

# 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-1) + fib(n-2)
}
```



```
Desktop — bash — 37x16
[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(6));
    System.out.println("Fib 5: " + calc(5));
  }
}
```

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] 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 [12] 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, [3], [3]], 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.

```

Desktop — bash — 37x16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$

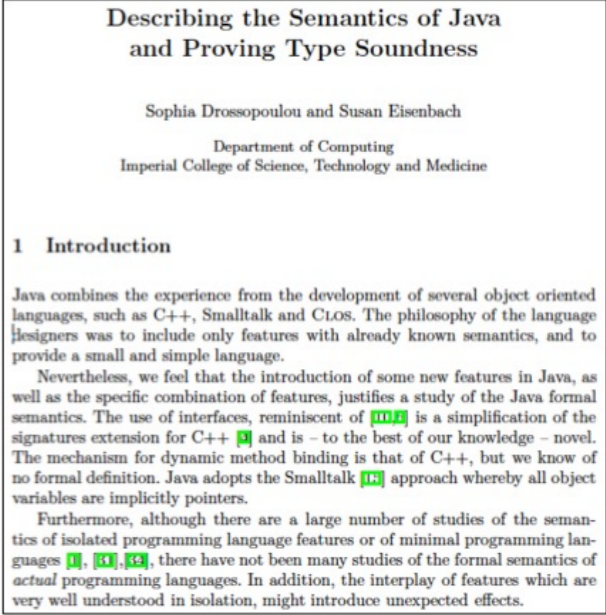
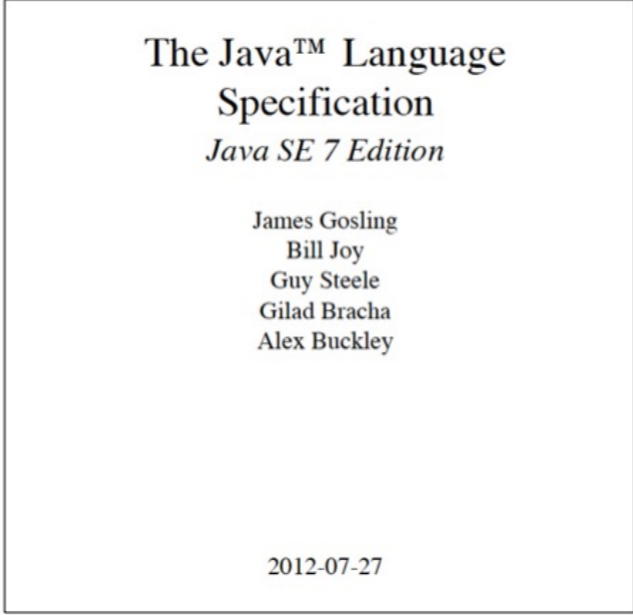
```

```

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(6));
        System.out.println("Fib 5: " + calc(5));
    }
}

```



parser

type checker

code generator

interpreter

parser

error recovery

syntax highlighting

outline

code completion

navigation

type checker

debugger

syntax definition

static semantics

dynamic semantics

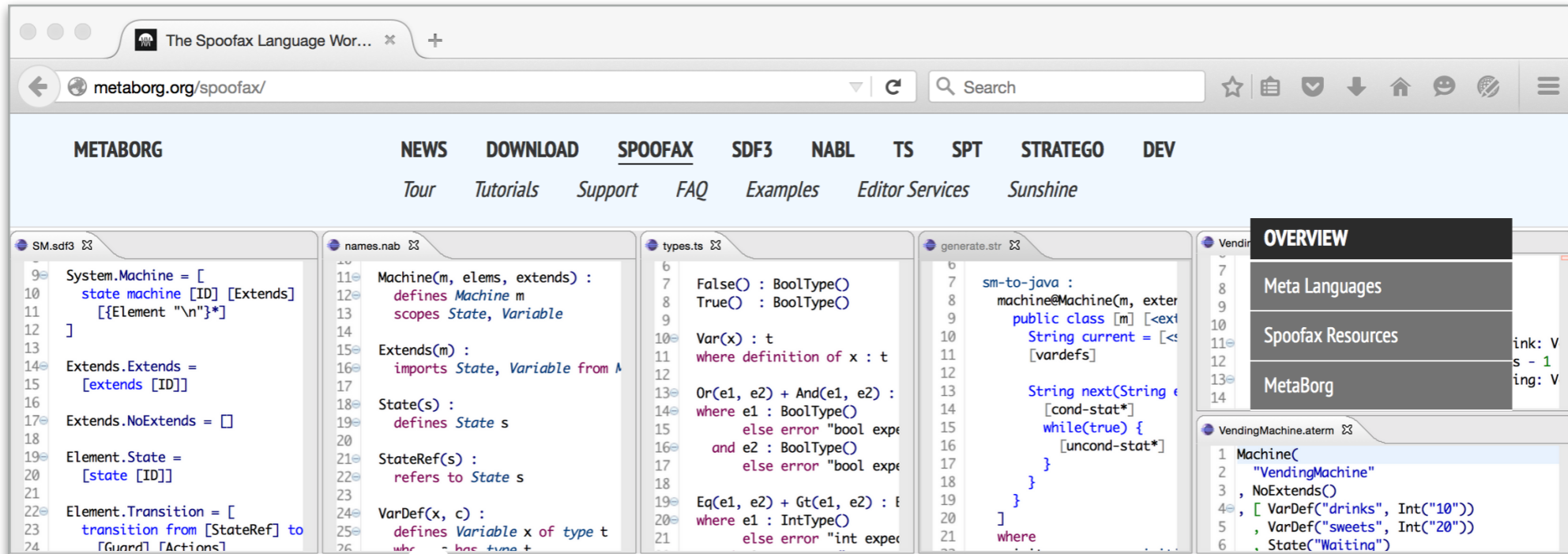
abstract syntax

type system

operational semantics

type soundness proof

# Spoofax Language Workbench



The screenshot shows a web browser window with the URL `metaborg.org/spoofax/`. The navigation menu includes **METABORG**, **NEWS**, **DOWNLOAD**, **SPOOFAX**, **SDF3**, **NABL**, **TS**, **SPT**, **STRATEGO**, and **DEV**. Below the menu are links for *Tour*, *Tutorials*, *Support*, *FAQ*, *Examples*, *Editor Services*, and *Sunshine*. The main content area displays several code editors with the following code snippets:

```
SM.sdf3
9 System.Machine = [
10   state machine [ID] [Extends]
11   [{Element "\n"}*]
12 ]
13
14 Extends.Extends =
15   [extends [ID]]
16
17 Extends.NoExtends = []
18
19 Element.State =
20   [state [ID]]
21
22 Element.Transition = [
23   transition from [StateRef] to
24   [Guard] [Actions]
25 ]

names.nab
11 Machine(m, elems, extends) :
12   defines Machine m
13   scopes State, Variable
14
15 Extends(m) :
16   imports State, Variable from A
17
18 State(s) :
19   defines State s
20
21 StateRef(s) :
22   refers to State s
23
24 VarDef(x, c) :
25   defines Variable x of type t
26   where x has type t

types.ts
6
7 False() : BoolType()
8 True() : BoolType()
9
10 Var(x) : t
11   where definition of x : t
12
13 Or(e1, e2) + And(e1, e2) :
14   where e1 : BoolType()
15   else error "bool expected"
16   and e2 : BoolType()
17   else error "bool expected"
18
19 Eq(e1, e2) + Gt(e1, e2) :
20   where e1 : IntType()
21   else error "int expected"

generate.str
6
7 sm-to-java :
8   machine@Machine(m, exte
9   public class [m] [<ext
10   String current = [<
11   [vardefs]
12
13 String next(String e
14   [cond-stat*]
15   while(true) {
16   [uncond-stat*]
17   }
18   }
19   ]
20   ]
21   where

VendingMachine.aterm
1 Machine(
2   "VendingMachine"
3   , NoExtends()
4   , [ VarDef("drinks", Int("10"))
5     , VarDef("sweets", Int("20"))
6     , State("Waiting")
```

**The Spoofax Language Workbench**

Spoofax is a platform for developing textual domain-specific languages with full-featured [Eclipse](#) editor plugins.

With the Spoofax language workbench, you can write the grammar of your language using the high-level SDF grammar formalism. Based on this grammar, basic editor services such as syntax highlighting and code folding are automatically provided. Using high-level descriptor languages, these services can be customized. More sophisticated services such as error marking and content completion can be specified using rewrite rules in the Stratego language.

### Meta Languages

Language definitions in Spoofax are constructed using the following meta-languages:

- The [SDF3](#) syntax definition formalism
- The [NaBL](#) name binding language
- The [TS](#) type specification language
- The [Stratego](#) transformation language

# Language Engineering

Syntax  
Checker

Name  
Resolver

Type  
Checker

Code  
Generator



```
Desktop — bash — 37x16
[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(6));
        System.out.println("Fib 5: " + calc(5));
    }
}
```

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] 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 [12] 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, 11, 13], 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.

## Language Design

Syntax  
Definition

Name  
Binding

Type  
Constraints

Dynamic  
Semantics

Transform



```
Desktop — bash — 37x16
[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);
  }
}
```

The Java™ Language  
Specification  
*Java SE 7 Edition*

James Gosling  
Bill Joy  
Guy Steele

Describing the Semantics of Java  
and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach  
Department of Computing  
Imperial College of Science, Technology and Medicine

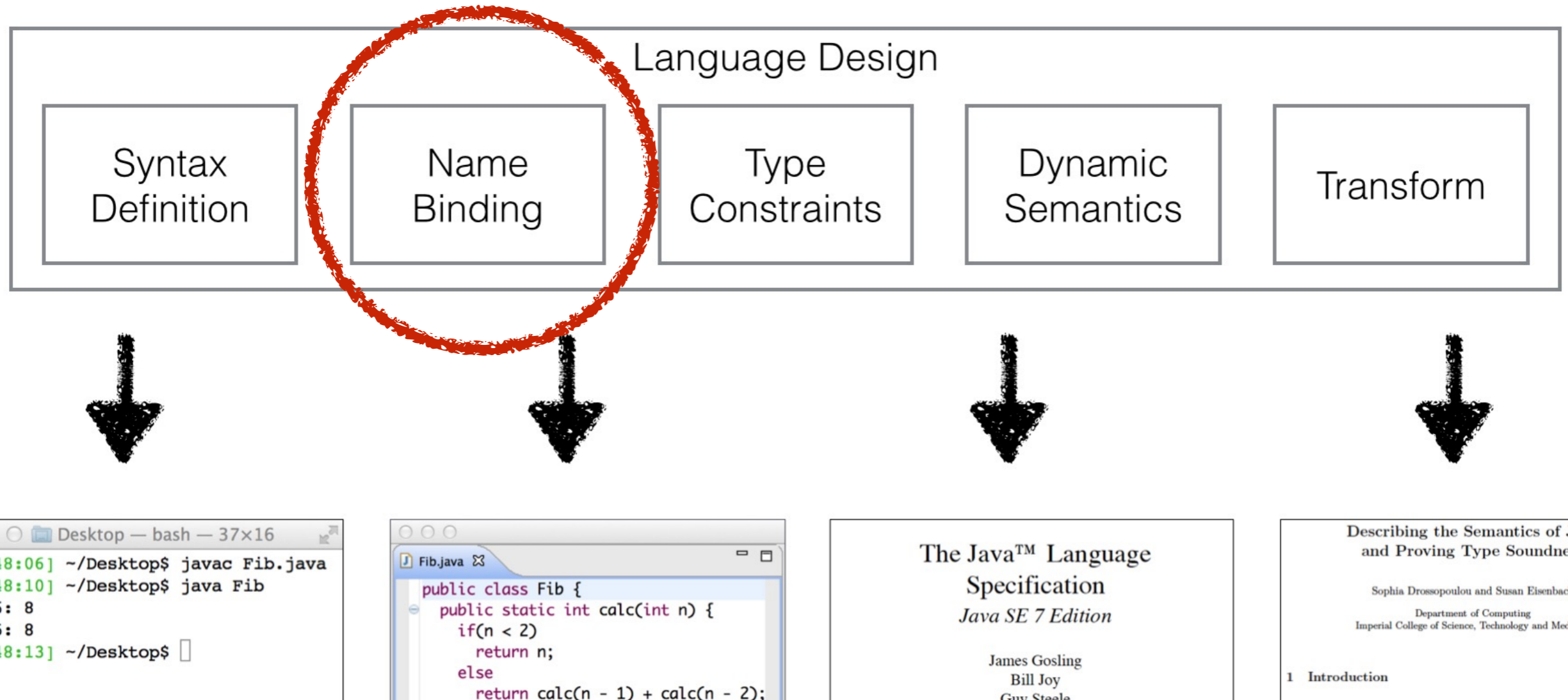
1 Introduction

# A Language Designer's Workbench

2012-07-27

languages [14], [15], [16], 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.

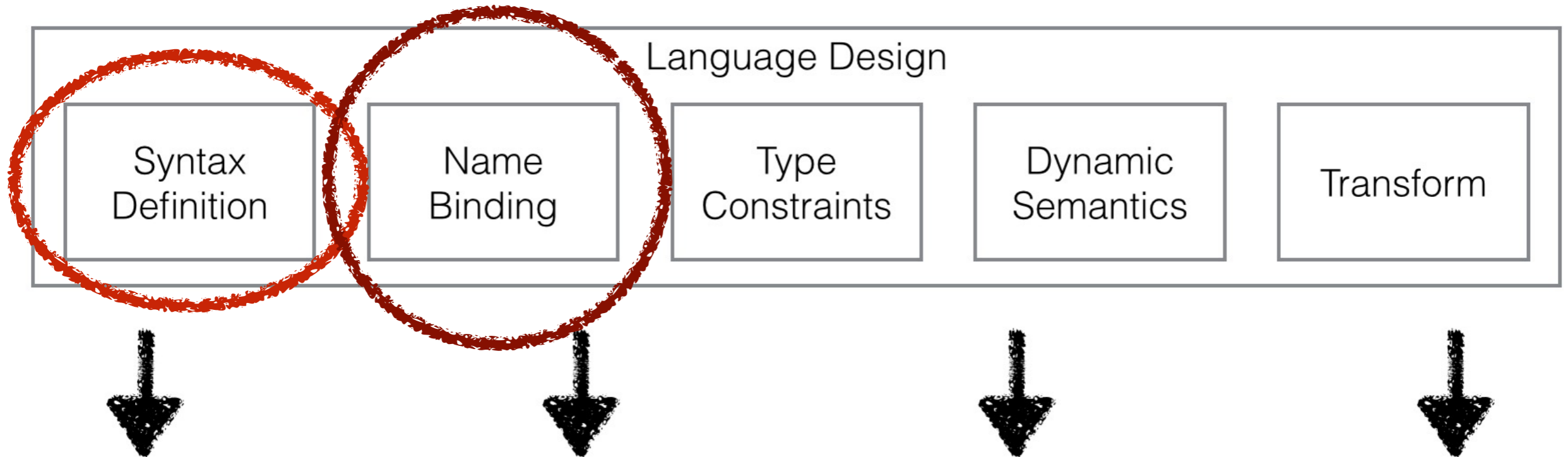
# A Theory of Name Resolution



## A Language Designer's Workbench

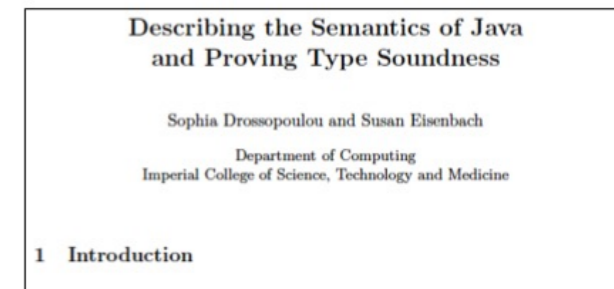
[Neron, Tolmach, Visser, Wachsmuth et al.; ESOP 2015]

# Declarative Syntax Definition



