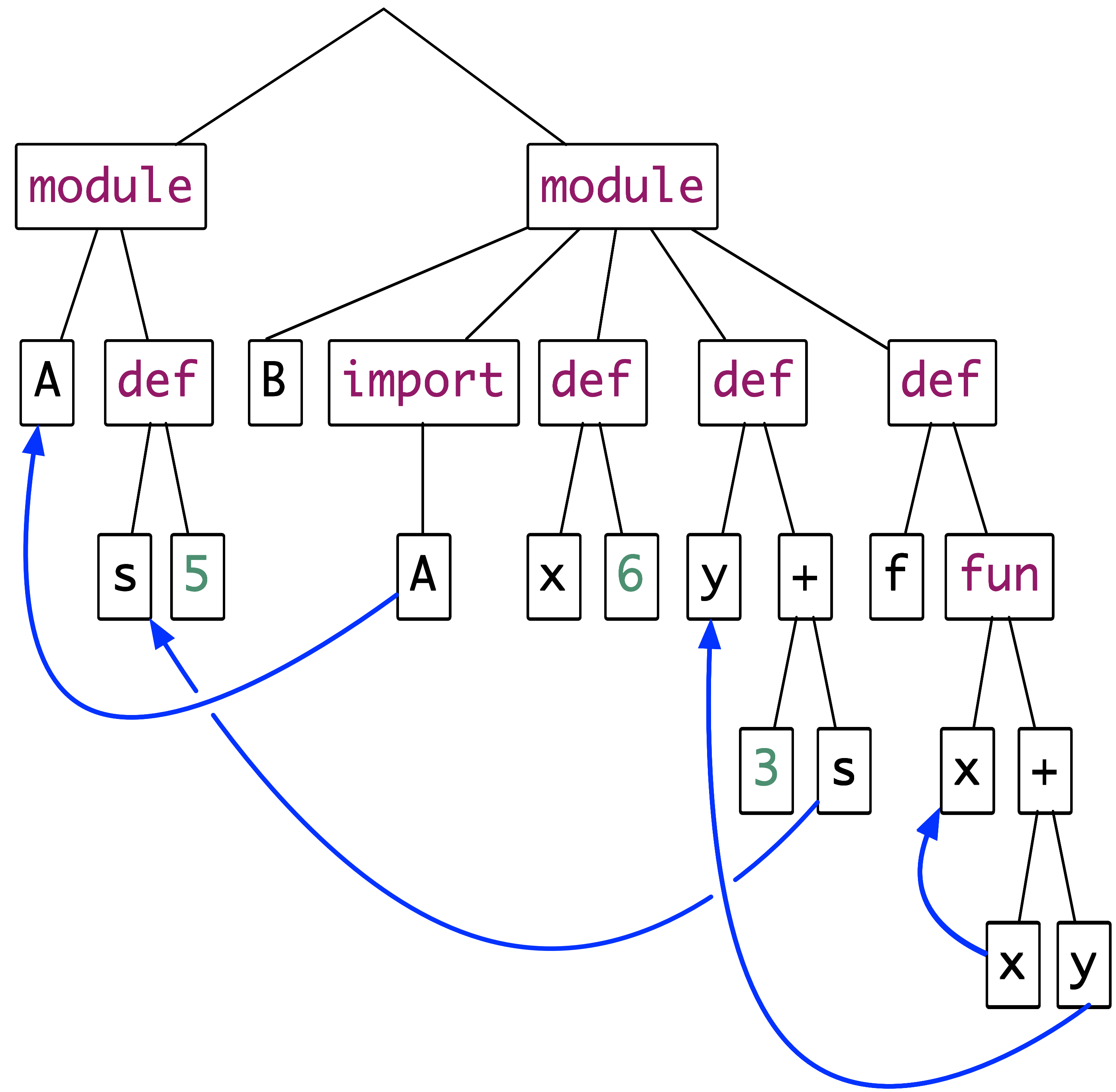
2 Portland State UNIVERSITY

[ESOP15]

Language = Set of Graphs

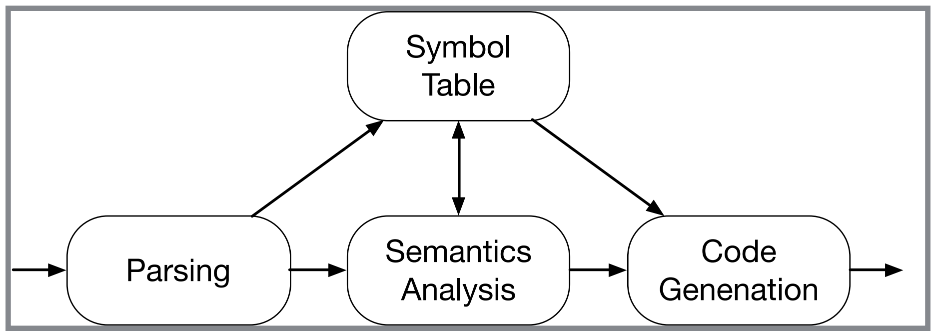
```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```



Name Resolution is Pervasive

Appears in many different artifacts…

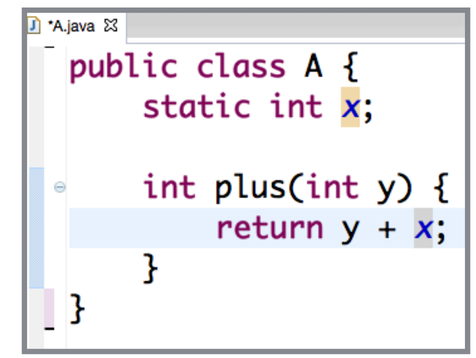


Compiler

$$\frac{x:\tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

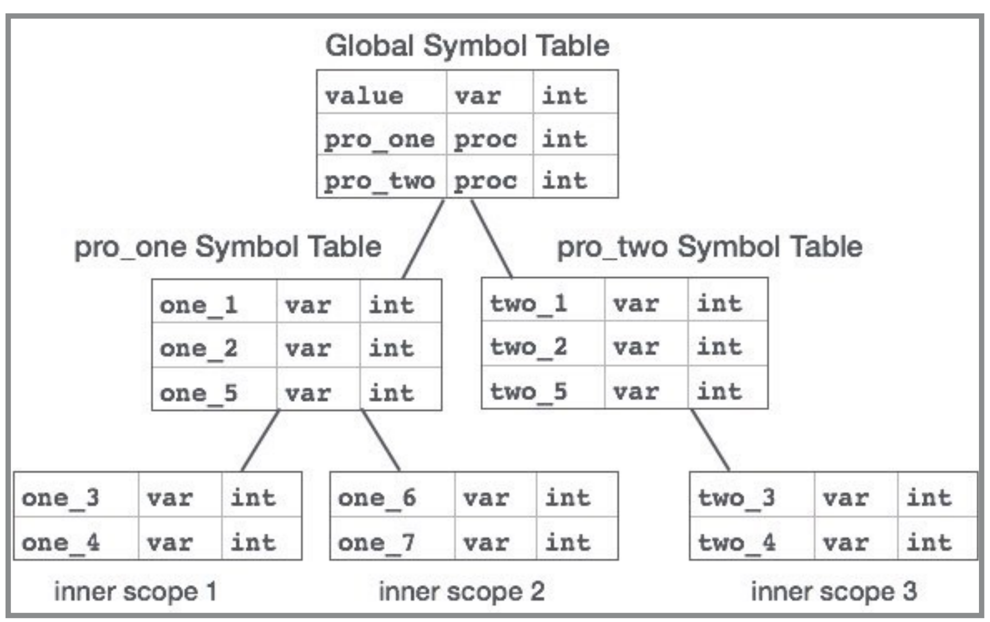
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics



IDE

… with rules encoded in many different ad-hoc ways



x:int, Γ

lookup(x_i)

[3/x].σ

No standard approach, no re-use

Contrast with Syntax

A unique definition

```
program  =  decl*
   decl  =  module id { decl* }
         |  import qid
         |  def id = exp
    exp  =  qid
         |  fun id { exp }
         |  fix id { exp }
         |  let bind* in exp
         |  letrec bind* in exp
         |  letpar bind* in exp
         |  exp exp
         |  exp ⊕ exp
         |  int
    qid  =  id
         |  id . qid
   bind  =  id = exp
```

A standard formalism

Context-Free Grammars

Supports

- Parser
- AST
- Pretty-Printing
- Highlighting

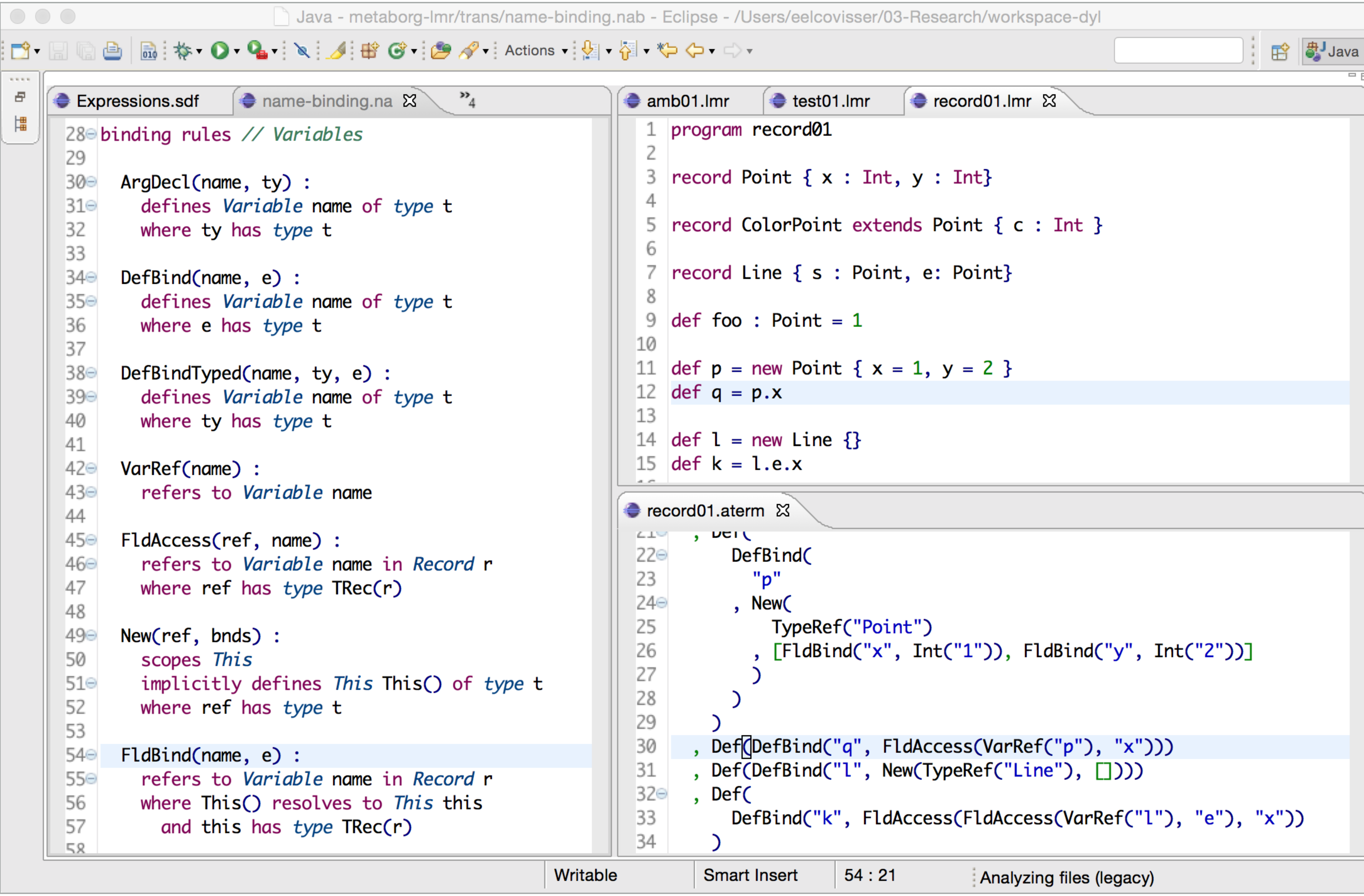
Representing Bound Programs

- Many approaches to representing the results of name resolution within an (extended) AST, e.g.
  - numeric indexing [deBruijn72]
  - higher-order abstract syntax [PfenningElliott88]
  - nominal logic approaches [GabbayPitts02]
- Good support for binding-sensitive AST manipulation
- But: Do not say how to resolve identifiers in the first place!
  - Also: Can’t represent ill-bound programs
  - And: Tend to be biased towards lambda-like bindings

Binding Specification Languages

- Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  - Ott [Sewell+10]
  - Romeo [StansiferWand14]
  - Unbound [Weirich+11]
  - Cαml [Pottier06]
  - NaBL [Konat+12]
- Generate code to do resolution and record results

The NaBL Name Binding Language



Multi-Purpose Name Binding Rules