```
Desktop — bash — 37x16
[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);
    }
}
```



## A Language Designer's Workbench

2012-07-27

languages [1], [2], [3], 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.

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



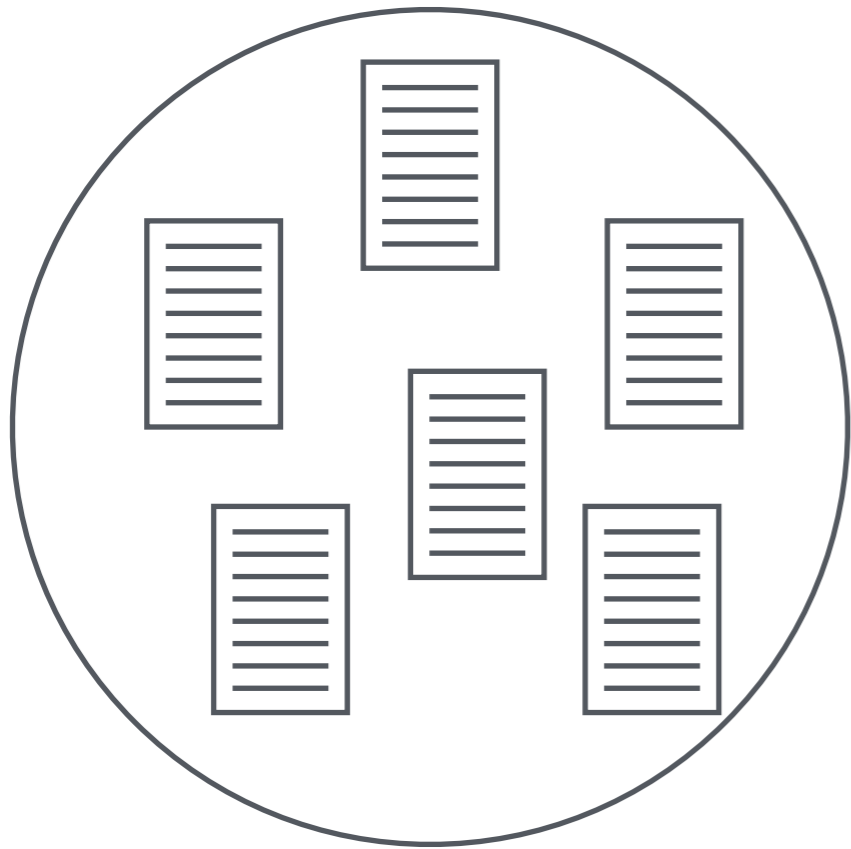
---

# Declare Your Syntax

---

[Kats, Visser, Wachsmuth; Onward 2010]

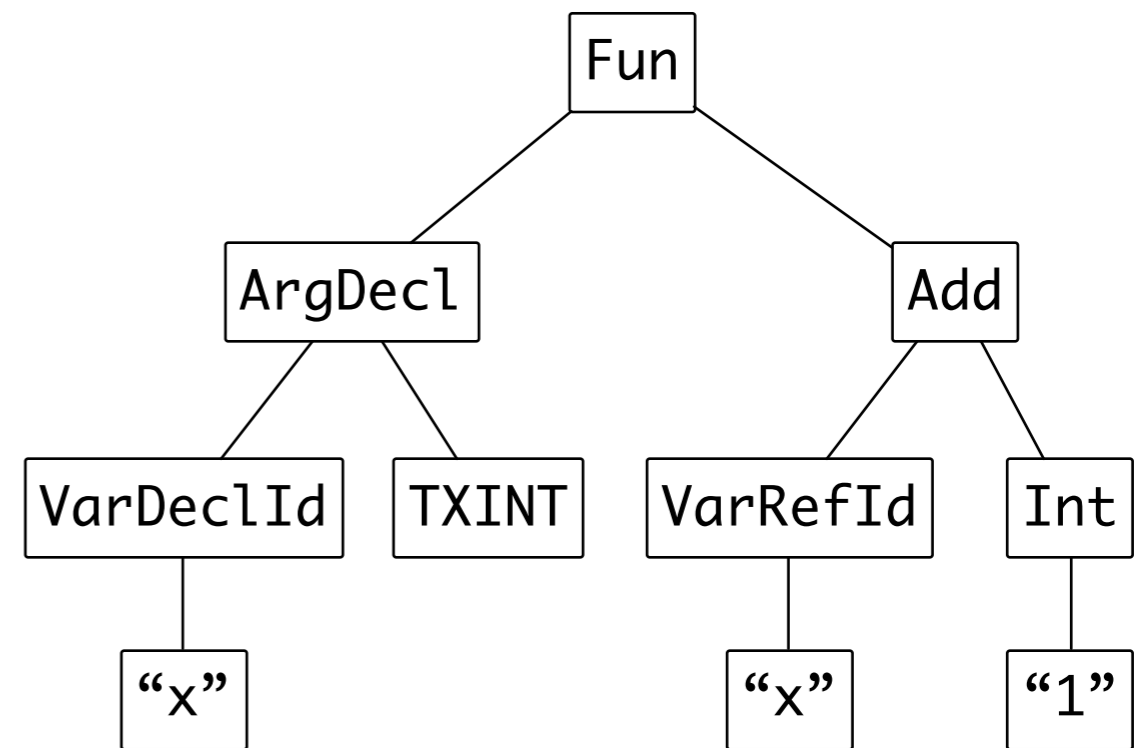
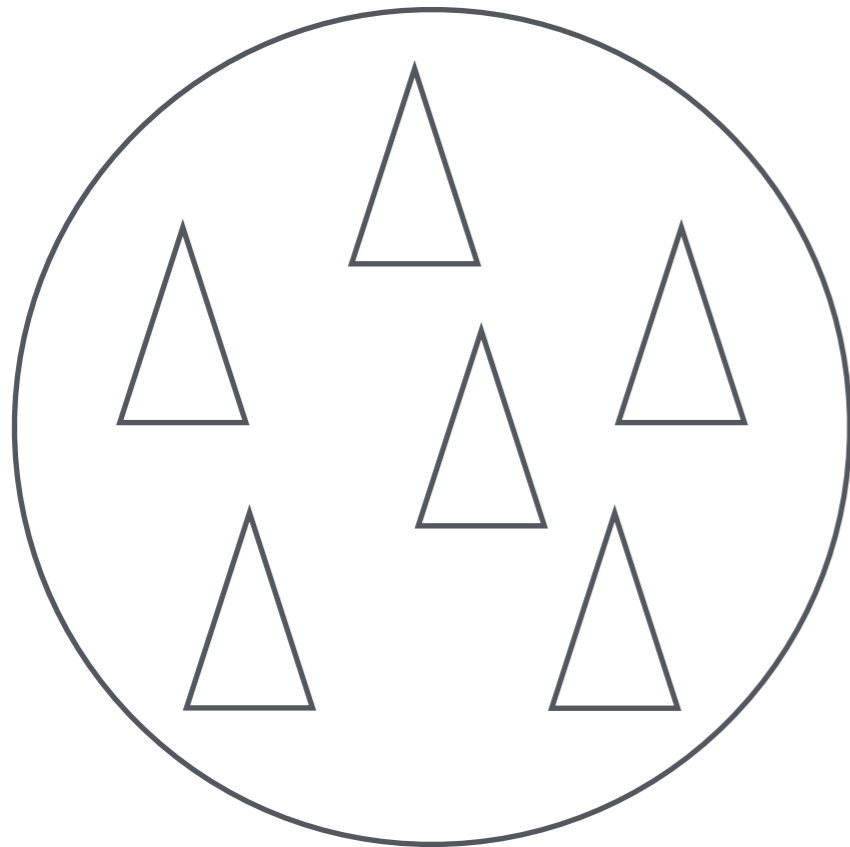
# Language = Set of Sentences



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

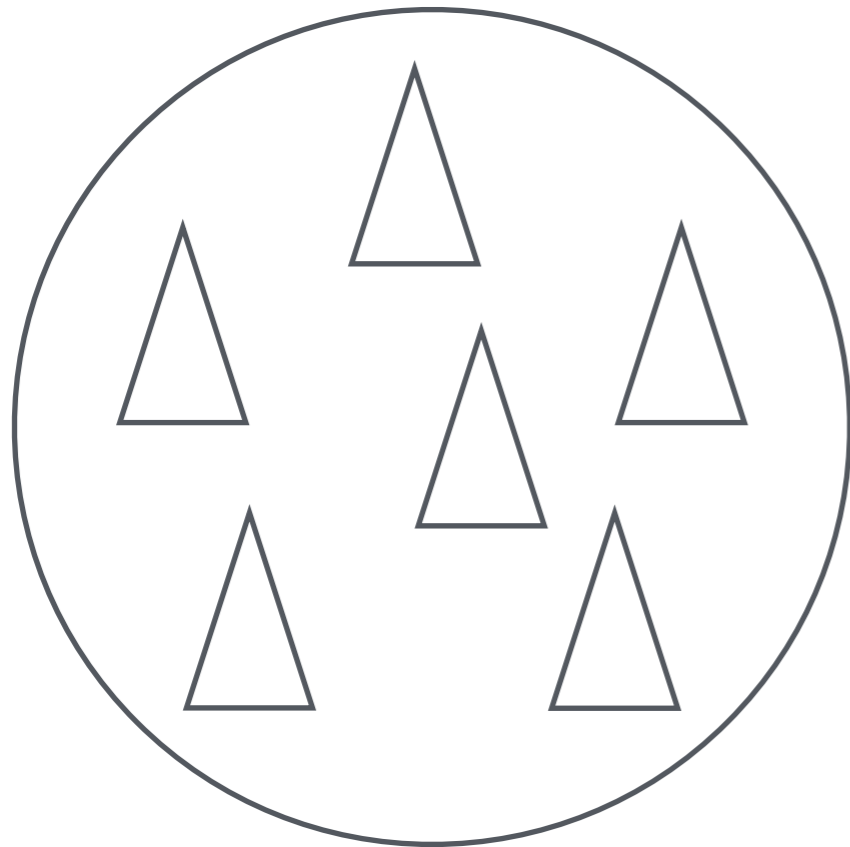
text is a convenient interface for writing and reading programs

# Language = Set of Trees



tree is a convenient interface for transforming programs

# Tree Transformation

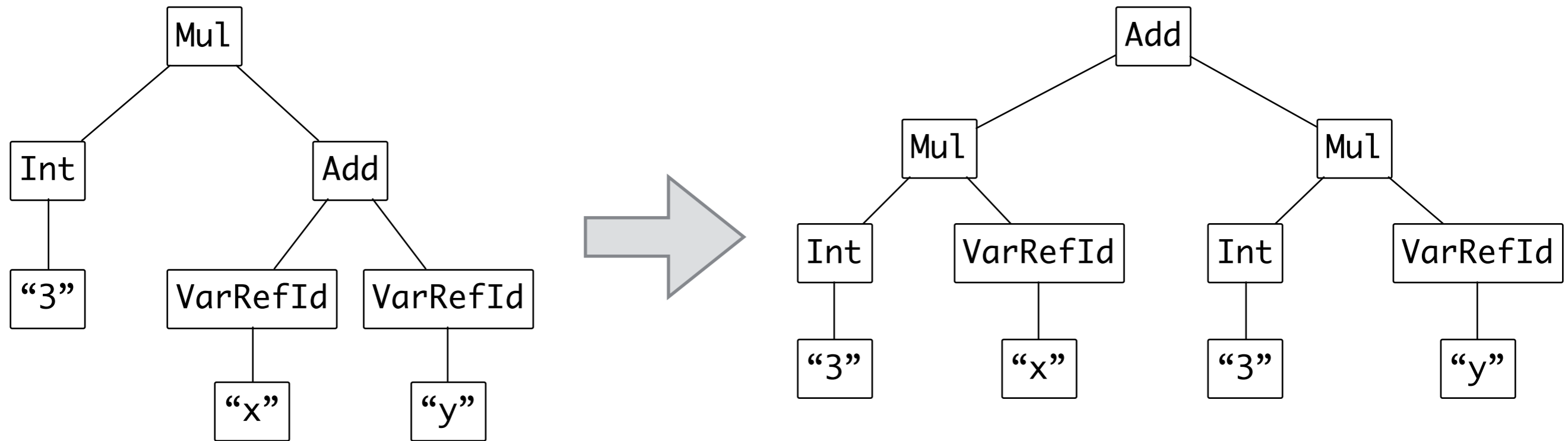


**Syntactic**  
coloring  
outline view  
completion

**Semantic**  
transform  
translate  
eval  
analyze  
refactor  
type check

tree is a convenient interface for transforming programs

# Tree Transformation

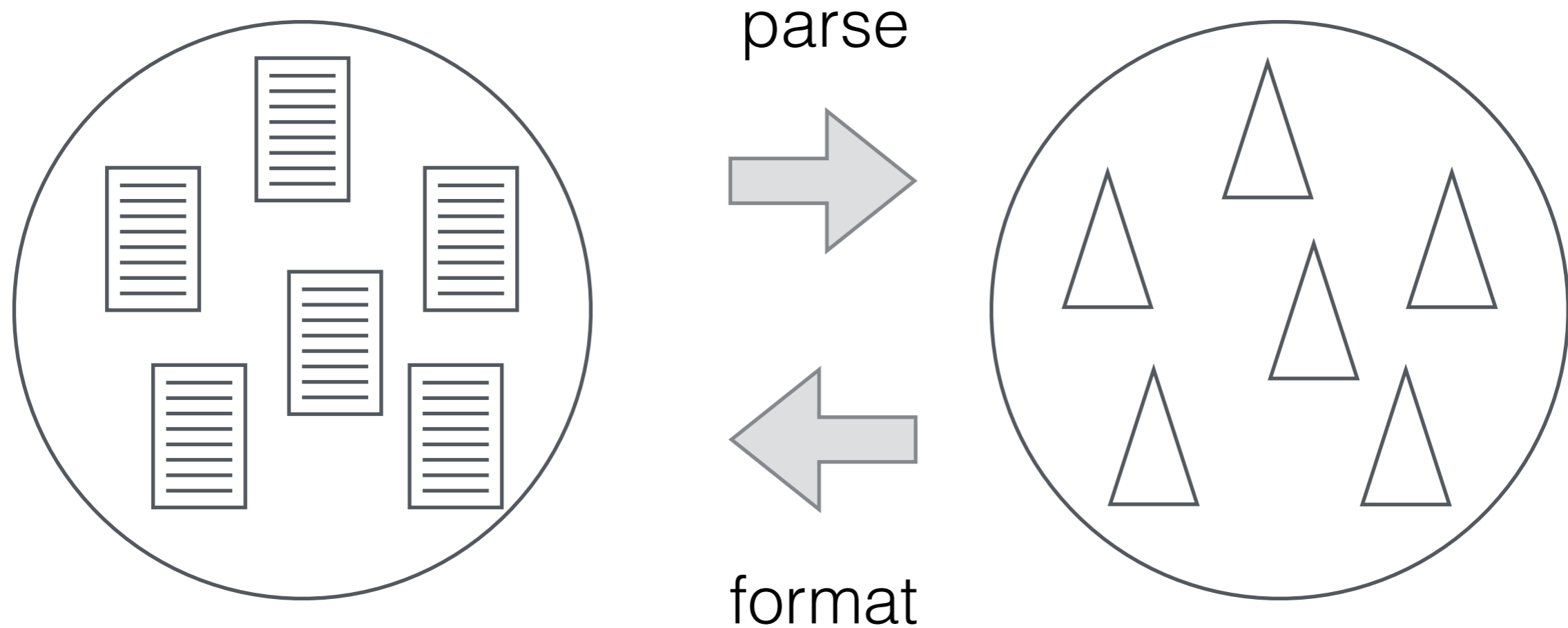