```
module names

namespaces Variable

binding rules

  Var(x) :
    refers to Variable x

  Param(x, t) :
    defines Variable x of type t

  Fun(p, e) :
    scopes Variable

  Fix(p, e) :
    scopes Variable

  Let(x, t, e1, e2) :
    defines Variable x of type t in e2
```

Incremental name resolution algorithm

Name checks

Reference resolution

Semantic code completion

Refactorings

Binding Specification Languages

- Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  - Ott [Sewell+10]
  - Romeo [StansiferWand14]
  - Unbound [Weirich+11]
  - Cαml [Pottier06]
  - NaBL [Konat+12]
- Generate code to do resolution and record results
- But: what are the **semantics** of such a language?

The Missing Piece

- Answer: the meaning of a binding specification for language L should be given by a function from L programs to their **“resolution structures”**
- So we need a (uniform, language-independent) method for describing such resolution structures...
- ...that can be used to compute the resolution of each program identifier
  - (or to verify that a claimed resolution is valid)

Design Goals

- Handle broad range of language binding features...
- ...using minimal number of constructs
- Make resolution structure language-independent
- Handle named collections of names (e.g. modules, classes, etc.) within the theory
- Allow description of programs with resolution errors

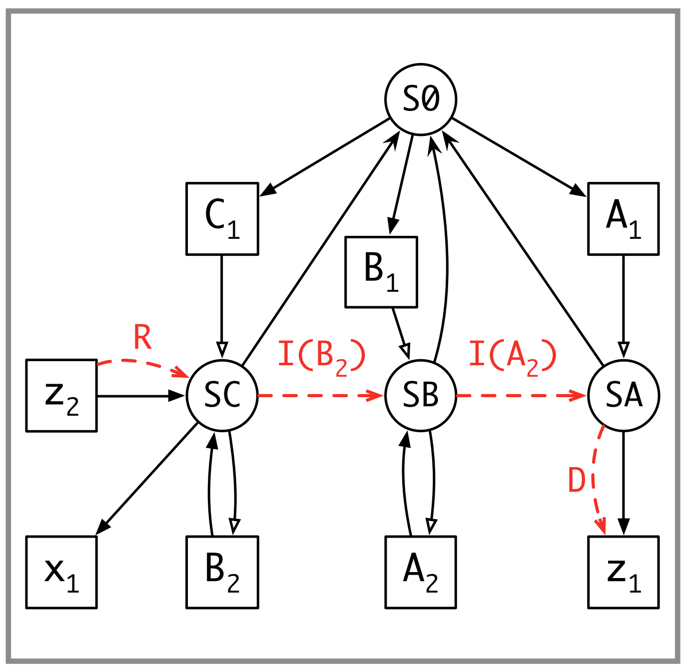
A Theory of Name Resolution

For ***statically lexically scoped*** *languages*

A unique representation

A standard formalism

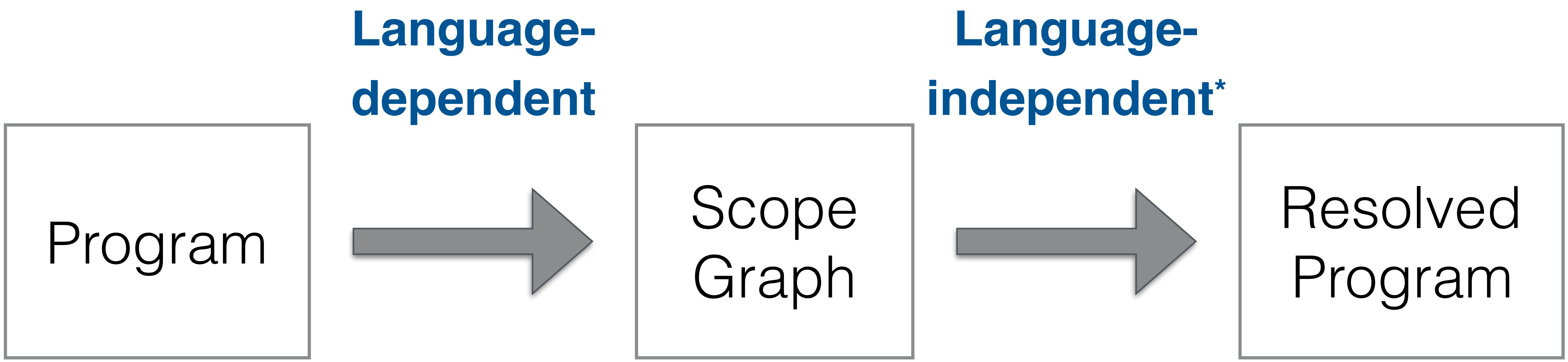
Supports



Scope Graphs

- Resolution
- α-equivalence
- IDE Navigation
- Refactoring tools
- Reasoning tools

Resolution Scheme



Resolution of a reference in a scope graph:

Building a **path**
from a **reference** node
to a **declaration** node
following path construction **rules**

*Parameterized by notions of path **well-formedness** and **ordering**

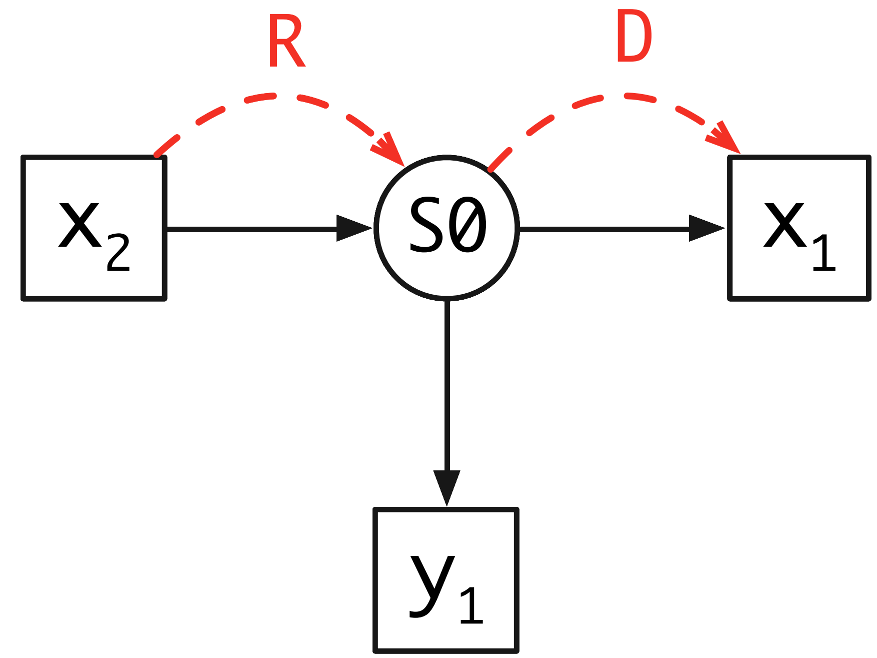
Scope Graphs by Example

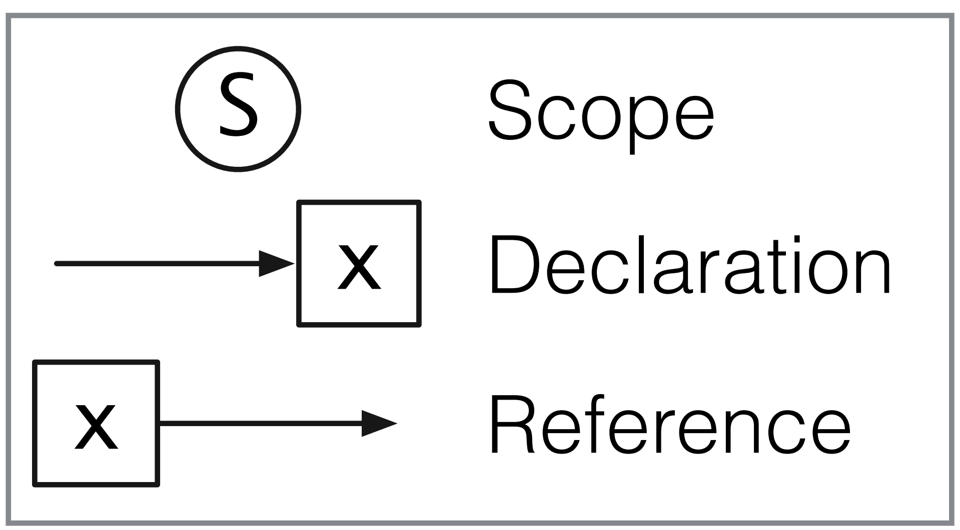
Simple Scopes

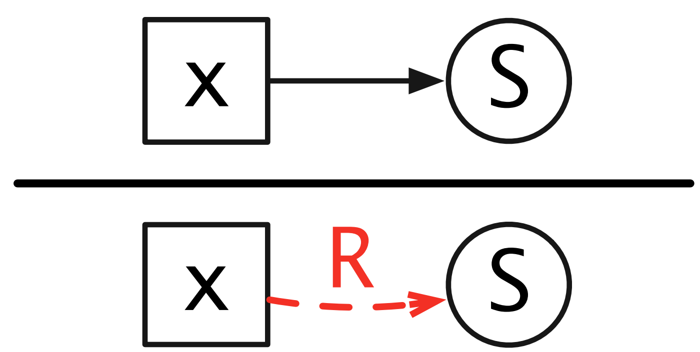
S0

$\texttt{def}\ y_1 = x_2 + 1$

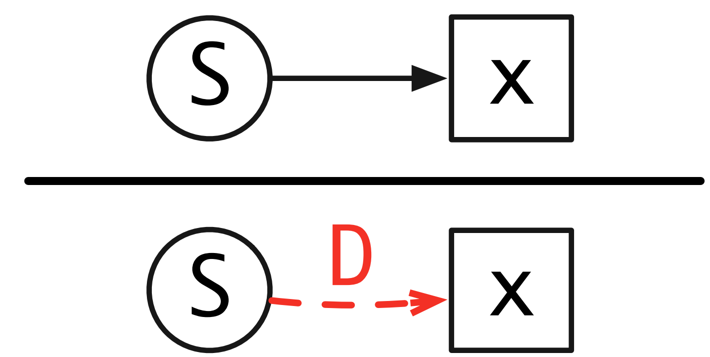
$\texttt{def}\ x_1 = 5$







Reference Step



Declaration Step

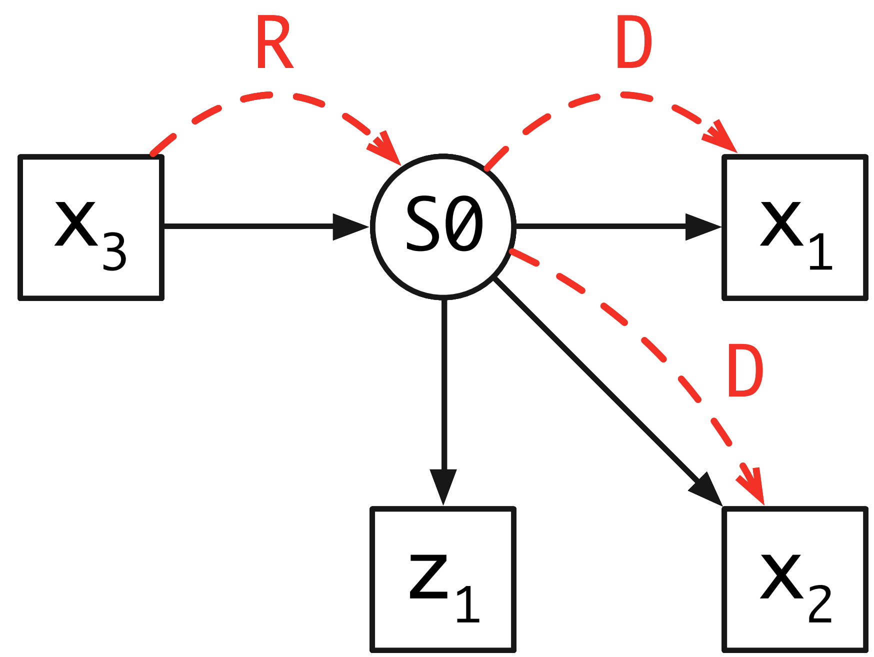
Ambiguous Resolutions

S0

def $\boxed{x_1}$ = 5

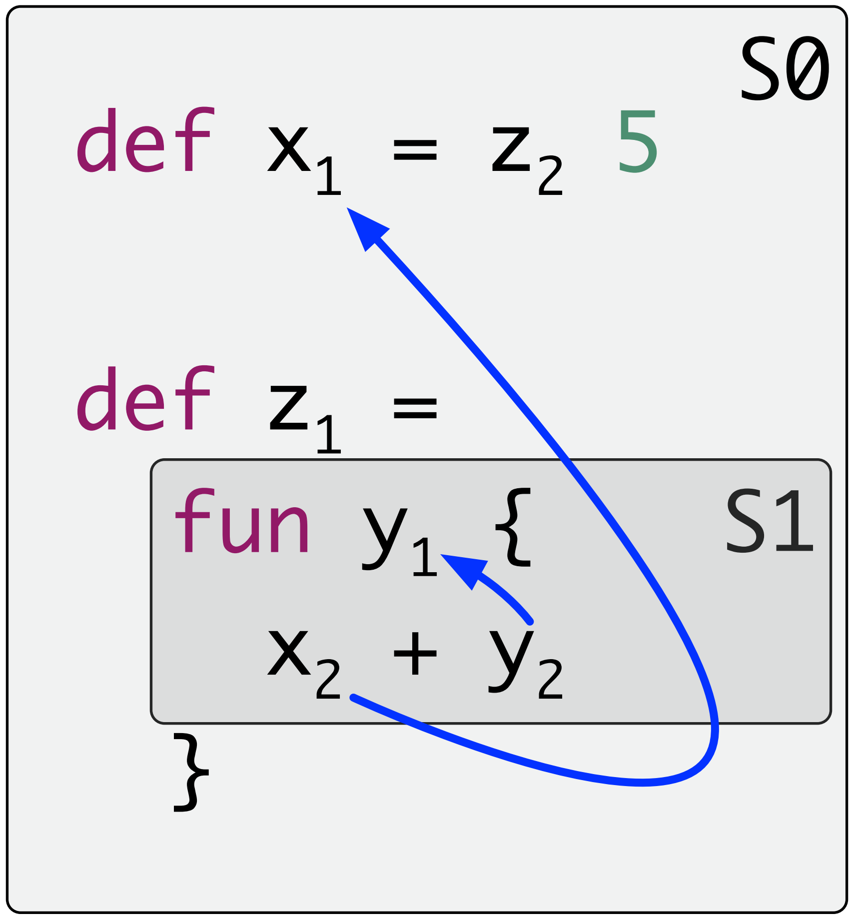
def $\boxed{x_2}$ = 3

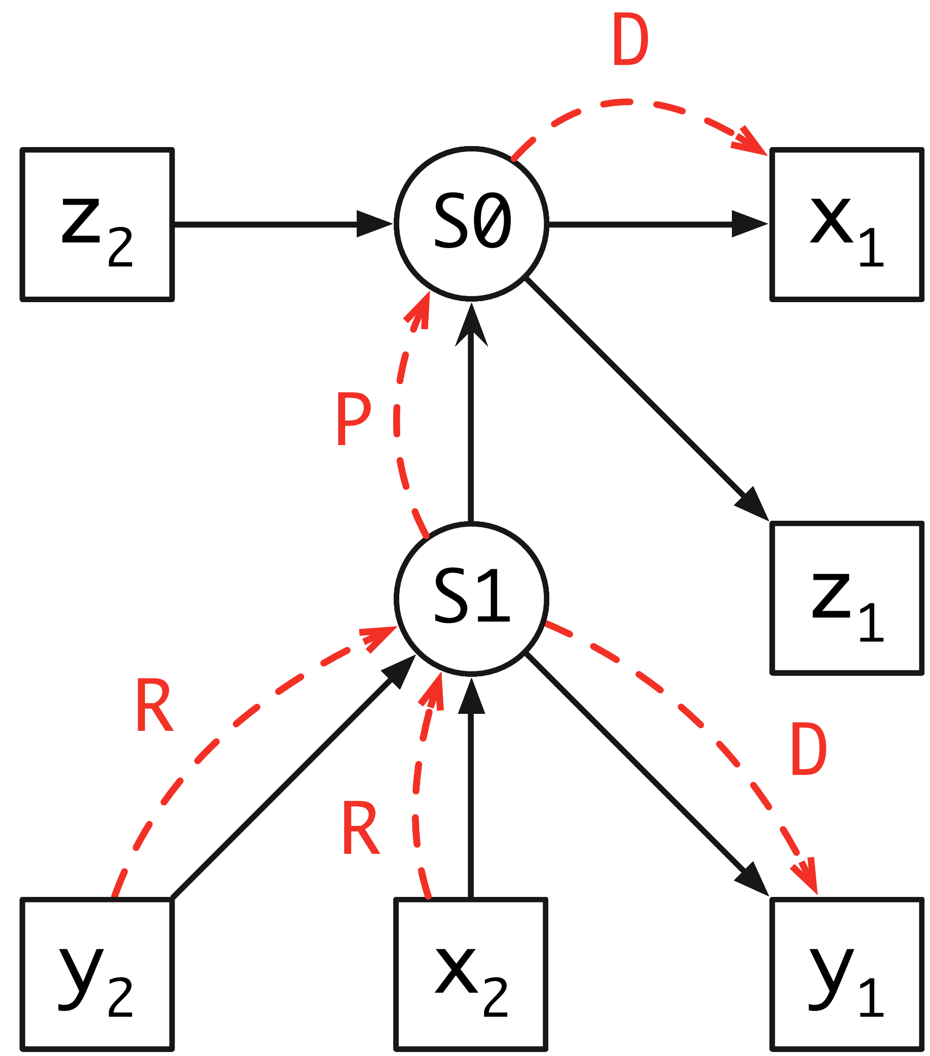
def $z_1 = x_3 + 1$

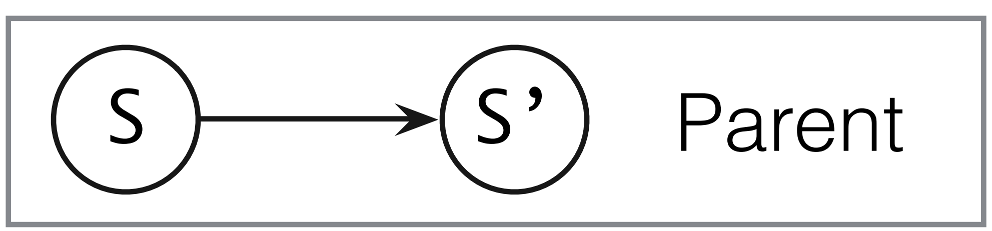