```
Mul(Int("3"),  
     Add(VarRefId("x"),  
         VarRefId("y")))
```

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

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

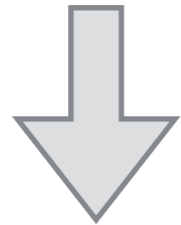
# Language = Sentences *and* Trees



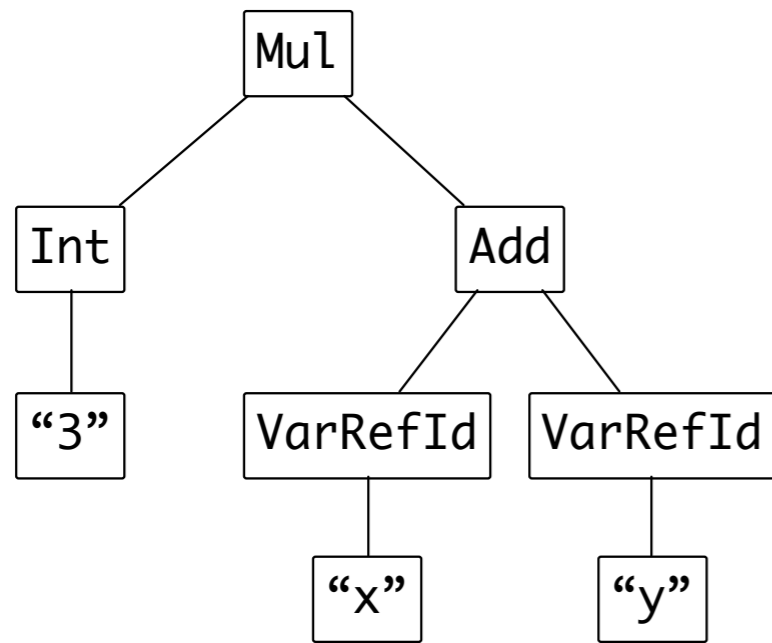
different representations convenient for different purposes

# From Text to Tree and Back

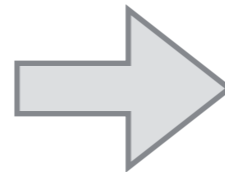
3 \* (x + y)



*parse*

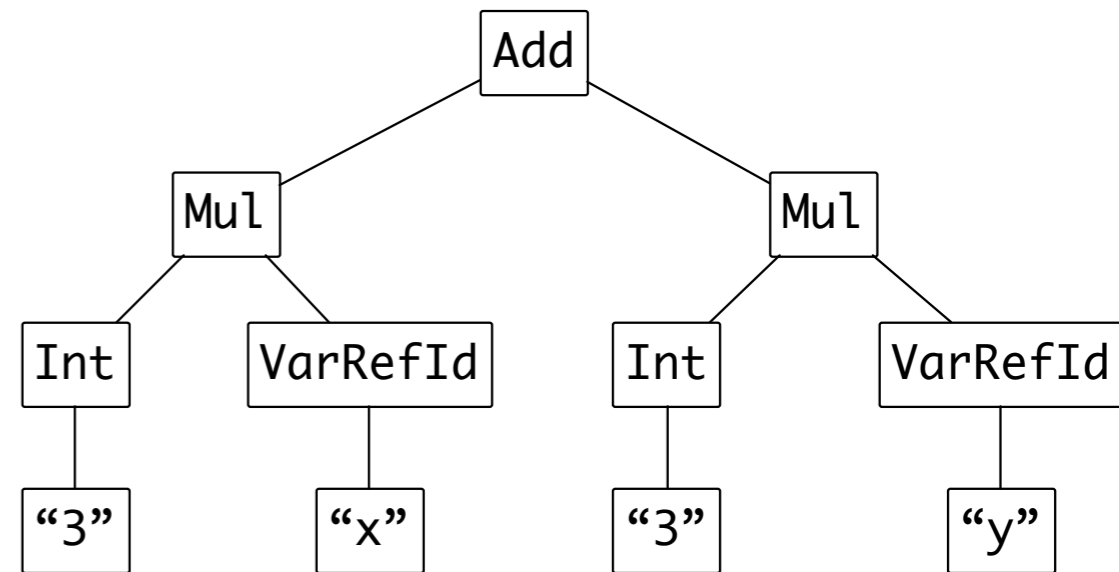


*transform*



(3 \* x) + (3 \* y)

*format*



# 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

Java - metaborg-lmr/examples/test01.lmr - Eclipse - /Users/eelcovisser/03-Research/workspace-dyl

Package Explorer

- record01.partition.index
- test00.aterm
- test00.lmr
- test00.partition.index
- test00.pp.lmr
- test01.aterm
- test01.lmr
- icons
- include
- lib
- META-INF
- src-gen
- syntax
  - Common.sdf3
  - Expressions.sdf3
  - ExpressionsAmb.sdf3
  - Id.sdf3
  - Identifiers.sdf3
  - LMR.sdf3
  - Records.sdf3
  - Types.sdf3
- target
- trans
- utils
  - .classpath
  - .gitignore
  - .project
  - build.generated.xml
  - build.main.xml
  - build.properties
  - plugin.xml
  - pom.xml

ExpressionsAmb.sdf3

```
10 Expr.True = <true>
11 Expr.False = <false>
12
13 Expr = <<VarRef>>
14
15 Expr.Add = <<Expr> + <Expr>>
16 Expr.Sub = <<Expr> - <Expr>>
17 Expr.Mul = <<Expr> * <Expr>>
18 Expr.Div = <<Expr> / <Expr>>
19 Expr.And = <<Expr> & <Expr>>
20 Expr.Or = <<Expr> | <Expr>>
21 Expr.Eq = <<Expr> == <Expr>>
22 Expr.App = <<Expr> <Expr>>
23
24 Expr.If = <
25   if <Expr> then
26     <Expr>
27   else
28     <Expr>
29 > {longest-match}
30
31 Expr.Fun = <fun (<ArgDecl>) { <Expr> }>
32 ArgDecl.ArgDecl = <<VarId> : <Type>>
33
34 Expr.Let = <let <DefBind+> in <Expr>>
35 Expr.LetRec = <letrec <DefBind+> in <Expr>>
36 Expr.LetPar = <letpar <DefBind+> in <Expr>>
37
38 DefBind.DefBind = <<VarId> = <Expr>>
39 DefBind.DefBindTyped = <<VarId> : <Type> = <Expr>>
40
```

amb01.lmr

```
1 program test01
2
3 module A {
4   def x = 1
5 }
6
7 module B {
8   import A
9   def y = x + 1
10 }
11
```

test01.aterm

```
1 Program(
2   "test01"
3   , [ Module("A", [ Def(DefBind("x", Int("1")))] ) ]
4   , Module(
5     "B"
6     , [ Import(ModRef("A"))
7       , Def(DefBind("y", Add(VarRef("x"), Int("1")))]
8     ]
9   )
10 ]
11 )
```

Writable Smart Insert 5 : 2 Analyzing files (legacy)

# Ambiguity

Java - metaborg-lmr/examples/amb01.aterm - Eclipse - /Users/eelcovisser/03-Research/workspace-dyl

ExpressionsAmb.sdf3 Records.sdf3 LMR.sdf3 \*amb01.lmr test01.lmr test01.aterm amb01.aterm

```
10 Expr.True = <true>
11 Expr.False = <false>
12
13 Expr = <<VarRef>>
14
15 Expr.Add = <<Expr> + <Expr>
16 Expr.Sub = <<Expr> - <Expr>
17 Expr.Mul = <<Expr> * <Expr>
18 Expr.Div = <<Expr> / <Expr>
19 Expr.And = <<Expr> & <Expr>
20 Expr.Or = <<Expr> | <Expr>
21 Expr.Eq = <<Expr> == <Expr>
22 Expr.App = <<Expr> <Expr>
23
24 Expr.If = <
25   if <Expr> then
26     <Expr>
27   else
28     <Expr>
29 > {longest-match}
30
31 Expr.Fun = <fun (<ArgDecl>) { <Expr> }>
32 ArgDecl.ArgDecl = <<VarId> : <Type>>
33
34 Expr.Let = <let <DefBind+> in <Expr>>
35 Expr.LetRec = <letrec <DefBind+> in <Expr>>
36 Expr.LetPar = <letpar <DefBind+> in <Expr>>
37
38 DefBind.DefBind = <<VarId> = <Expr>>
39 DefBind.DefBindTyped = <<VarId> : <Type> = <Expr>>
40
```

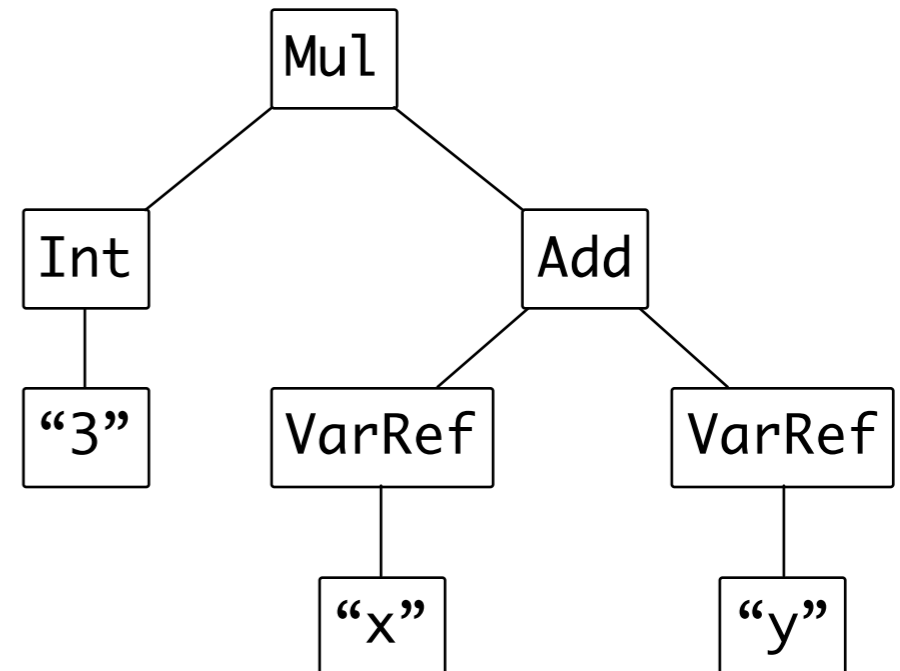
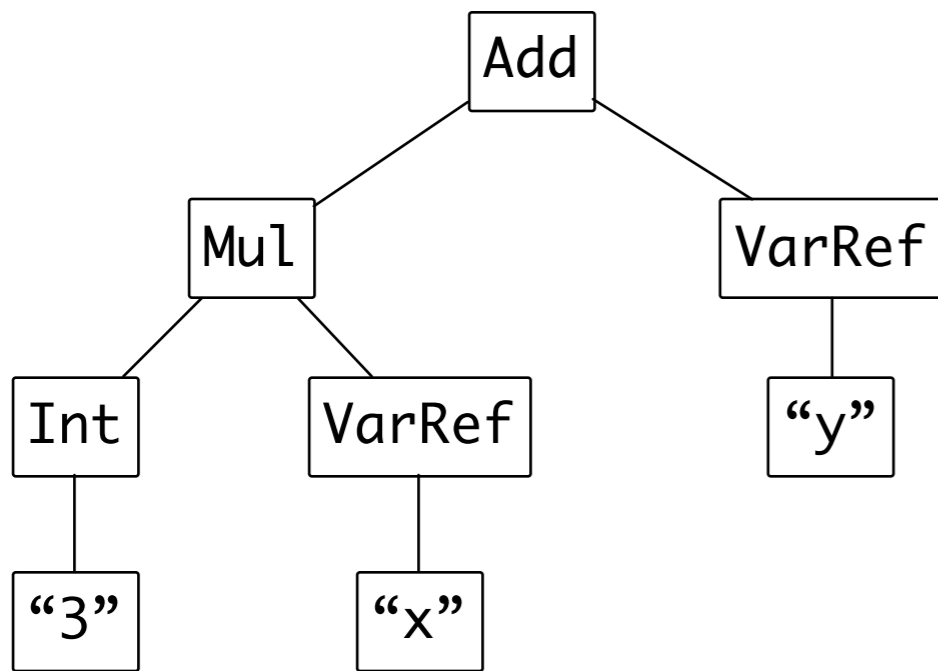
```
1 a + b * x - 1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

```
1 amb(
2   [ amb(
3     [ Sub(
4       amb(
5         [ Mul(Add(VarRef("a"), VarRef("b")), VarRef("x"))
6           , Add(VarRef("a"), Mul(VarRef("b"), VarRef("x")))
7         ]
8       )
9     , Int("1")
10   ]
11   , Add(
12     VarRef("a")
13   , amb(
14     [ Sub(Mul(VarRef("b"), VarRef("x")), Int("1"))
15       , Mul(VarRef("b"), Sub(VarRef("x"), Int("1")))
16     ]
17   )
18 )
19 ]
20 )
21 , Mul(
22   Add(VarRef("a"), VarRef("b"))
23   , Sub(VarRef("x"), Int("1"))
24 )
25 ]
26 )
```

Writable Smart Insert 10 : 10 Analyzing files (legacy)

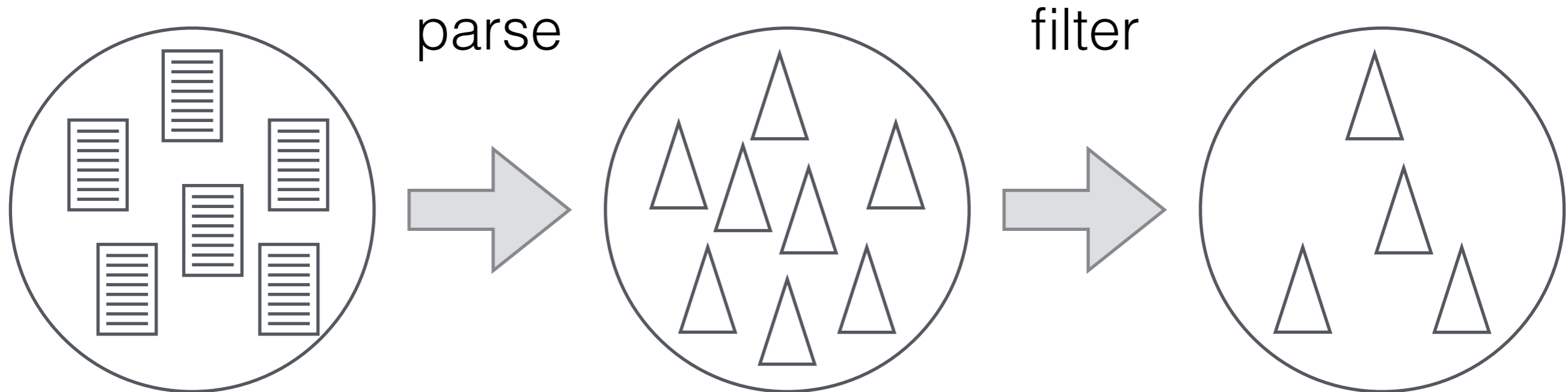
# Ambiguity

3 \* x + y



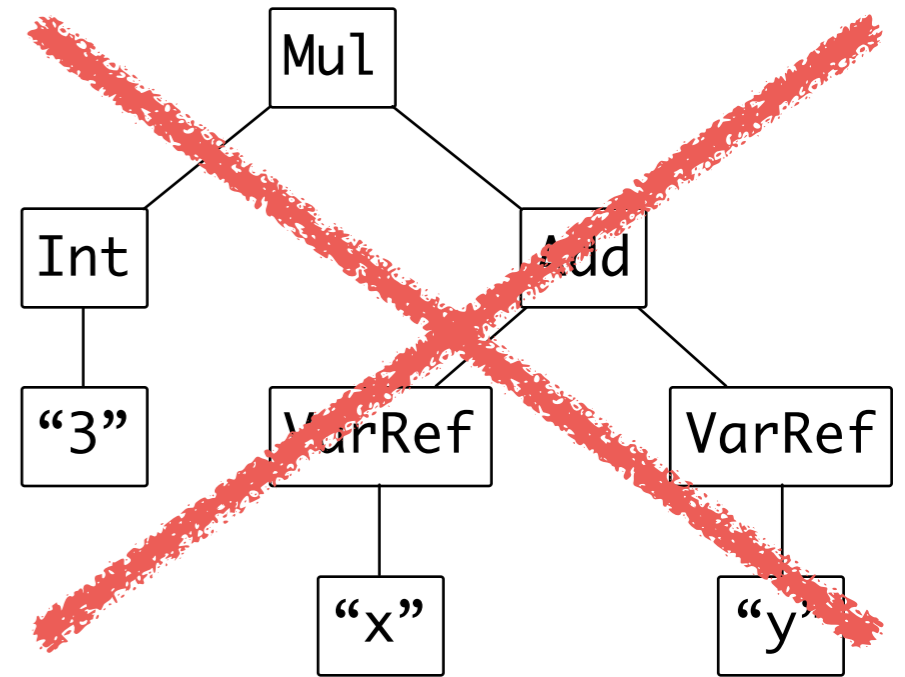
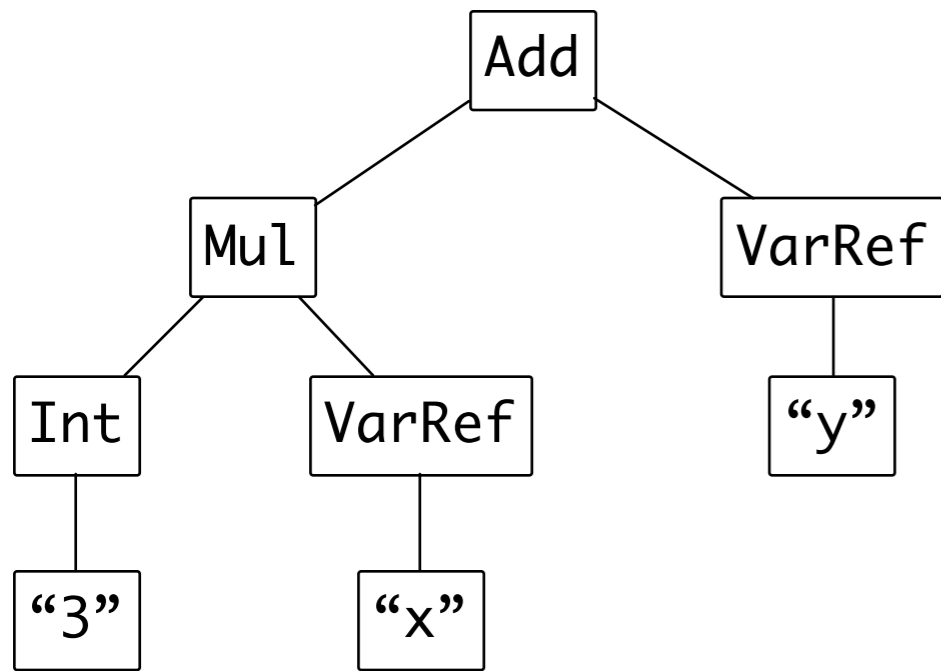
$t1 \neq t2 \wedge \text{format}(t1) = \text{format}(t2)$

# Declarative Disambiguation



# Priority and Associativity

3 \* x + y



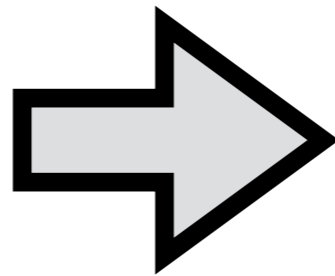
context-free syntax  
Expr.Int = INT  
Expr.Add = <<Expr> + <Expr> {left}  
Expr.MuL = <<Expr> \* <Expr> {left}

context-free priorities  
Expr.MuL > Expr.Add

# 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



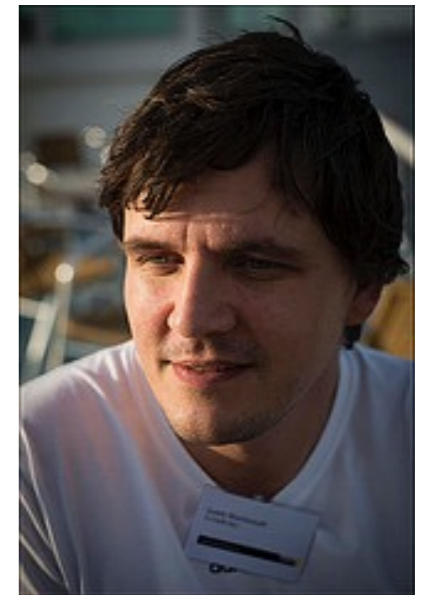
Pierre  
Neron<sup>1</sup>



Andrew  
Tolmach<sup>2</sup>



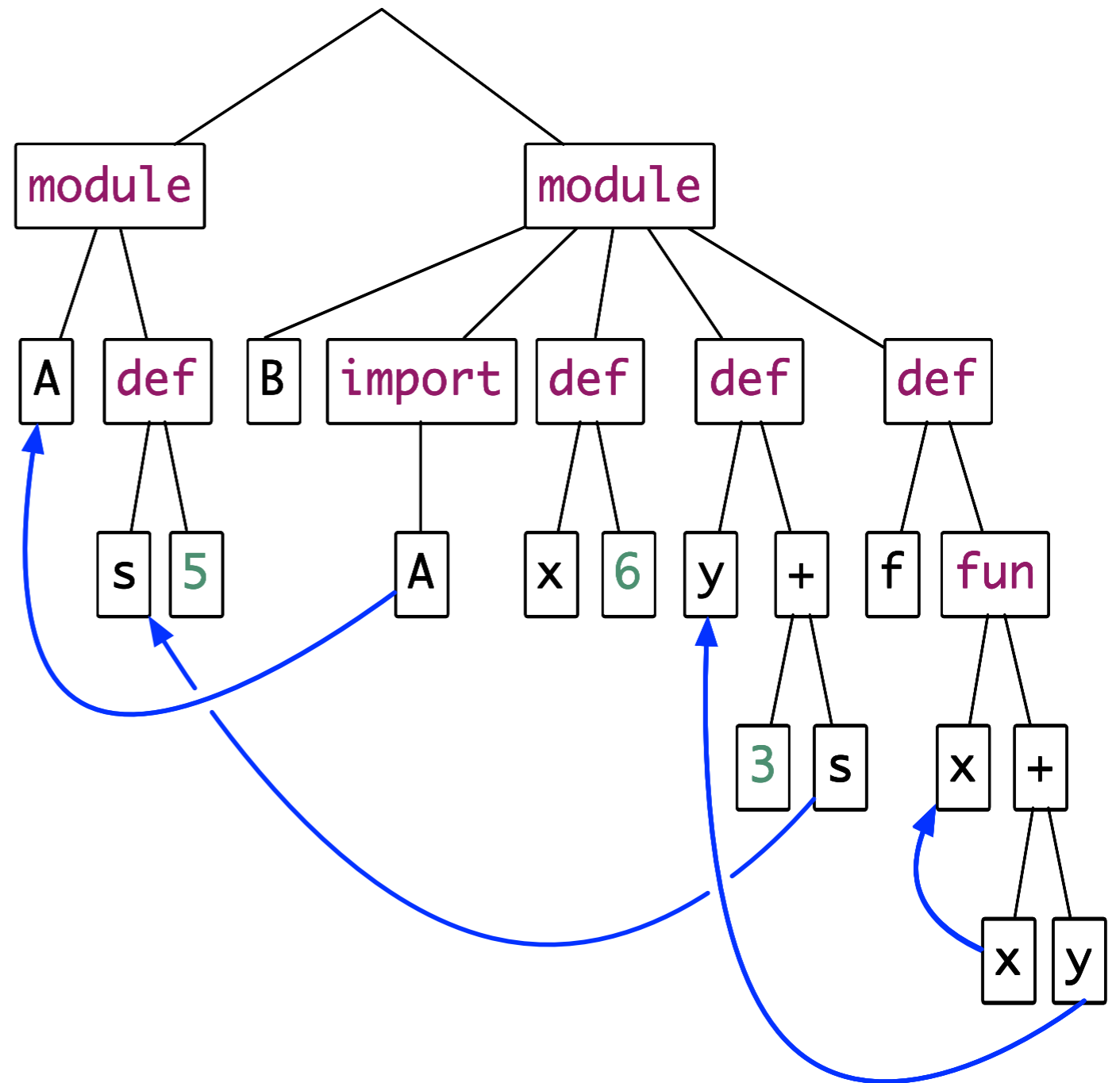
Eelco  
Visser<sup>1</sup>



Guido<sup>1</sup>  
Wachsmuth

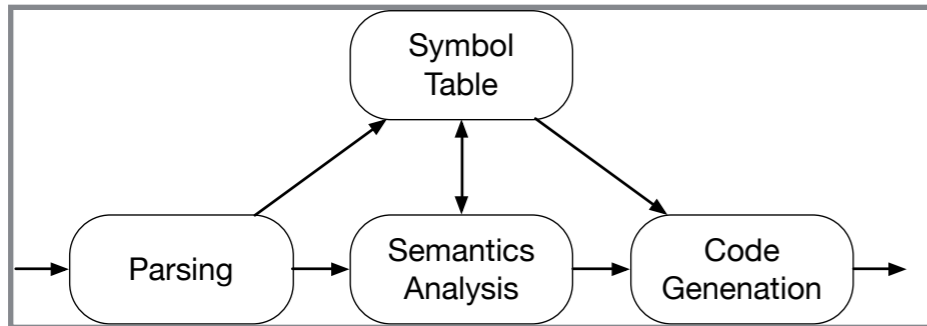
# Language = Set of Graphs

```
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

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

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

Semantics

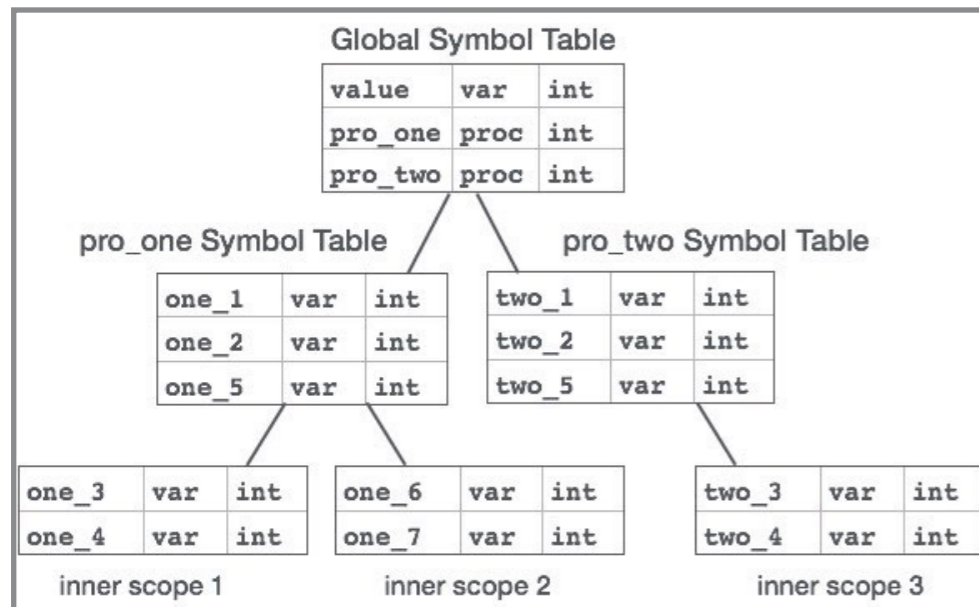
```

public class A {
    static int x;

    int plus(int y) {
        return y + x;
    }
}
  
```

IDE

... with rules encoded in many different ad-hoc ways



$$x : \text{int}, \Gamma$$

$$\text{lookup}(x_i)$$

$$[3/x].\sigma$$

**No standard approach, no re-use**

# Contrast with Syntax

*A unique definition*

*A standard formalism*

*Supports*

```
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
```

**Context-Free  
Grammars**

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]
  - Caml [Pottier06]
  - NaBL [Konat+12]
- Generate code to do resolution and record results

# The NaBL Name Binding Language

Java - metaborg-lmr/trans/name-binding.nab - Eclipse - /Users/eelcovisser/03-Research/workspace-dyl

Expressions.sdf name-binding.nab

```
28 binding rules // Variables
29
30 ArgDecl(name, ty) :
31   defines Variable name of type t
32   where ty has type t
33
34 DefBind(name, e) :
35   defines Variable name of type t
36   where e has type t
37
38 DefBindTyped(name, ty, e) :
39   defines Variable name of type t
40   where ty has type t
41
42 VarRef(name) :
43   refers to Variable name
44
45 FldAccess(ref, name) :
46   refers to Variable name in Record r
47   where ref has type TRec(r)
48
49 New(ref, bnds) :
50   scopes This
51   implicitly defines This This() of type t
52   where ref has type t
53
54 FldBind(name, e) :
55   refers to Variable name in Record r
56   where This() resolves to This this
57   and this has type TRec(r)
58
```

amb01.lmr test01.lmr record01.lmr

```
1 program record01
2
3 record Point { x : Int, y : Int}
4
5 record ColorPoint extends Point { c : Int }
6
7 record Line { s : Point, e: Point}
8
9 def foo : Point = 1
10
11 def p = new Point { x = 1, y = 2 }
12 def q = p.x
13
14 def l = new Line {}
15 def k = l.e.x
```

record01.aterm

```
21 , Def(
22   DefBind(
23     "p"
24     , New(
25       TypeRef("Point")
26       , [FldBind("x", Int("1")), FldBind("y", Int("2"))])
27     )
28   )
29 )
30 , Def(DefBind("q", FldAccess(VarRef("p"), "x")))
31 , Def(DefBind("l", New(TypeRef("Line"), [])))
32 , Def(
33   DefBind("k", FldAccess(FldAccess(VarRef("l"), "e"), "x"))
34 )
```

Writable Smart Insert 54 : 21 Analyzing files (legacy)

# Multi-Purpose Name Binding Rules

module names

namespaces Variable

binding rules

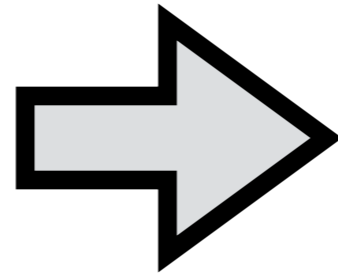
Var(x) :  
refers to Variable x

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]
  - Caml [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

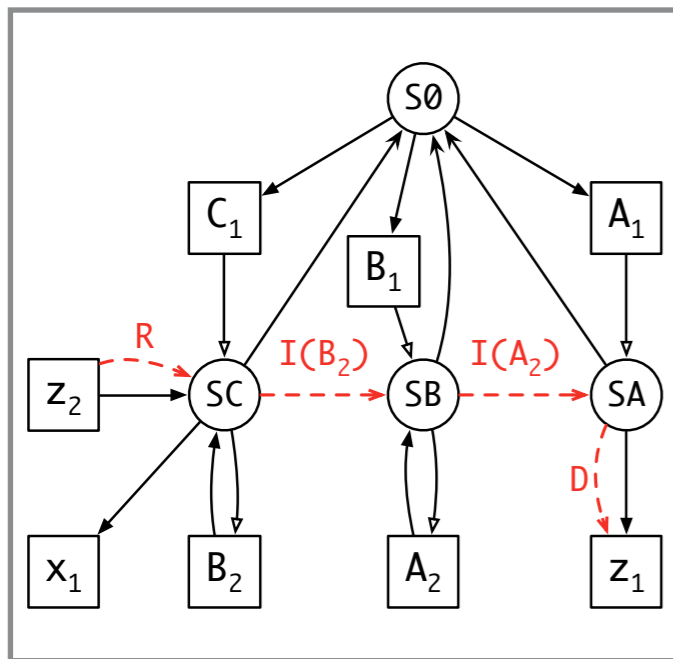
# A Theory of Name Resolution

For *statically lexically scoped* languages

*A unique  
representation*

*A standard  
formalism*

*Supports*



**Scope  
Graphs**

Resolution

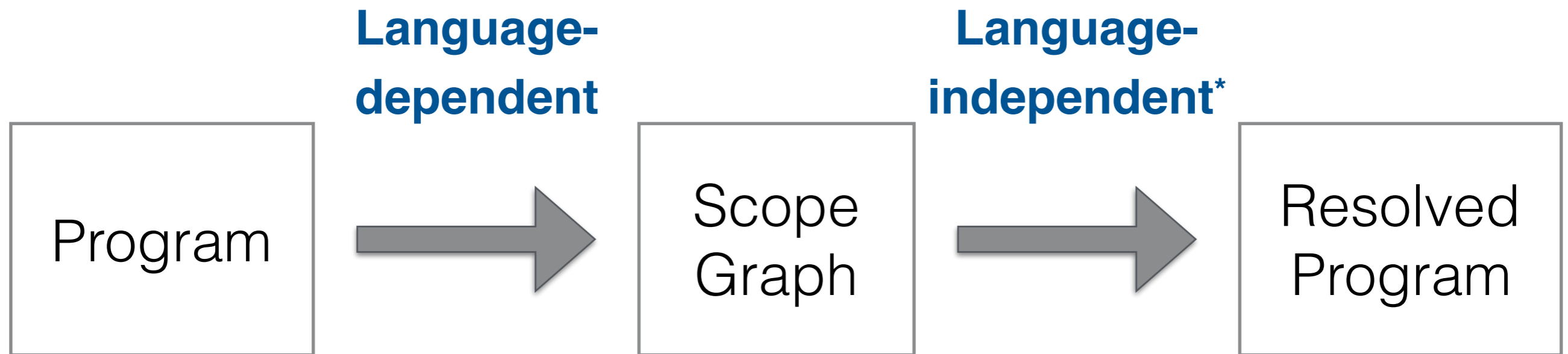
$\alpha$ -equivalence

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**

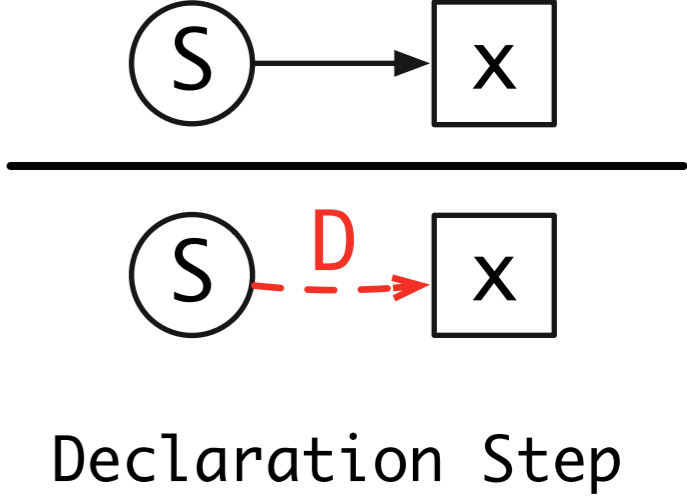
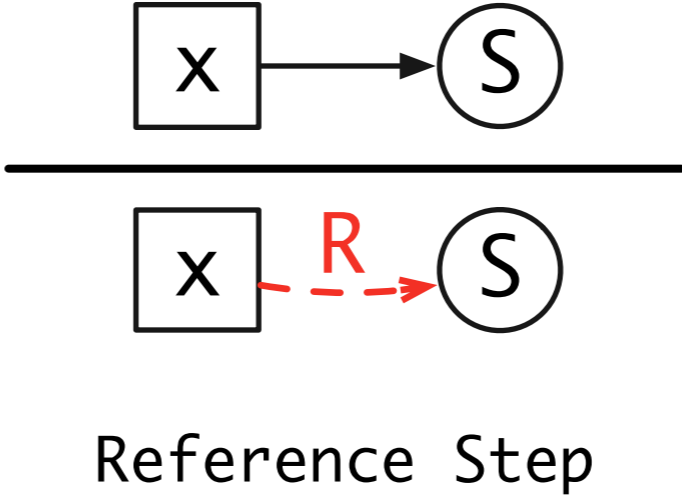
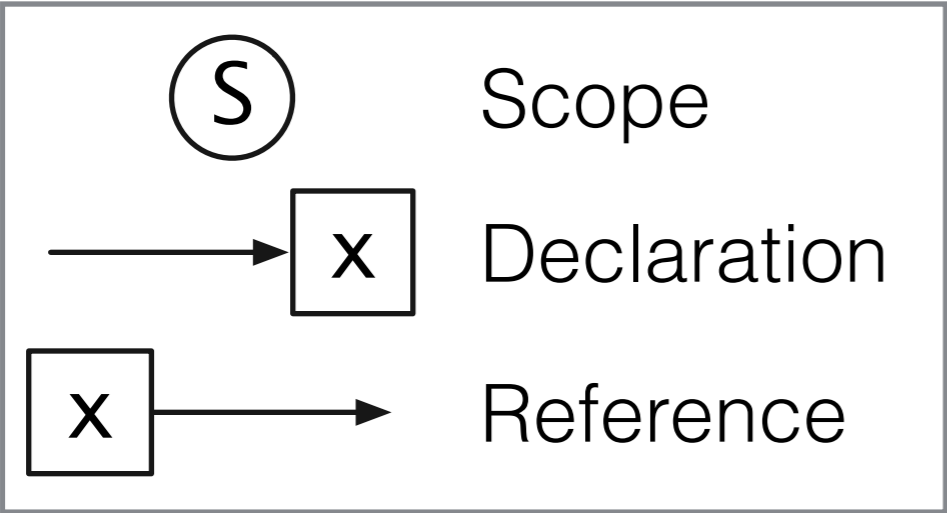
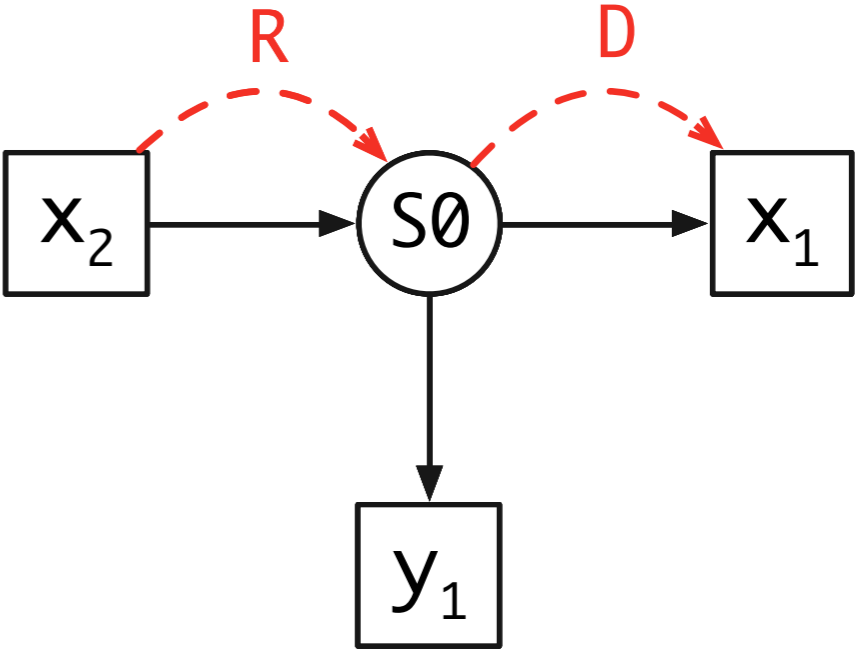
---

# Simple Scopes