```
match t with
 | A [x] | B [x] => …
```

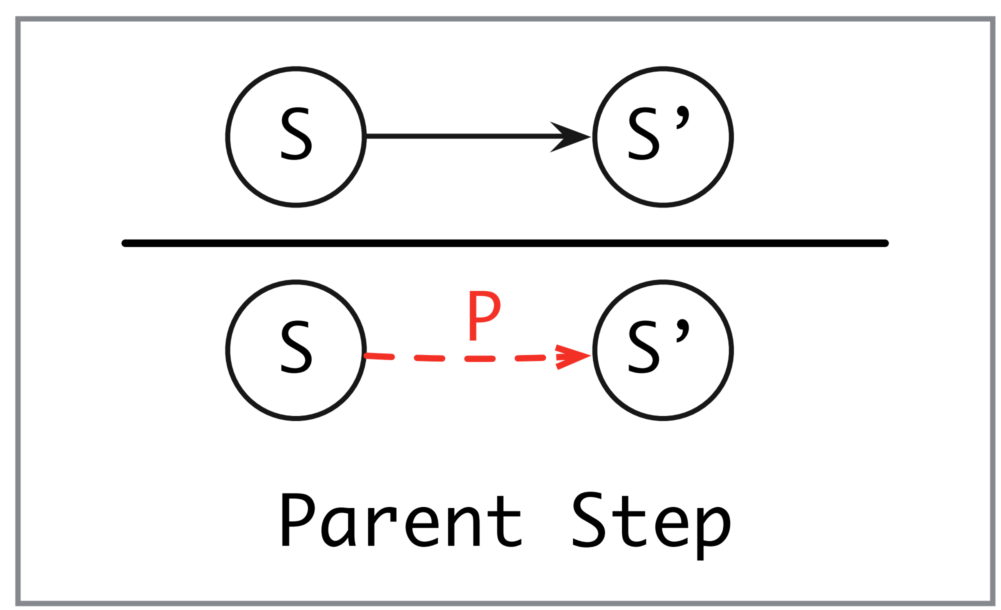
Lexical Scoping



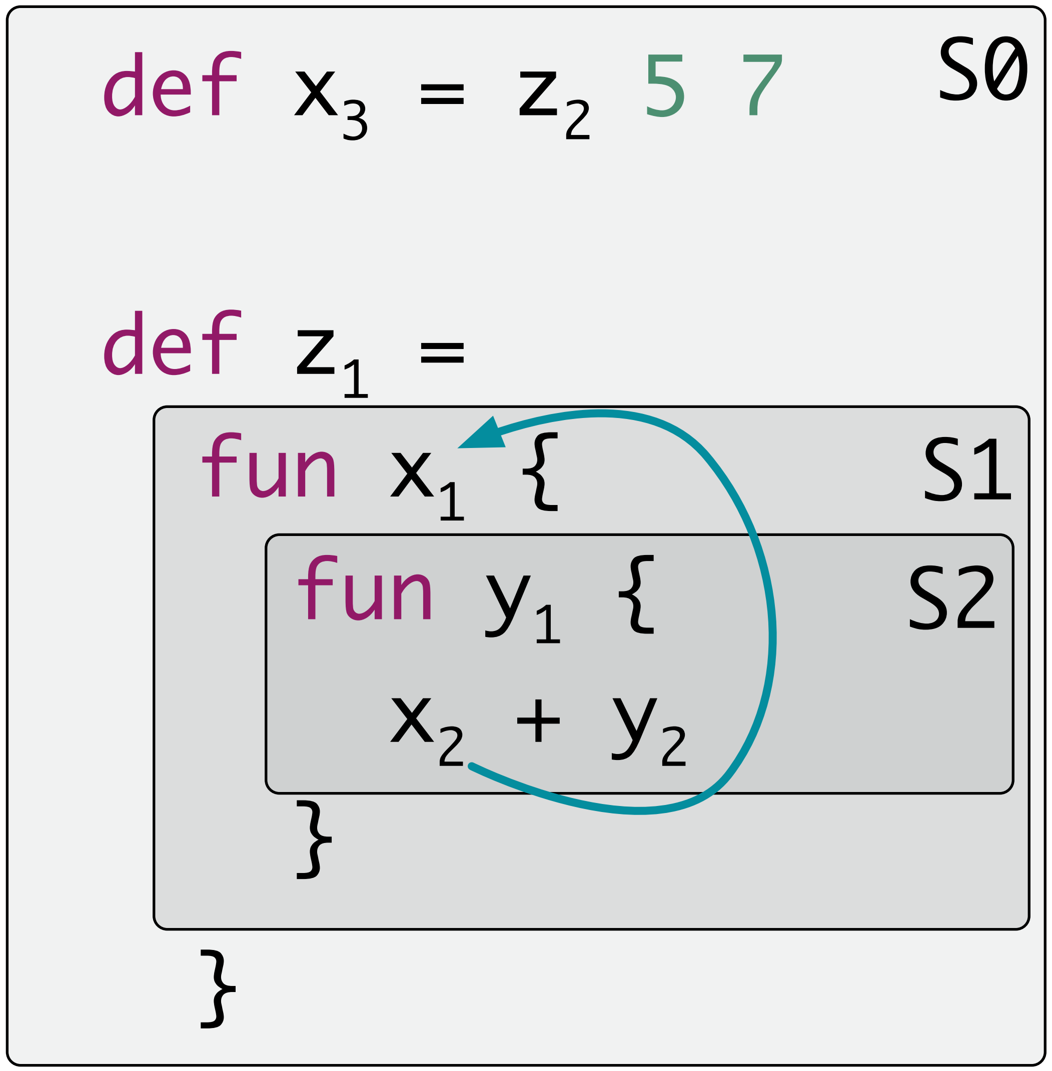




Well formed path: R.P*.D

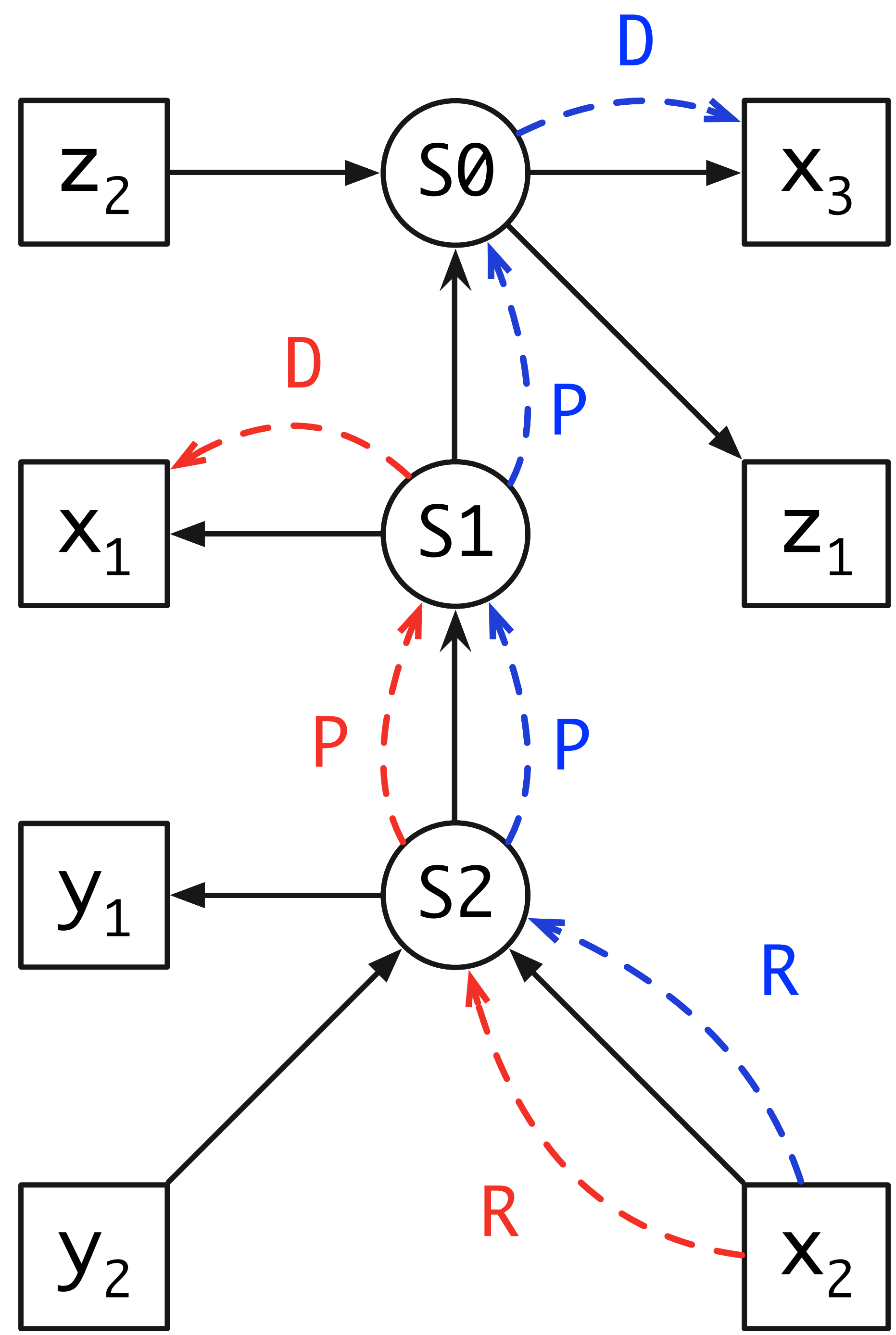


Shadowing



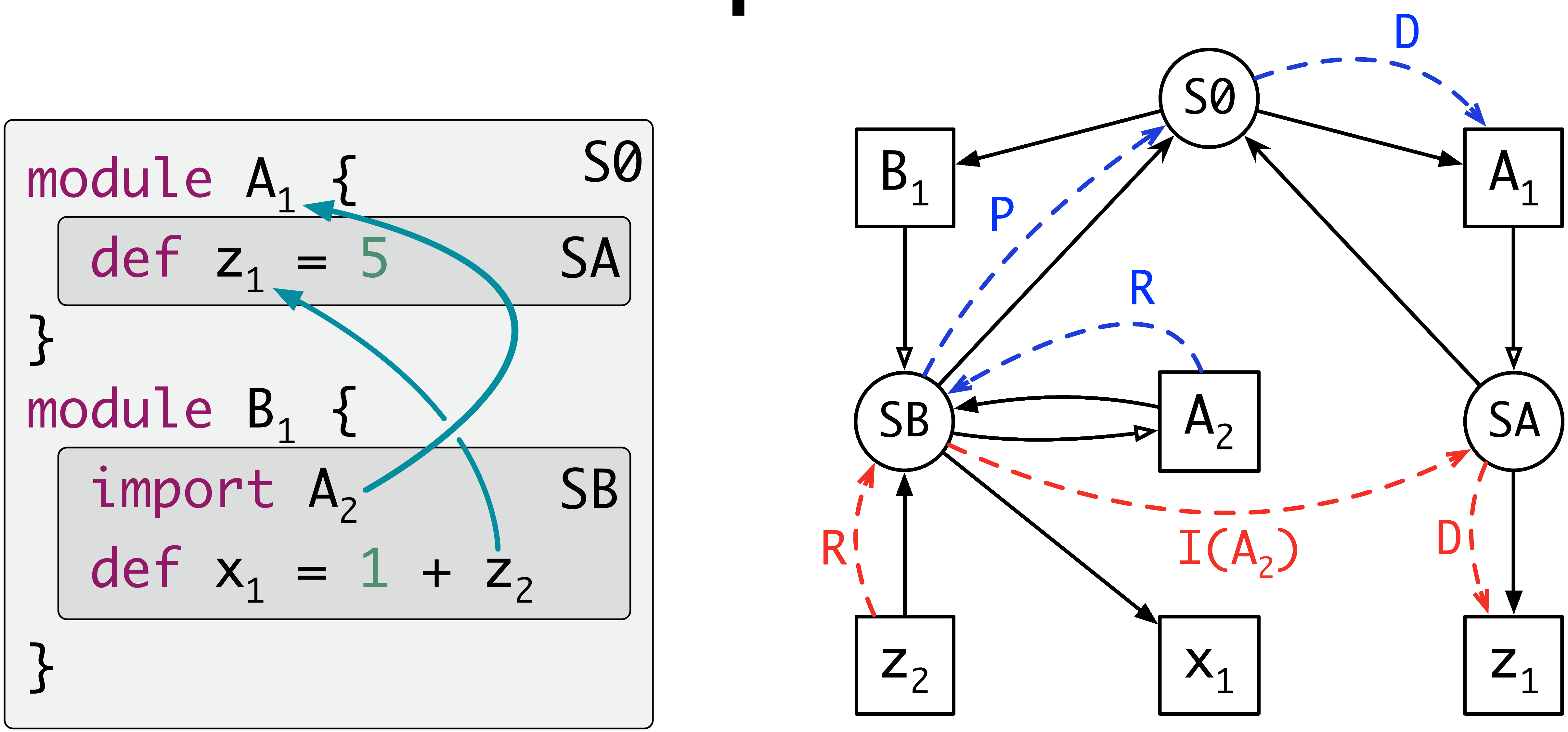
$$\frac{}{D < P.p}$$

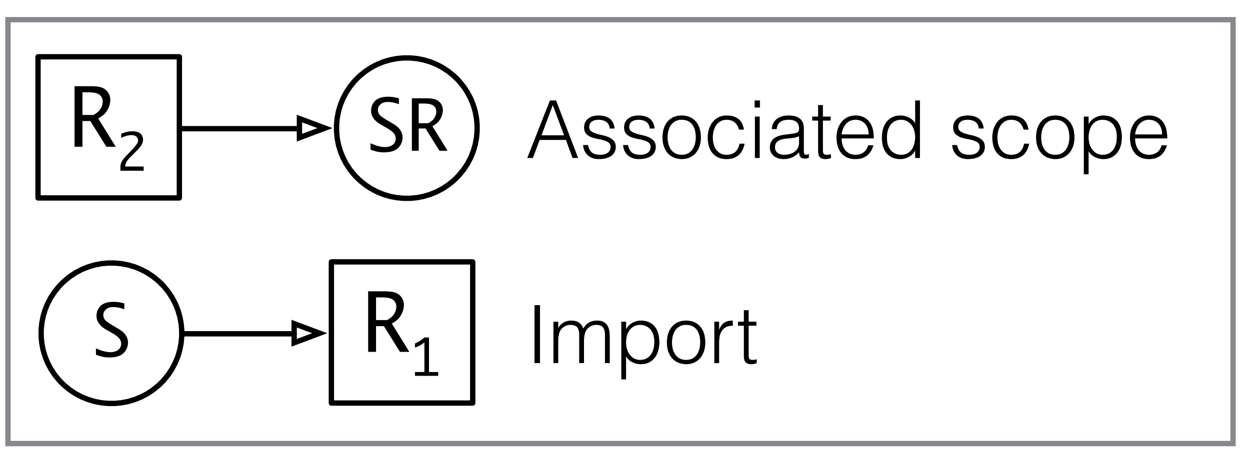