```

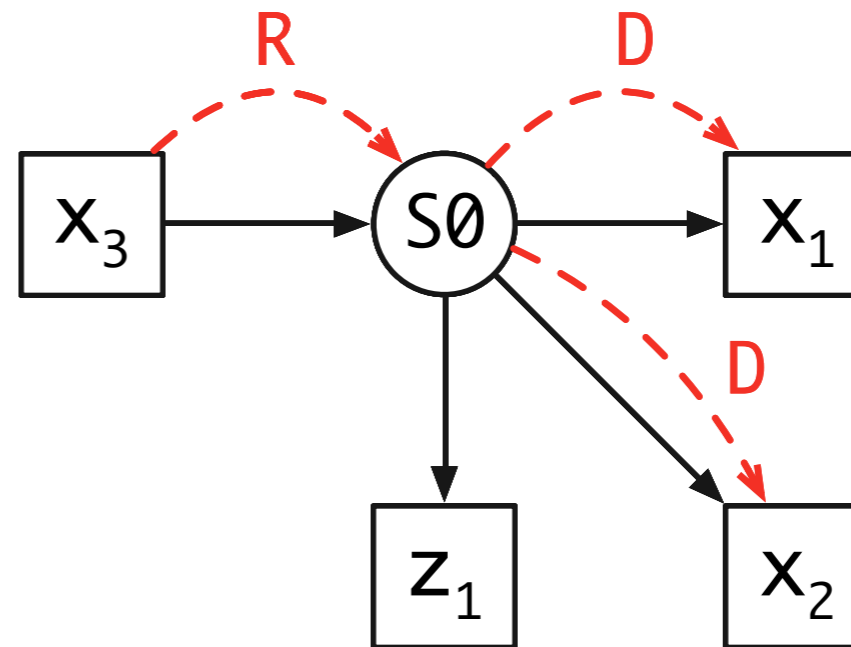
def y1 = x2 + 1
def x1 = 5
    
```

S<sub>0</sub>



# Ambiguous Resolutions

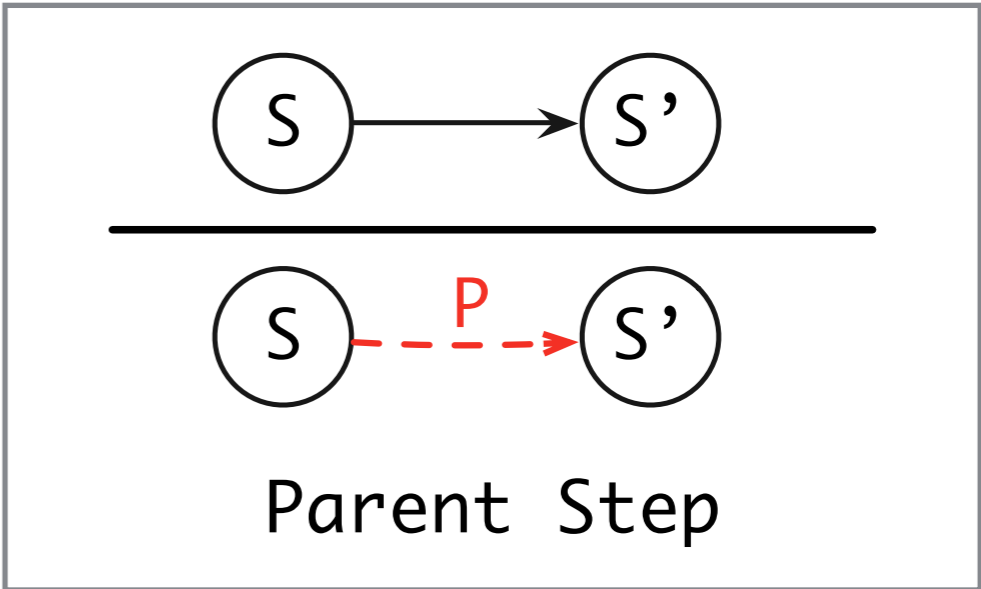
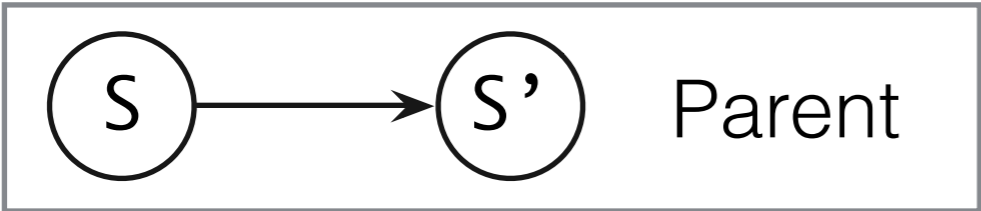
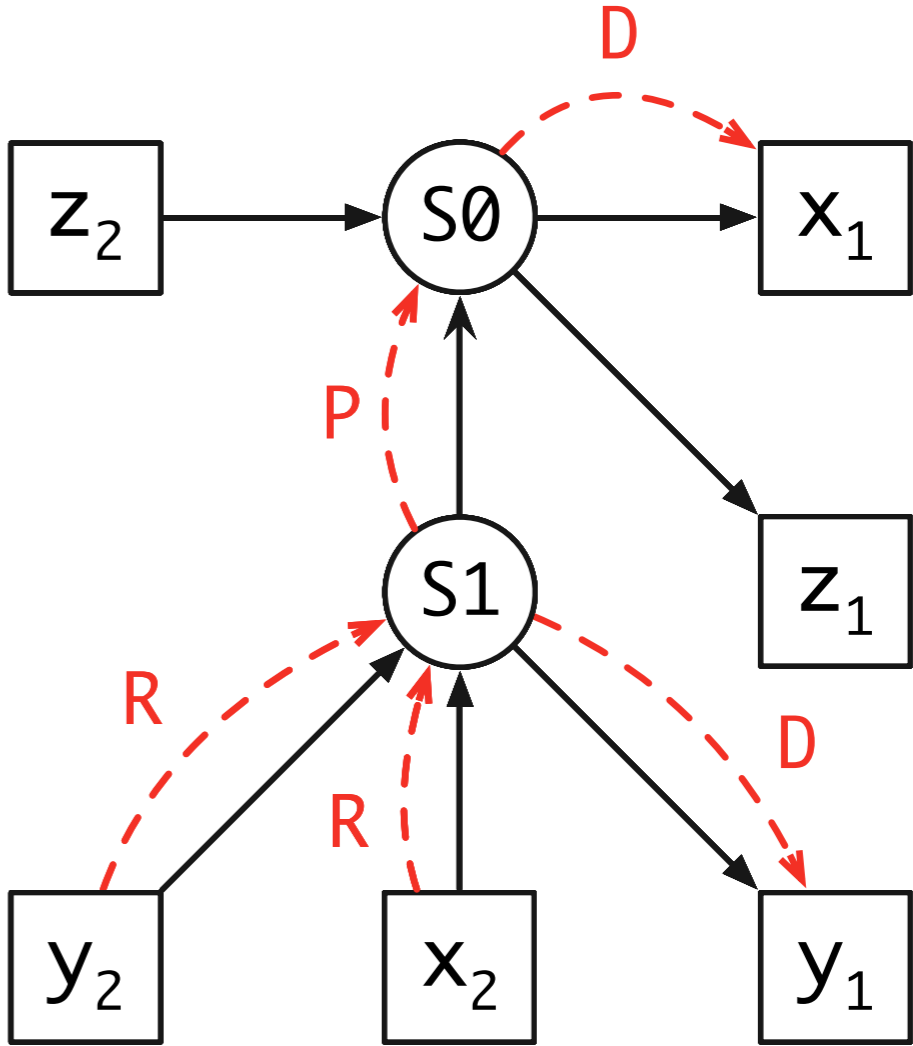
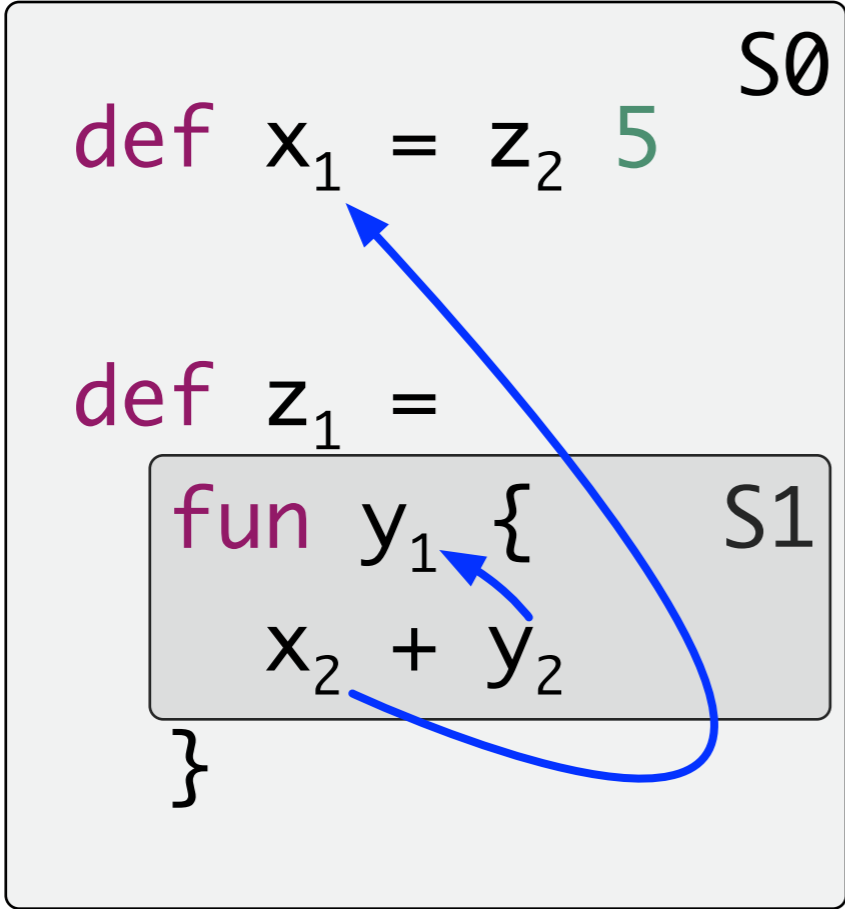
```
def  $x_1 = 5$   $S_0$   
def  $x_2 = 3$   
def  $z_1 = x_3 + 1$ 
```



```
match t with  
| A  $x$  | B  $x$  => ...
```



# Lexical Scoping



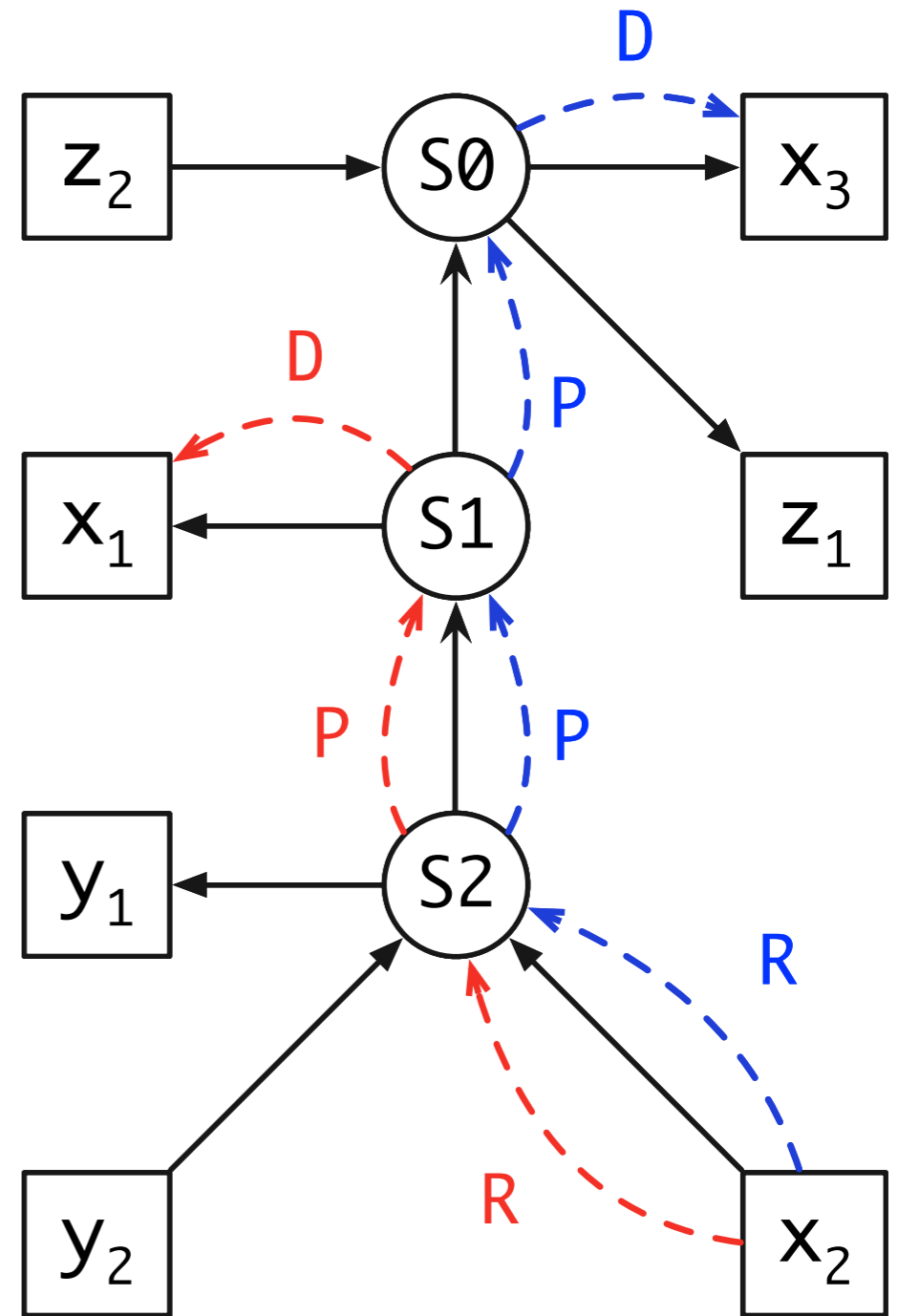
Well formed path: **R.P\*.D**

# Shadowing