$$\frac{p < p'}{s.p < s.p'}$$

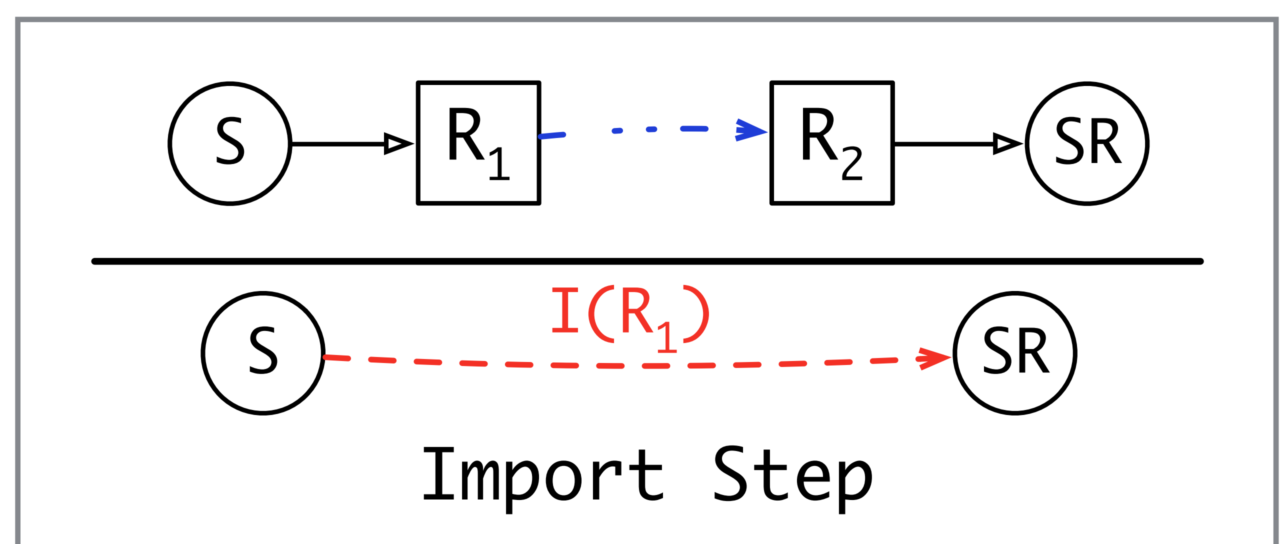


$$R.P.D < R.P.P.D$$

Imports


module A1 {
def z1 = 5
}
module B1 {
import A2
def x1 = 1 + z2
}
S0
SA
SB
D
P
R
I(A2)
B1
A1
A2
z2
x1
z1

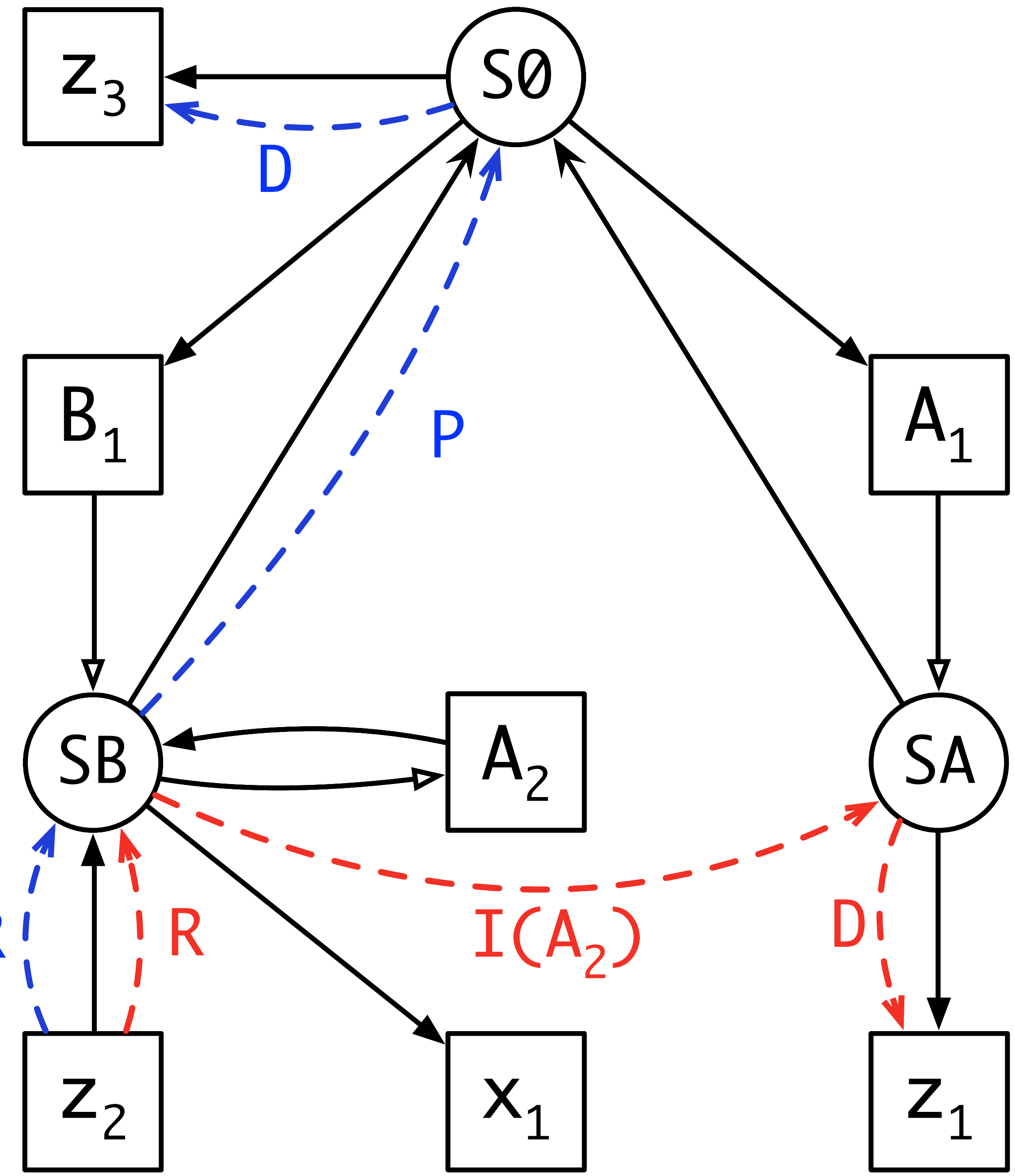

R2
SR
Associated scope
S
R1
Import


S
R1
R2
SR
I(R1)
Import Step

Imports shadow Parents

```
def z3 = 2                       S0

module A1 {
   def z1 = 5                    SA
}
module B1 {
   import A2                     SB
   def x1 = 1 + z2
}
```



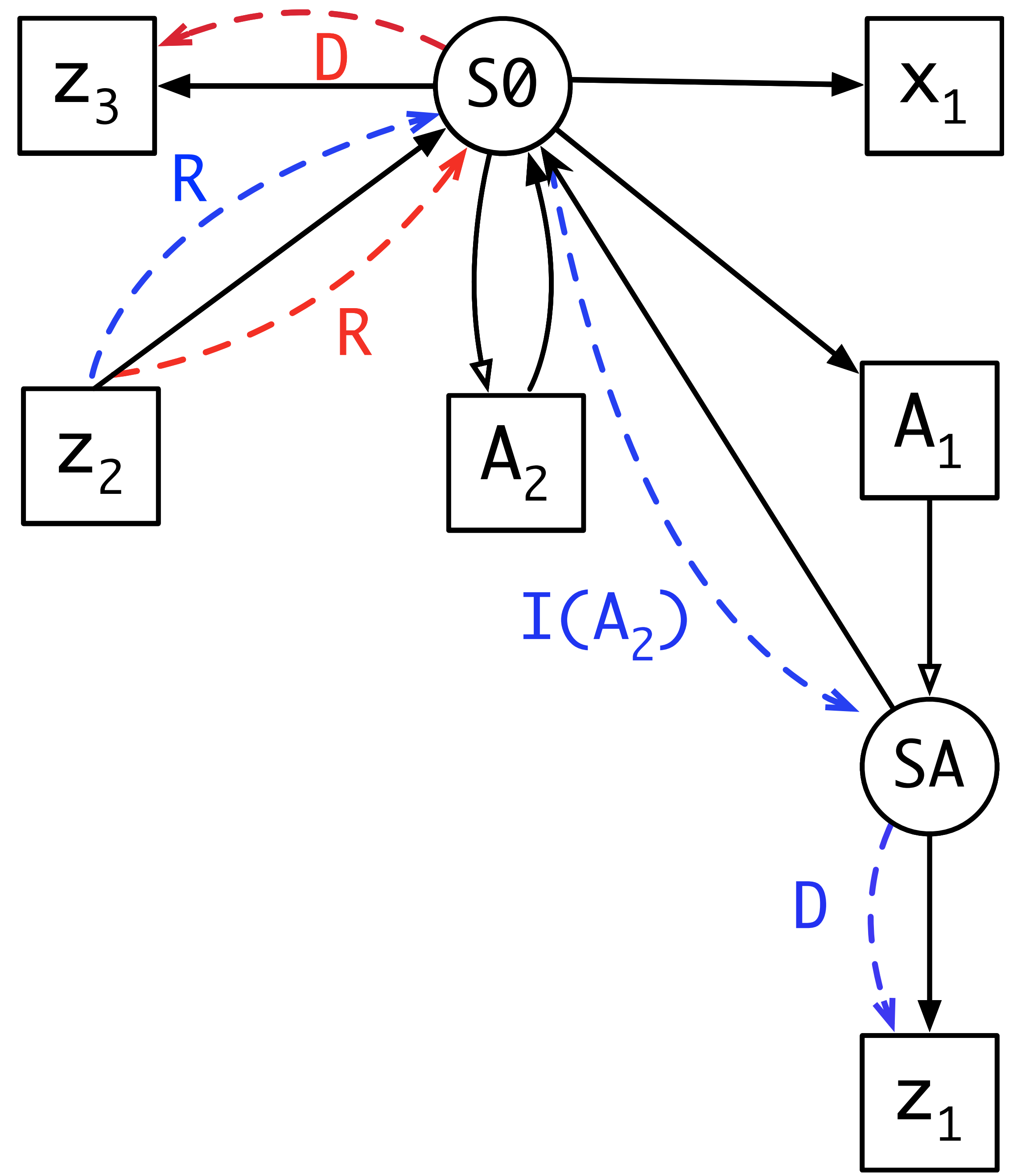
$$\frac{}{\mathrm{I}(\_).p' < \mathrm{P}.p} \quad \Rightarrow \quad \mathrm{R}.\mathrm{I}(A_2).\mathrm{D} < \mathrm{R}.\mathrm{P}.\mathrm{D}$$

Imports vs. Includes

```
def z3 = 2                S0

module A1 {
  def z1 = 5              SA
}

import A2
include A2
def x1 = 1 + z2
```

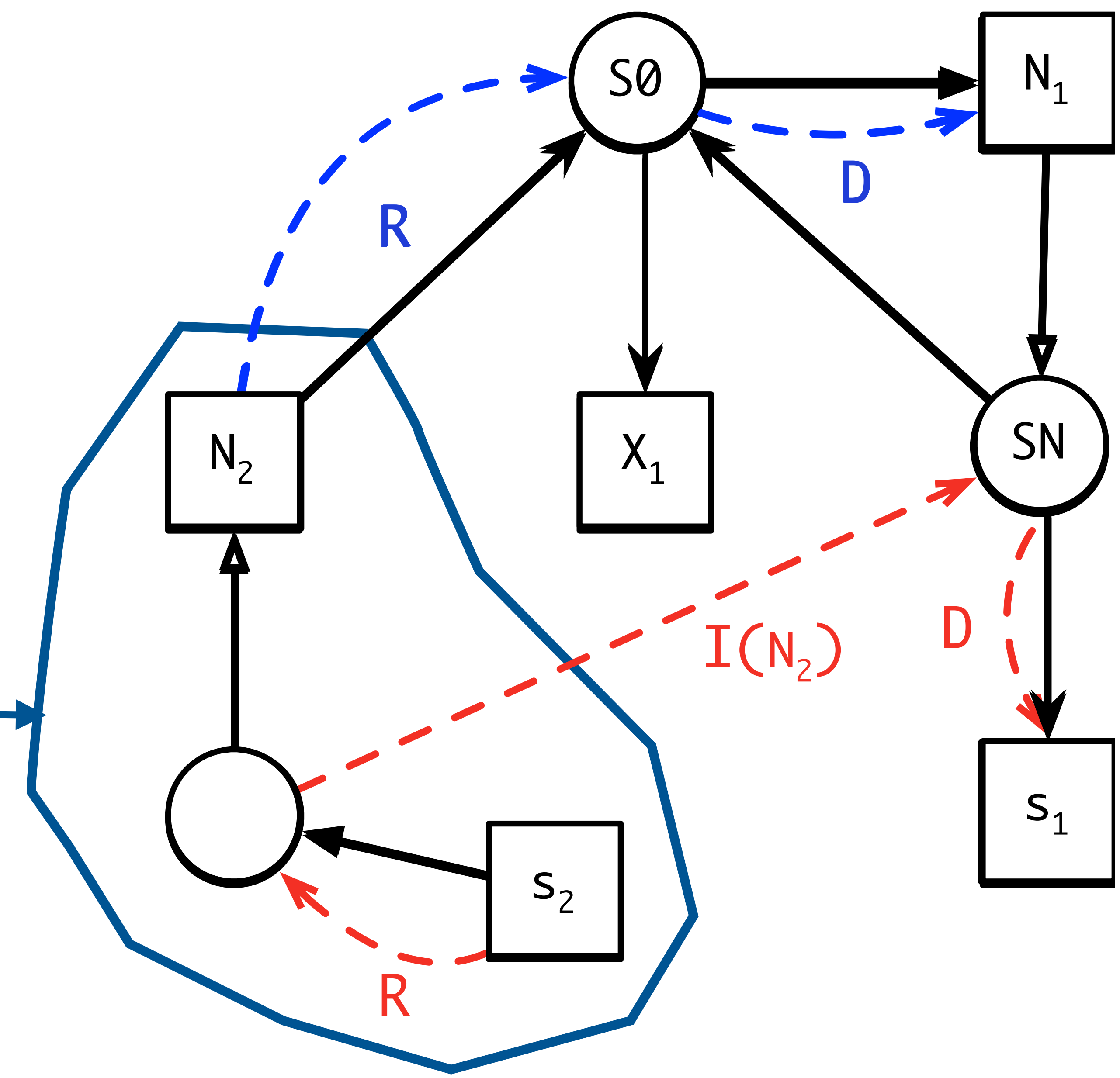


$$\cancel{\frac{}{D < I(\_).p'}} \quad \Rightarrow \quad R.D < R.I(A_2).D$$

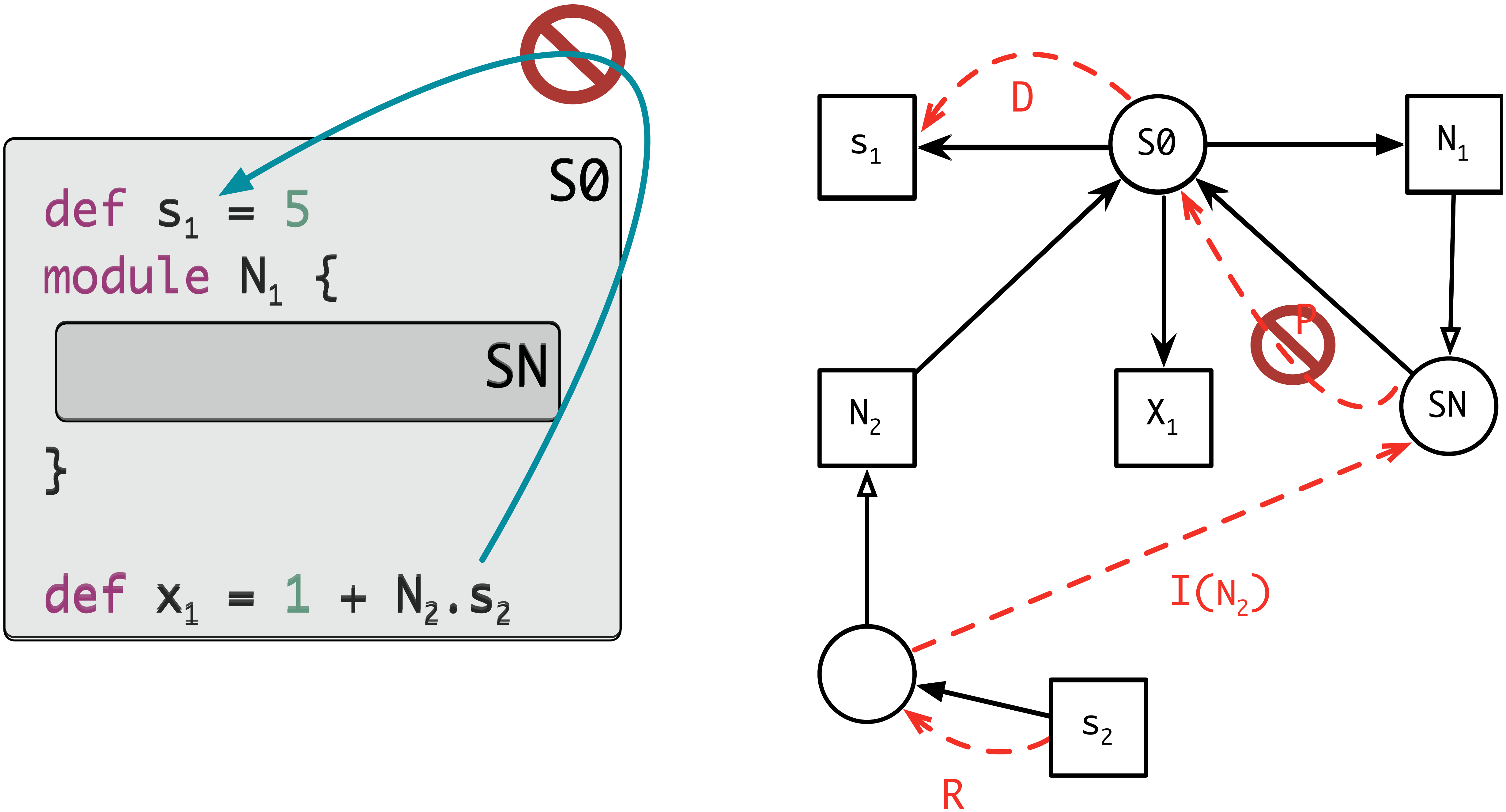
Qualified Names