```

def x3 = z2 5 7 S0

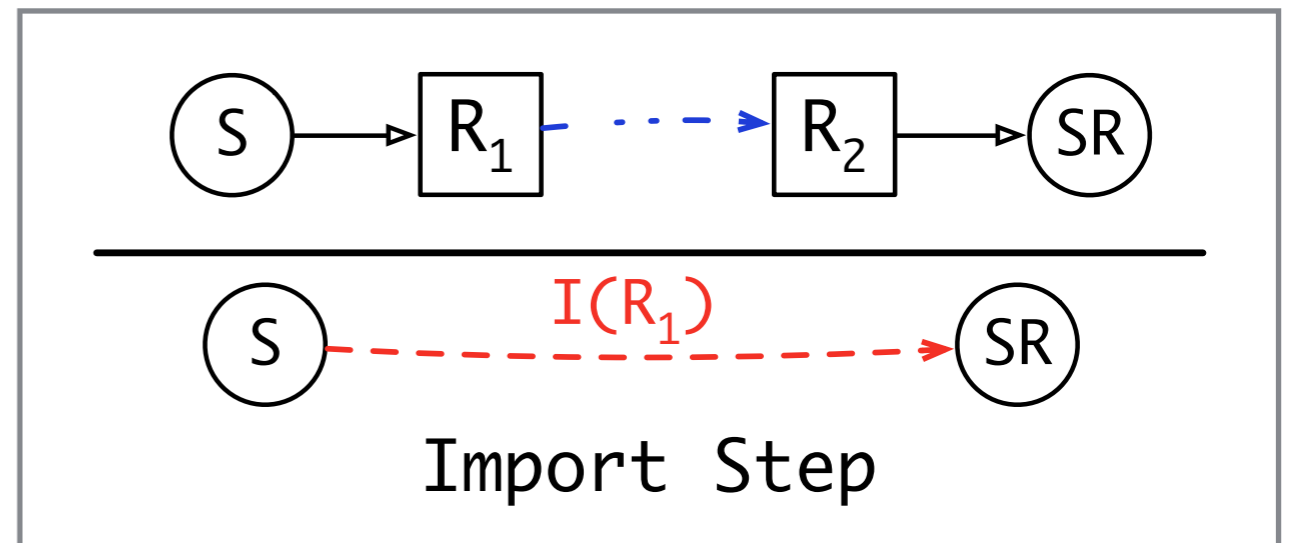
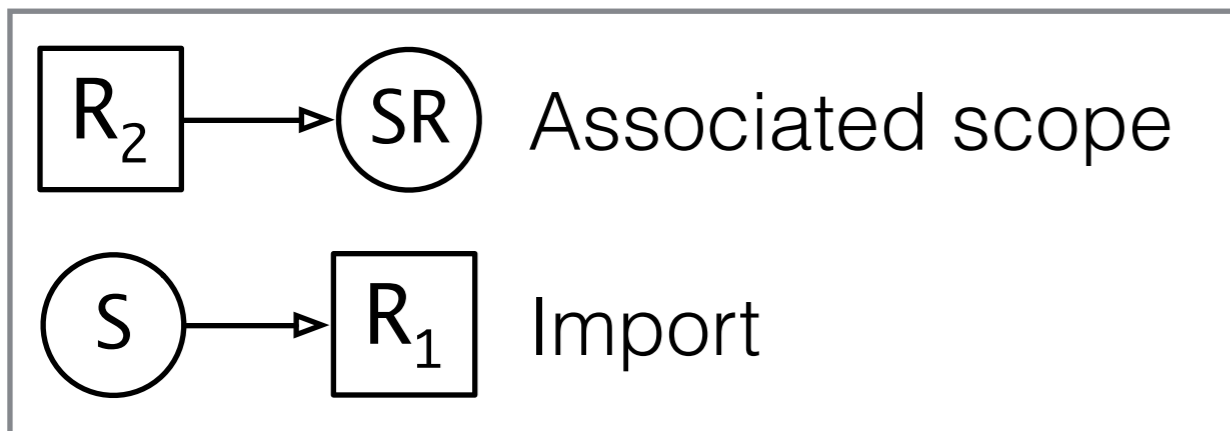
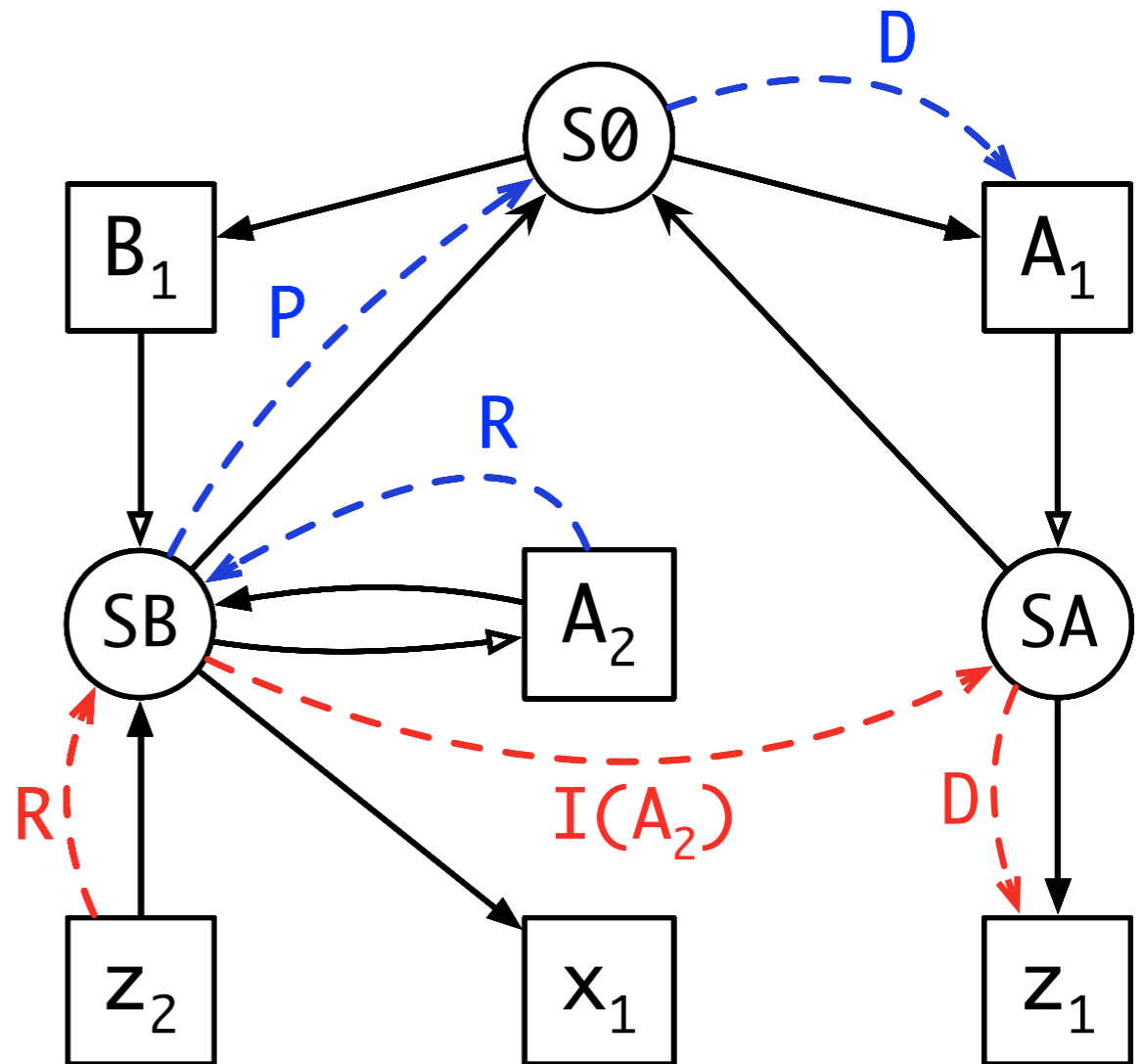
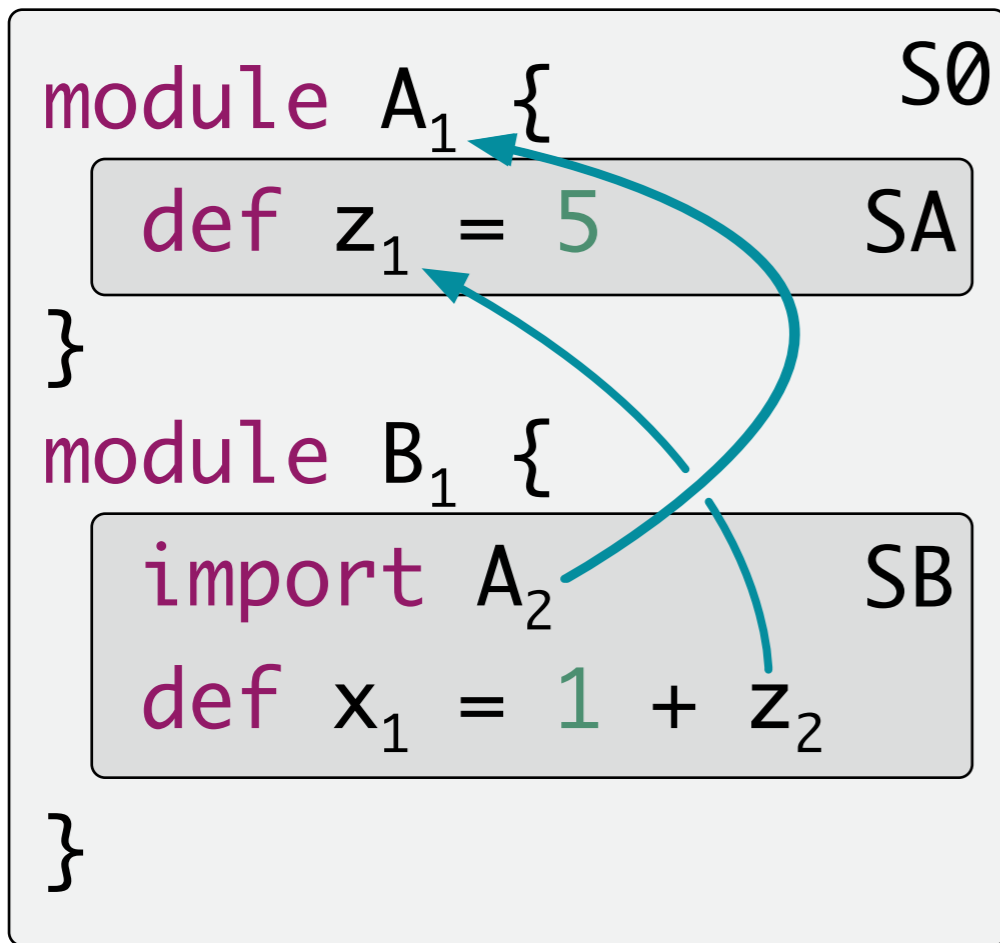
def z1 =
  fun x1 { S1
    fun y1 { S2
      x2 + y2
    }
  }
  }
  
```



$$\begin{array}{c}
 \hline
 D < P.p \\
 \\
 p < p' \\
 \hline
 s.p < s.p'
 \end{array}$$

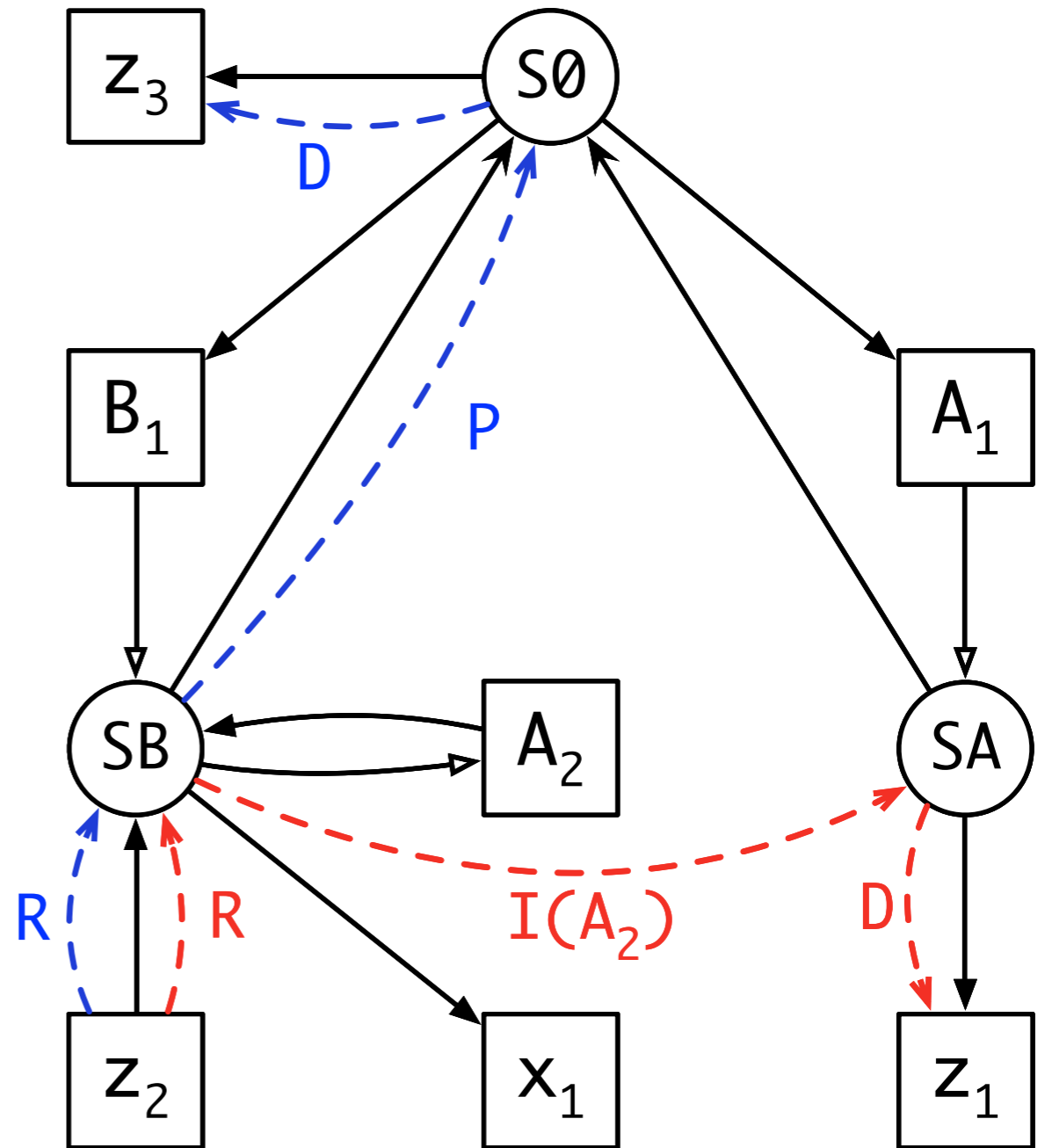
$$R.P.D < R.P.P.D$$

# Imports



# Imports shadow Parents