```
module N1 {                 S0
    def s1 = 5
}

module M1 {
    def x1 = 1 + N2.s2
}
```



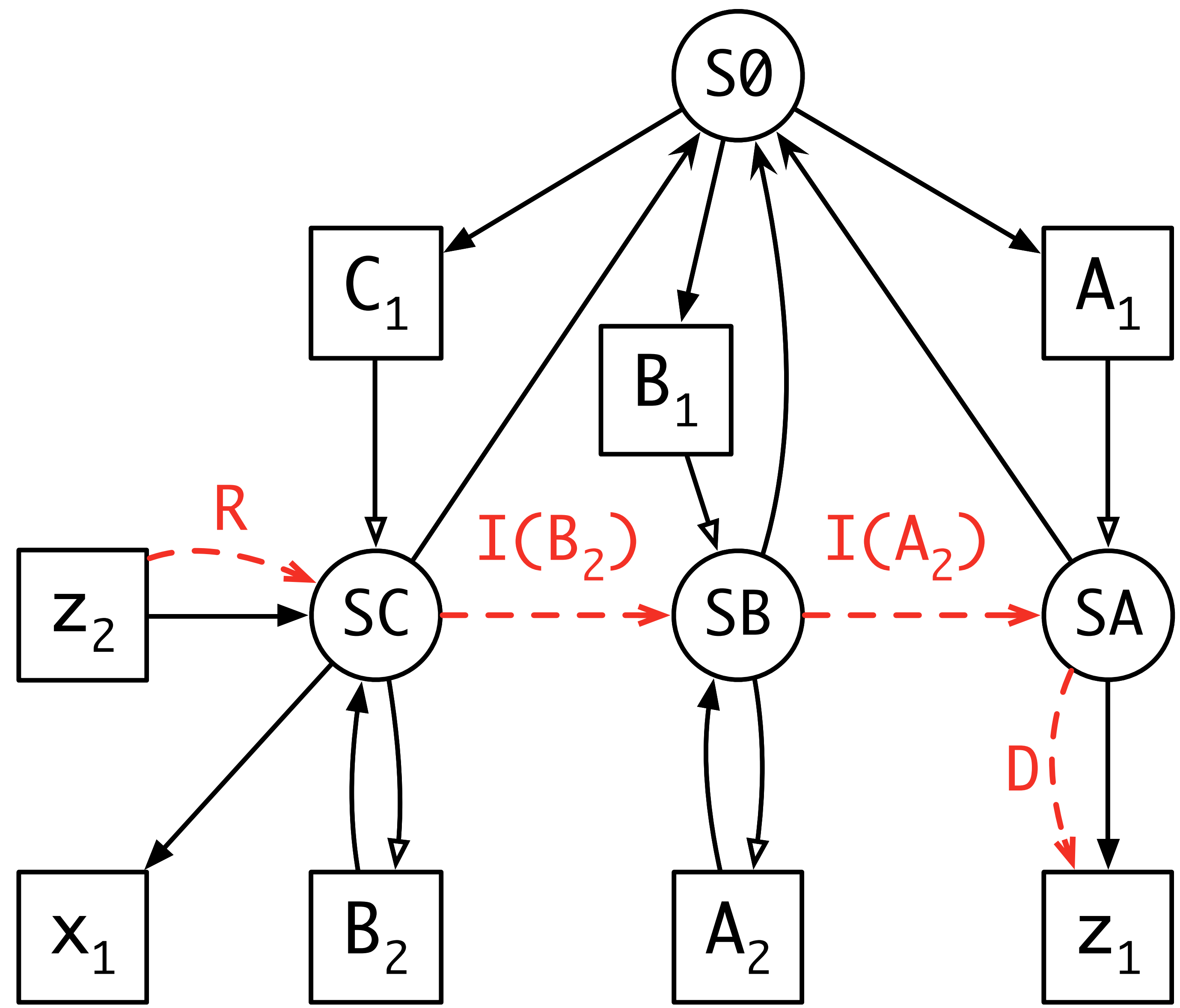
Import Parents



Well formed path: R.P*.I(_)*.D

Transitive vs. Non-Transitive

```
module A1 {
    def z1 = 5              SA
}              ??
module B1 {
    import A2               SB
}
module C1 {
    import B2               SC
    def x1 = 1 + z2
}
```

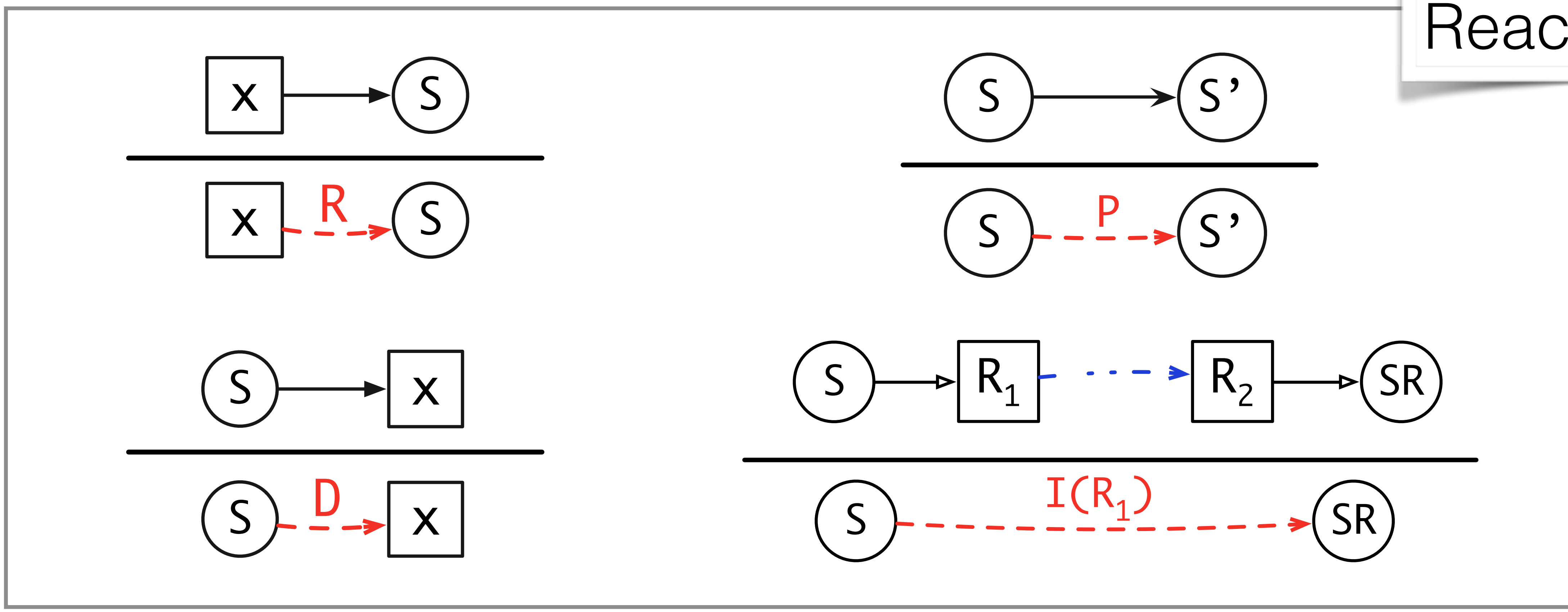


With transitive imports, a well formed path is `R.P*.I(_)*.D`

With non-transitive imports, a well formed path is `R.P*.I(_)?.D`

A Calculus for Name Resolution

Reachability

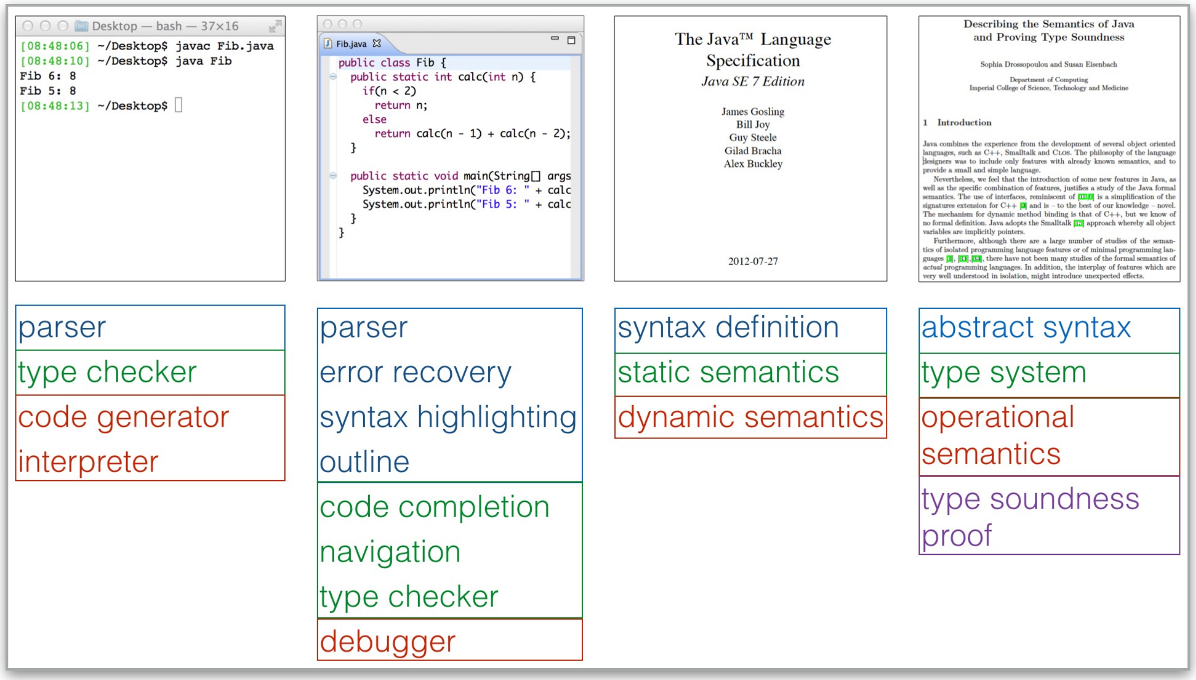


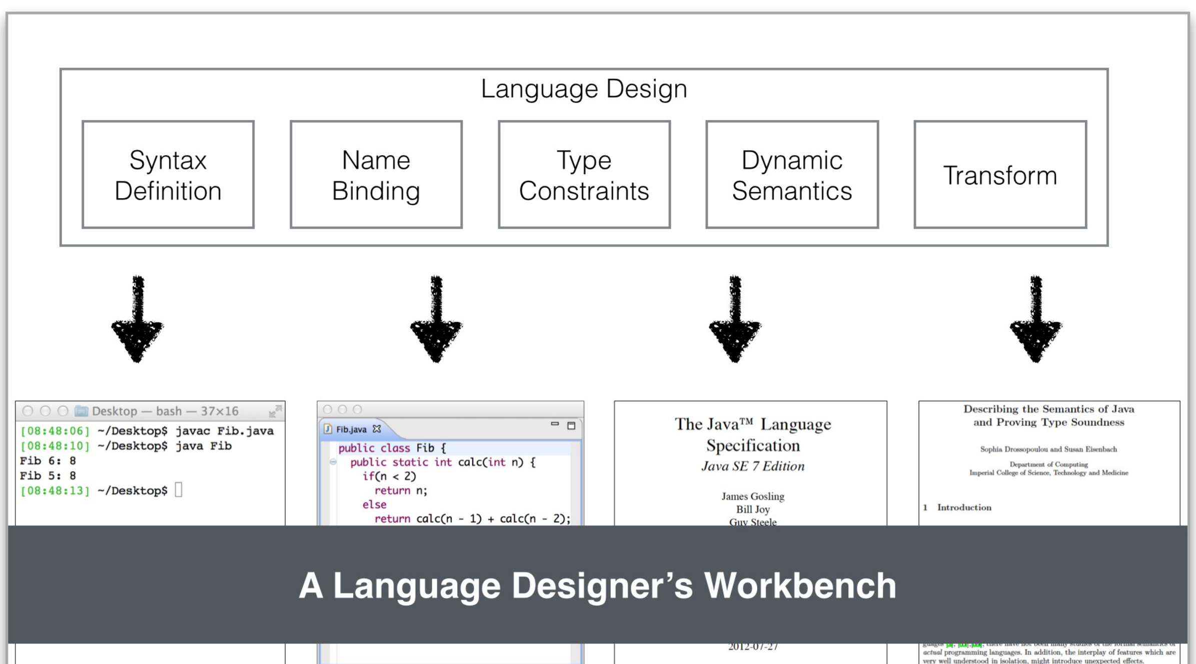
Well formed path: `R.P*.I(_)*.D`

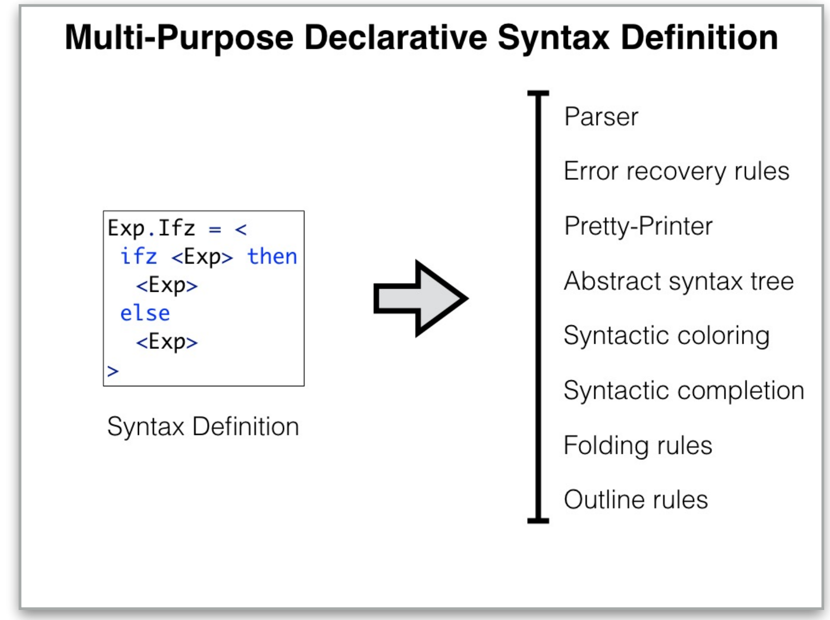
Visibility

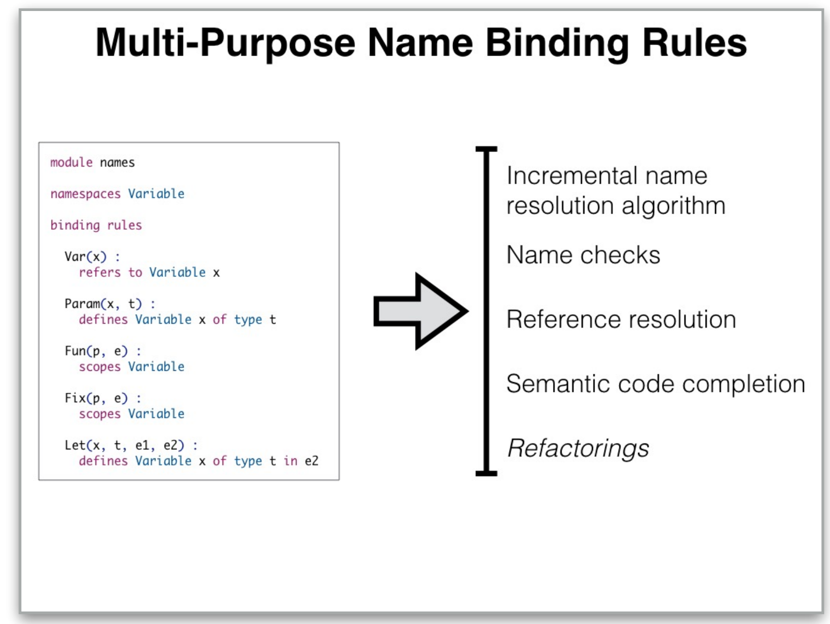
$$\frac{}{D < P.p} \qquad \frac{}{I(\_).p' < P.p}$$

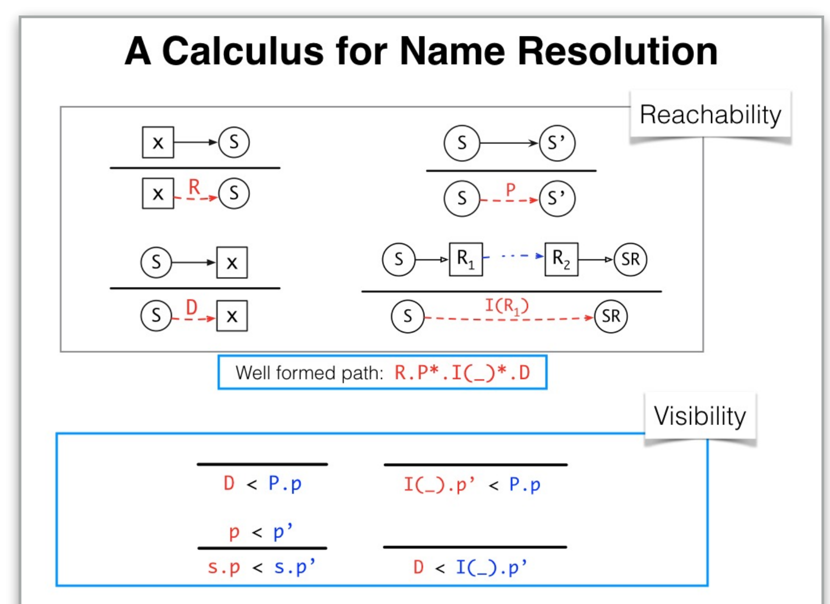
$$\frac{p < p'}{s.p < s.p'} \qquad \frac{}{D < I(\_).p'}$$


The Java™ Language Specification
Java SE 7 Edition
James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
2012-07-27
Describing the Semantics of Java and Proving Type Soundness
parser
type checker
code generator
interpreter
parser
error recovery
syntax highlighting
outline
code completion
navigation
type checker
debugger
syntax definition
static semantics
dynamic semantics
abstract syntax
type system
operational semantics
type soundness proof


Language Design
Syntax Definition
Name Binding
Type Constraints
Dynamic Semantics
Transform
A Language Designer's Workbench


Multi-Purpose Declarative Syntax Definition
Exp.Ifz = <
ifz <Exp> then
<Exp>
else
<Exp>
>
Syntax Definition
Parser
Error recovery rules
Pretty-Printer
Abstract syntax tree
Syntactic coloring
Syntactic completion
Folding rules
Outline rules


Multi-Purpose Name Binding Rules
module names
namespaces Variable
binding rules
Var(x) :
refers to Variable x
Param(x, t) :
defines Variable x of type t
Fun(p, e) :
scopes Variable
Fix(p, e) :
scopes Variable
Let(x, t, e1, e2) :
defines Variable x of type t in e2
Incremental name resolution algorithm
Name checks
Reference resolution
Semantic code completion
Refactorings


A Calculus for Name Resolution
Reachability
Well formed path: R.P*.I(_)*.D
Visibility
D < P.p
I(_).p' < P.p
p < p'
s.p < s.p'
D < I(_).p'