```
def z3 = 2 S0  
  
module A1 {  
  def z1 = 5 SA  
}  
  
module B1 {  
  import A2 SB  
  def x1 = 1 + z2  
}
```

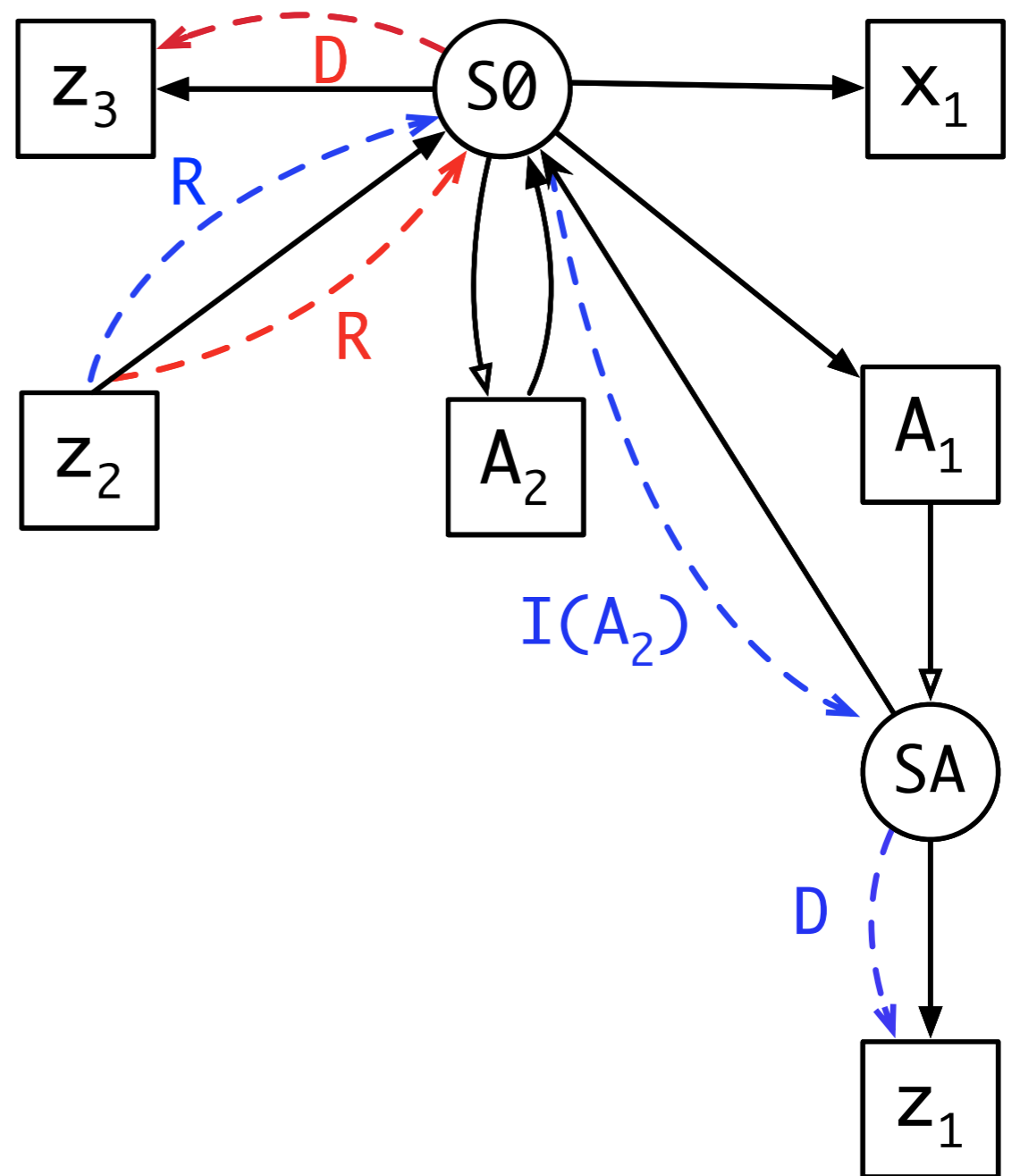
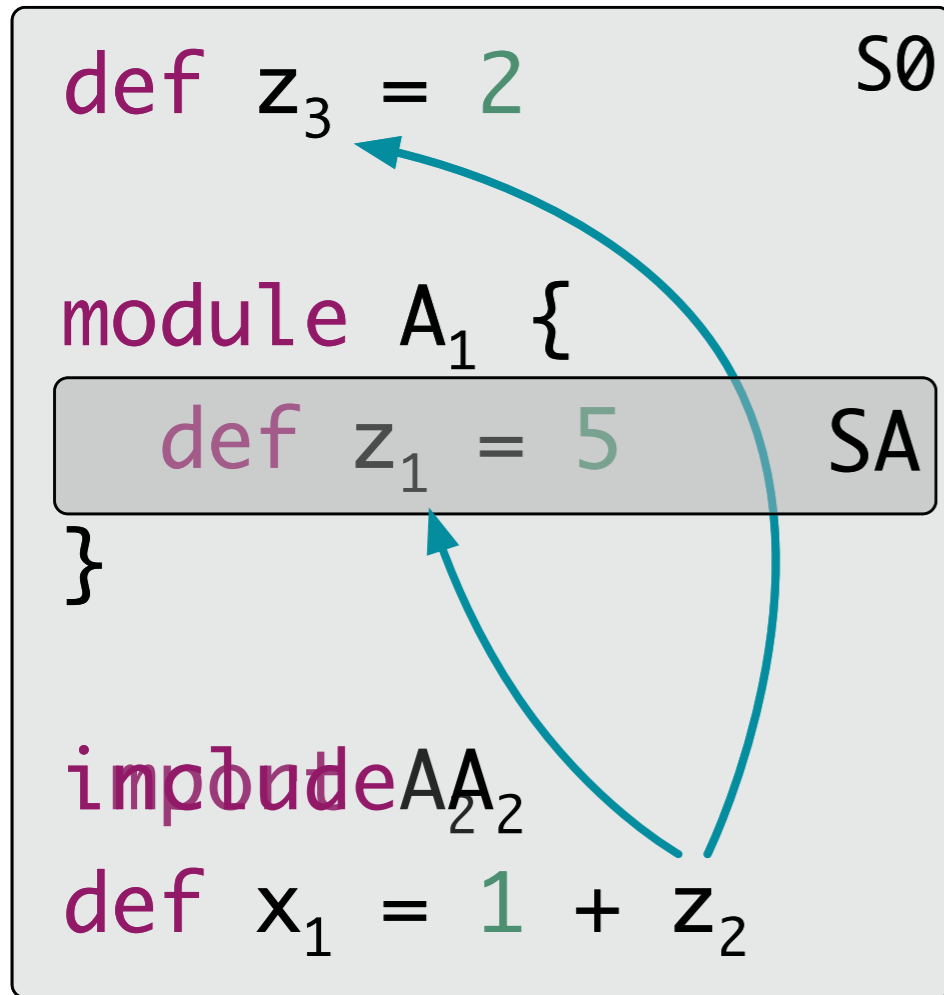


---

$I(\_).p' < P.p$

$\Rightarrow R.I(A_2).D < R.P.D$

# Imports vs. Includes



~~$D < I(-).p$~~



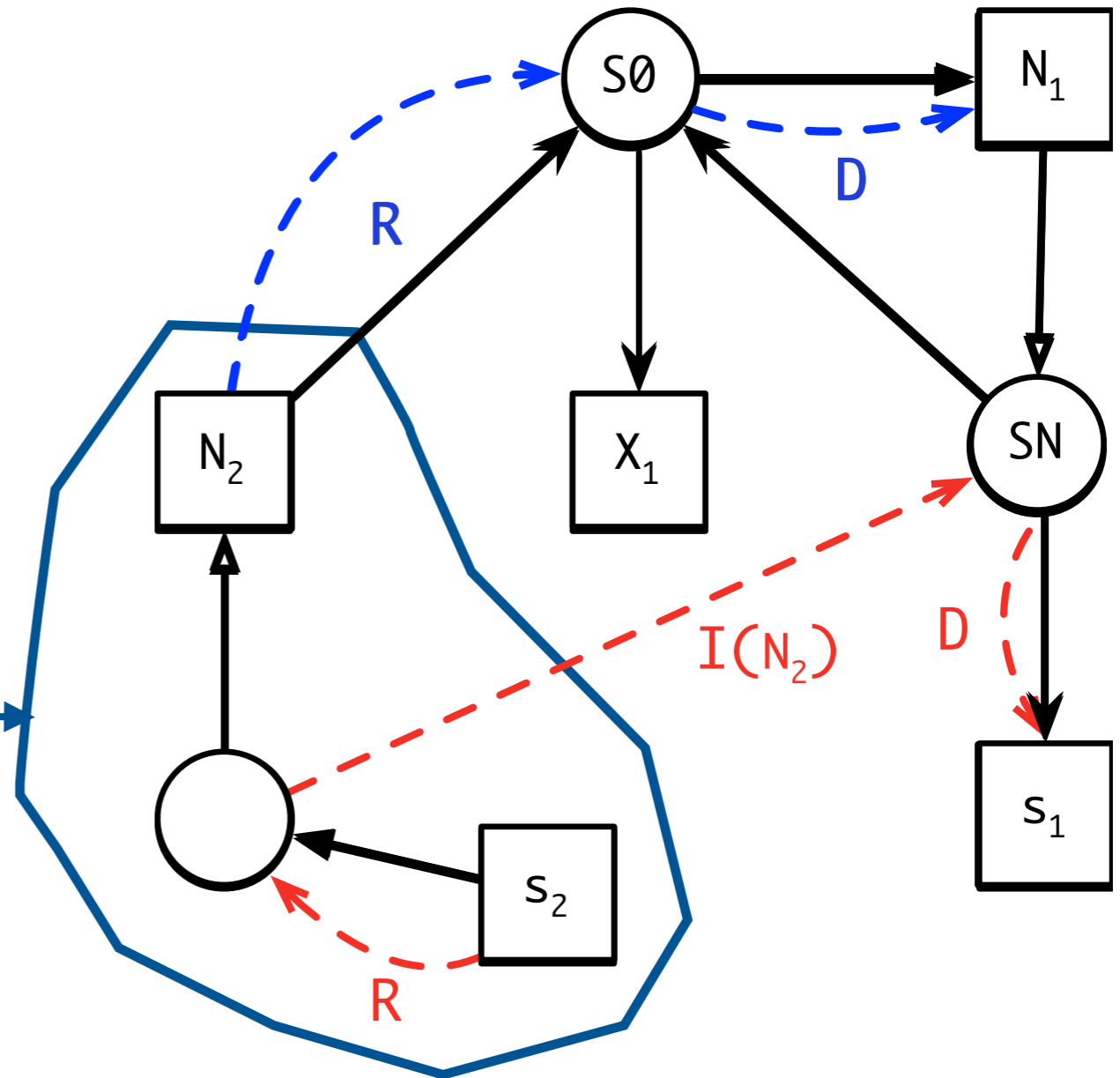
$R.D < R.I(A_2).D$

# Qualified Names

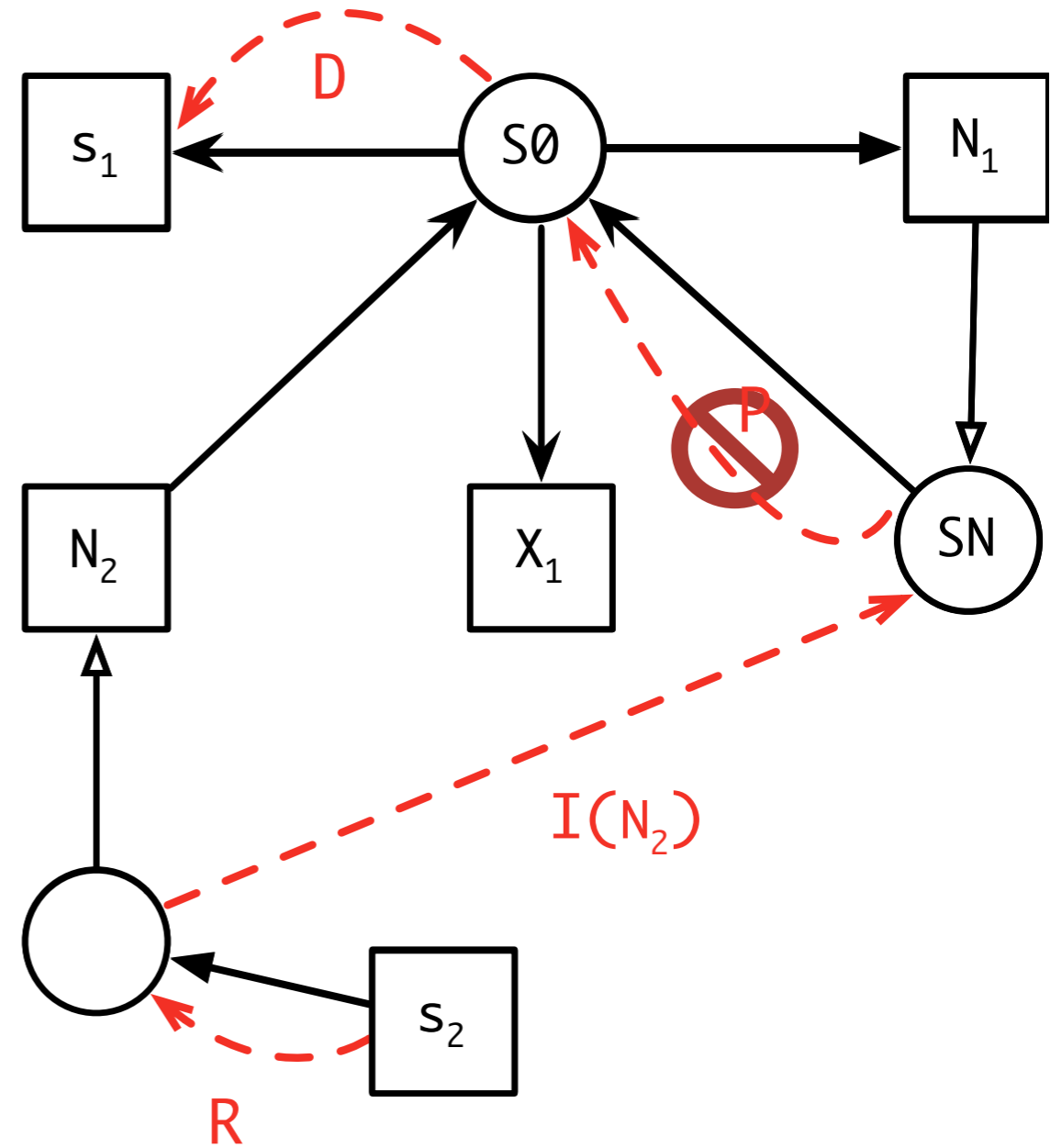
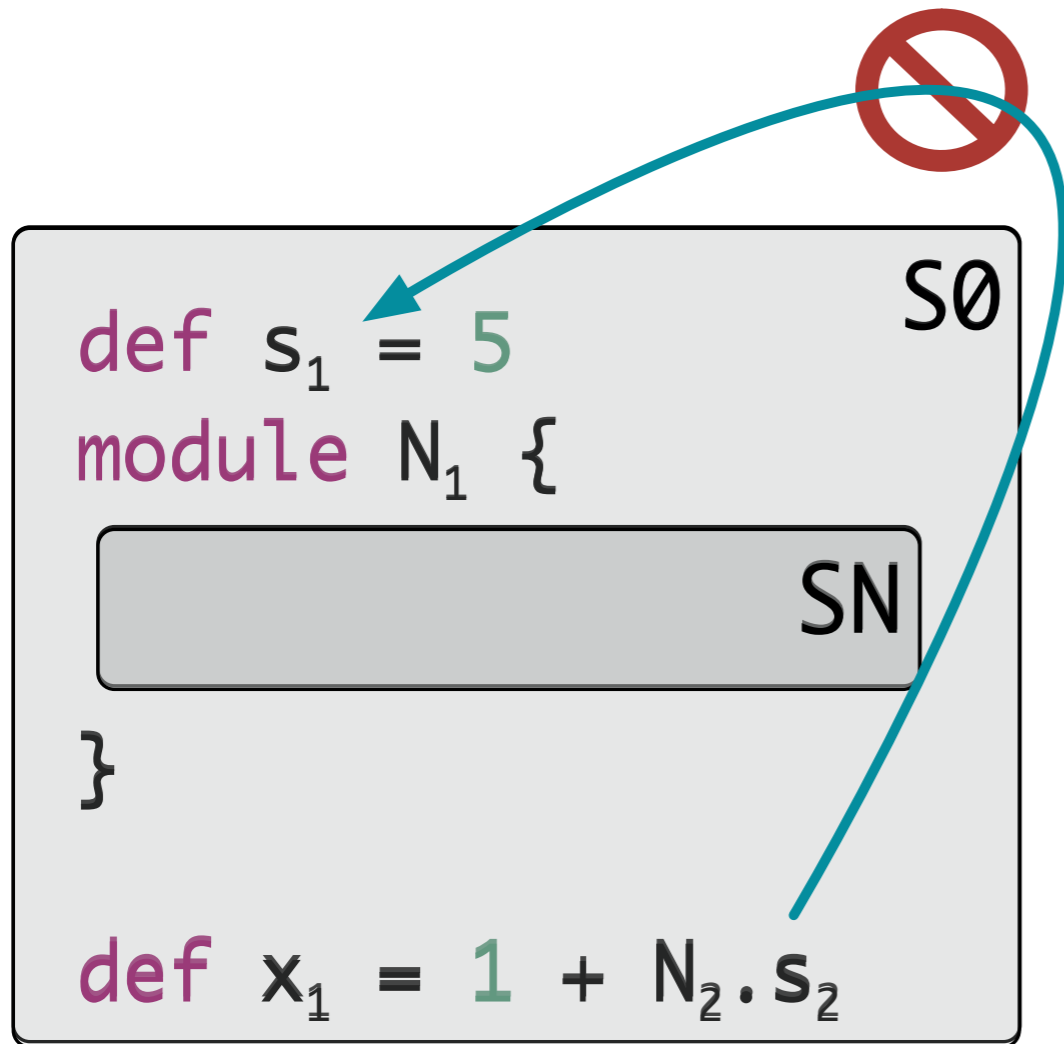
```
module N1 {  
  def s1 = 5  
}
```

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

S<sub>0</sub>

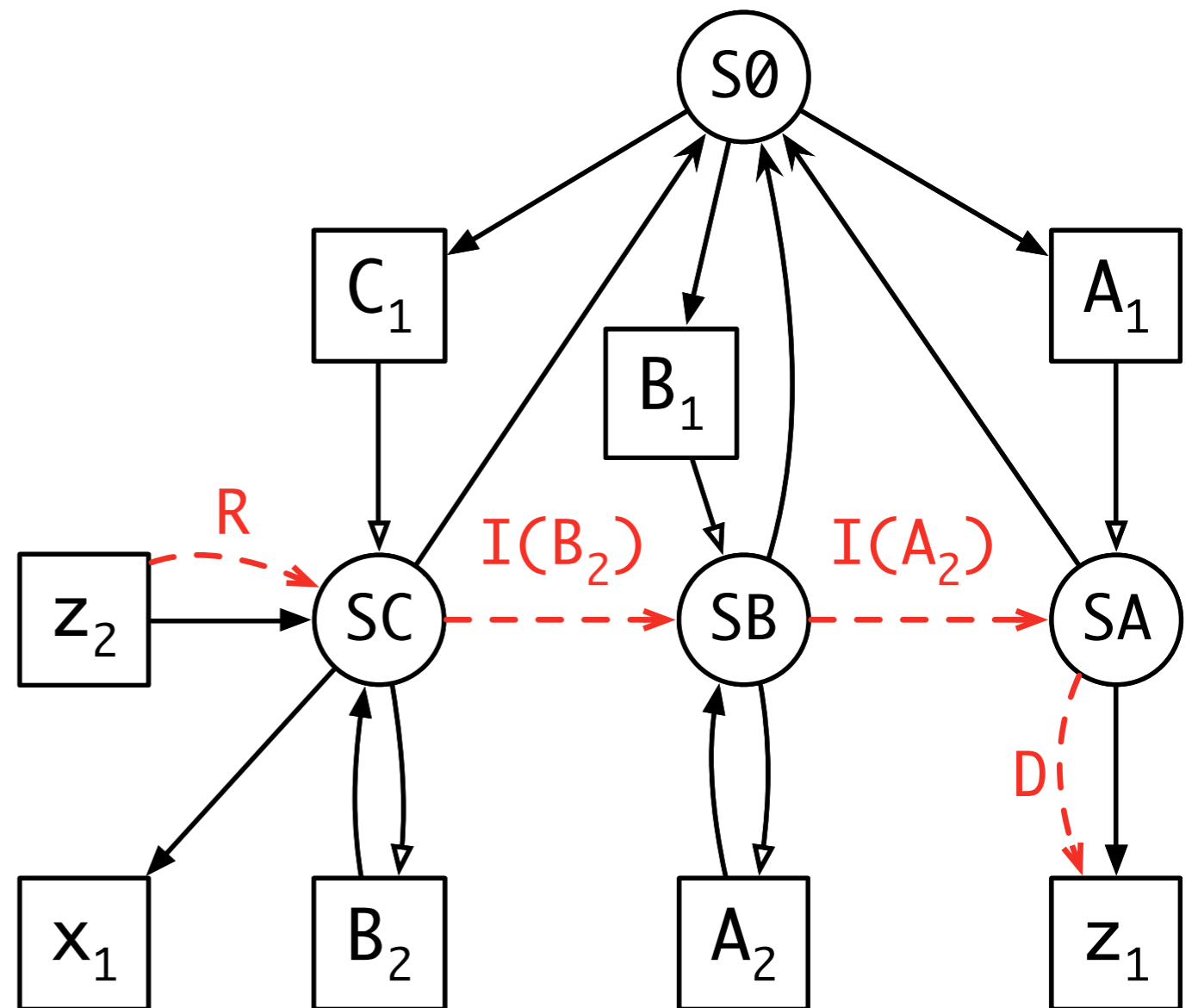
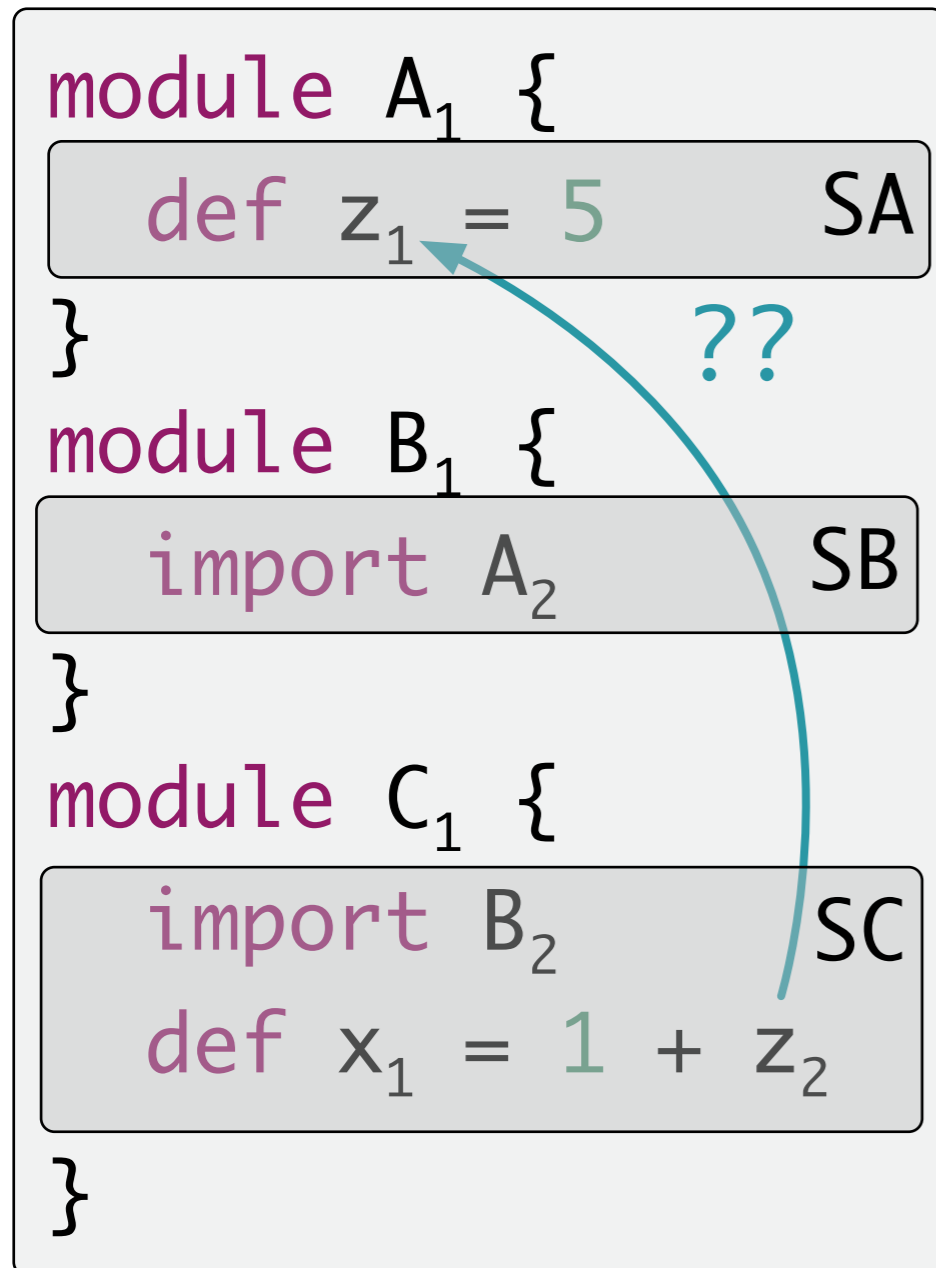


# Import Parents



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

# Transitive vs. Non-Transitive



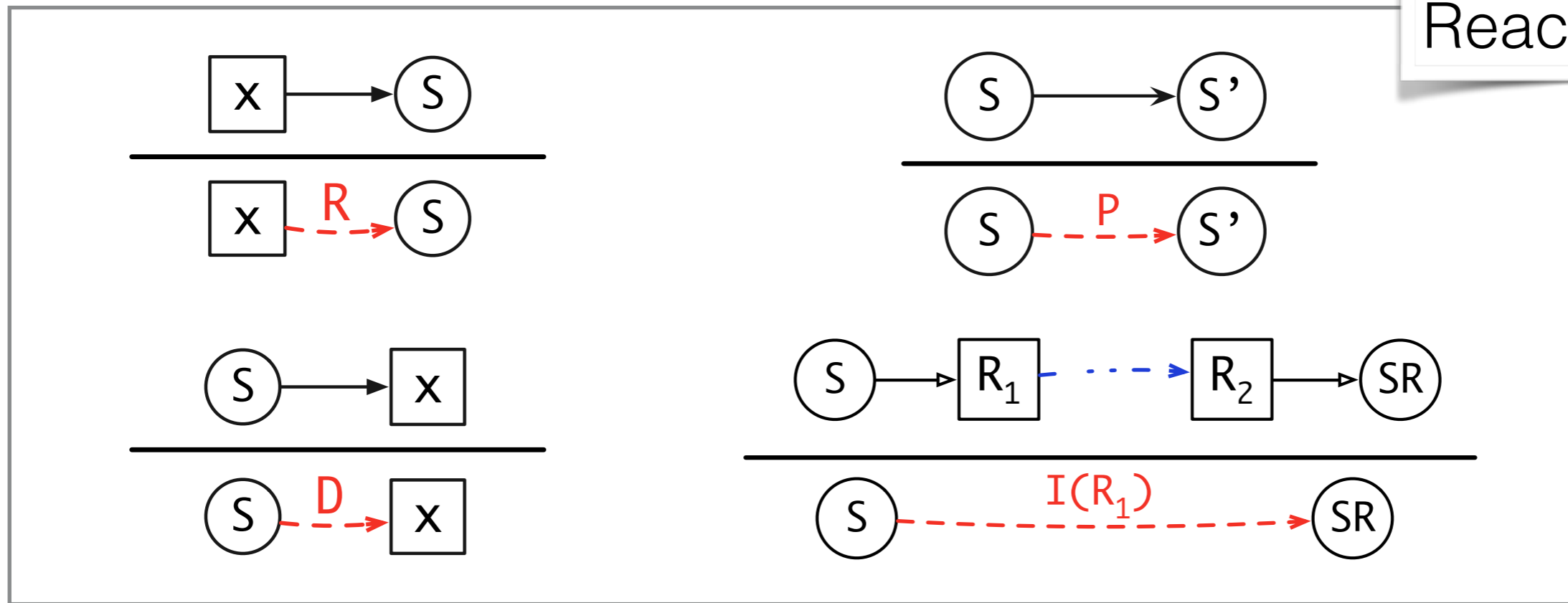
With transitive imports, a well formed path is  $R.P^*.I(\_)*.D$

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



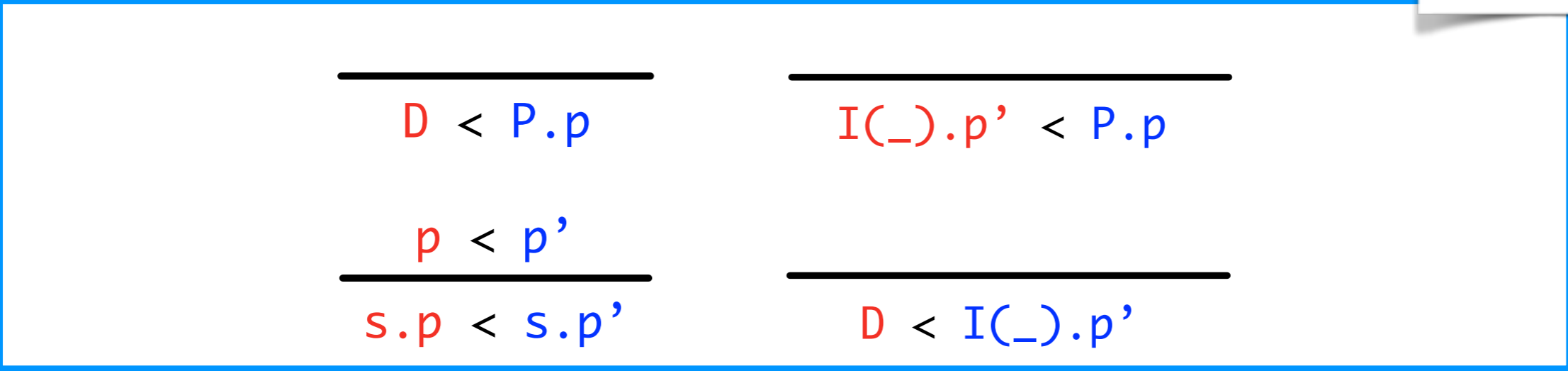
# A Calculus for Name Resolution

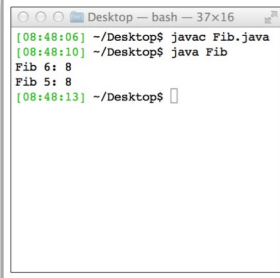
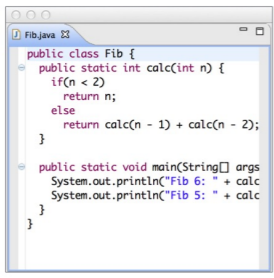
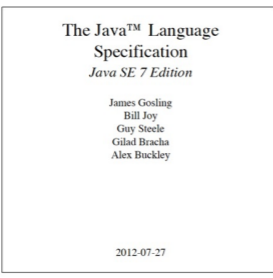

Reachability



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

Visibility

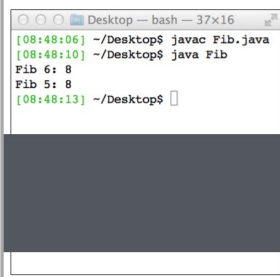
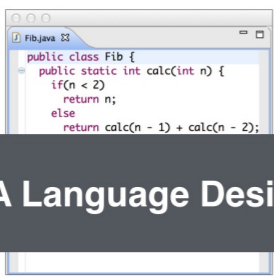
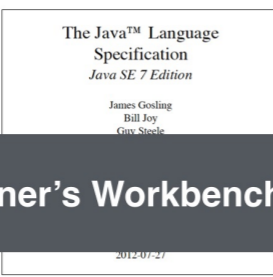
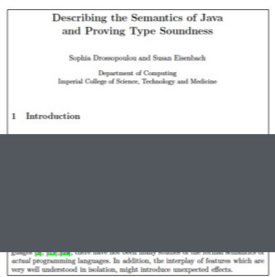


			
parser type checker code generator interpreter	parser error recovery syntax highlighting outline code completion navigation type checker debugger	syntax definition static semantics dynamic semantics	abstract syntax type system operational semantics type soundness proof

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

$$\frac{x \rightarrow S}{x \overset{R}{\dashrightarrow} S}$$

$$\frac{S \rightarrow x}{S \overset{D}{\dashrightarrow} x}$$

$$\frac{S \rightarrow S'}{S \overset{P}{\dashrightarrow} S'}$$

$$\frac{S \rightarrow R_1 \quad \dots \quad R_2 \rightarrow SR}{S \overset{I(R_2)}{\dashrightarrow} SR}$$

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

Visibility

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