

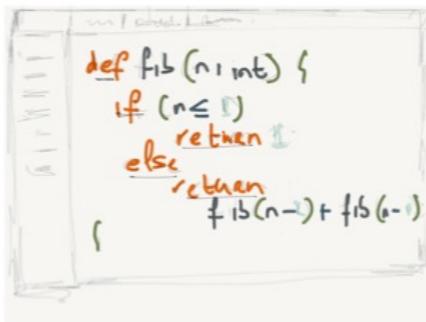
Declare Your Language

Eelco Visser

Dynamic Languages Symposium
October 27, 2015



Delft University of Technology



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$ 

```

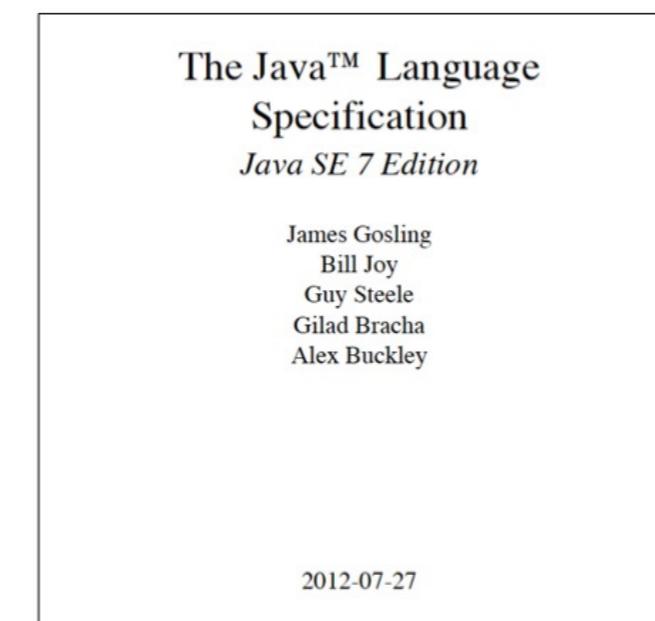
Fib.java

```

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));
    }
}

```



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 [Smalltalk](#), is a simplification of the signatures extension for C++ [\[1\]](#) 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 [\[2\]](#) 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 [\[3\]](#), [\[4\]](#), [\[5\]](#), 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$ 

```

```

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 [Smalltalk](#), is a simplification of the signatures extension for C++ [\[1\]](#) 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 [\[2\]](#) 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 [\[3\]](#), [\[4\]](#), [\[5\]](#), 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.

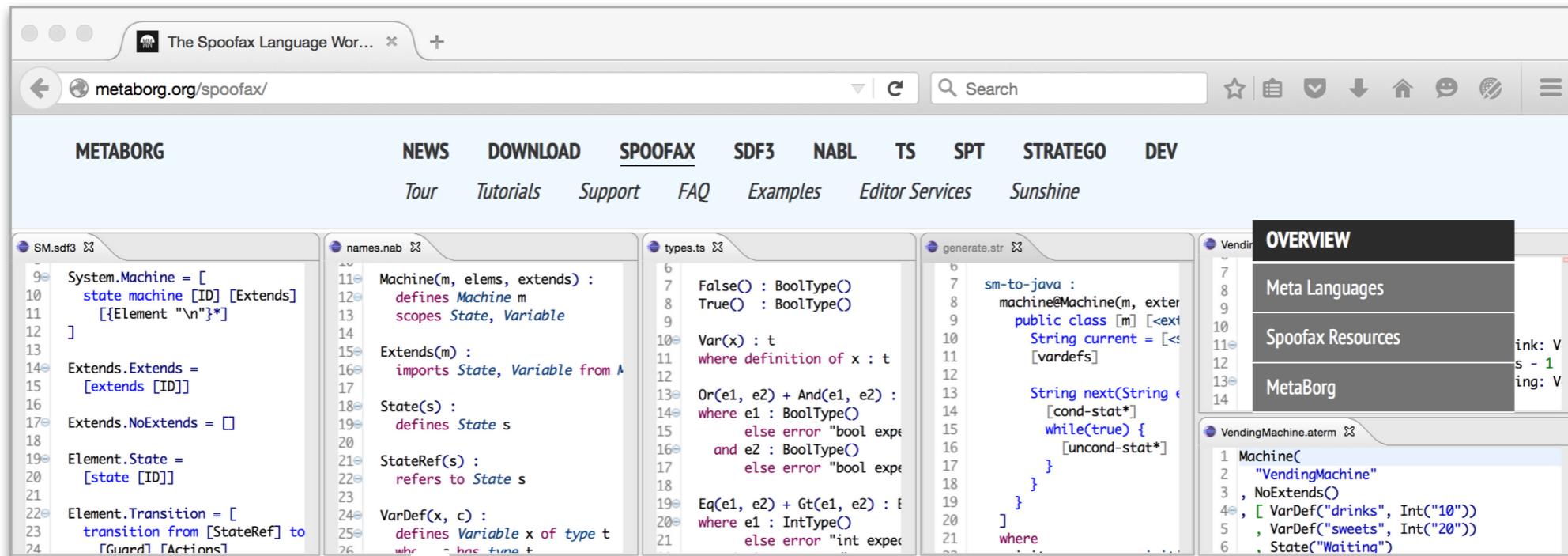
parser
type checker
code generator
interpreter

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 the Spoofax Language Workbench interface. At the top, there's a navigation bar with links for METABORG, NEWS, DOWNLOAD, SPOOFAX (which is underlined), SDF3, NABL, TS, SPT, STRATEGO, and DEV. Below the navigation bar are sub-links: Tour, Tutorials, Support, FAQ, Examples, Editor Services, and Sunshine. The main area consists of several code editors. From left to right, they contain:

- SM.sdf3: A grammar definition for a system machine.
- names.nab: A name binding file.
- types.ts: A type specification file.
- generate.str: A Stratego transformation file.
- VendingMachine.aterm: An abstract syntax tree for a vending machine.

On the right side, there's an "OVERVIEW" panel with sections for Meta Languages, Spoofax Resources, and MetaBorg. Below the overview panel, there's a small preview of the VendingMachine.aterm file.

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

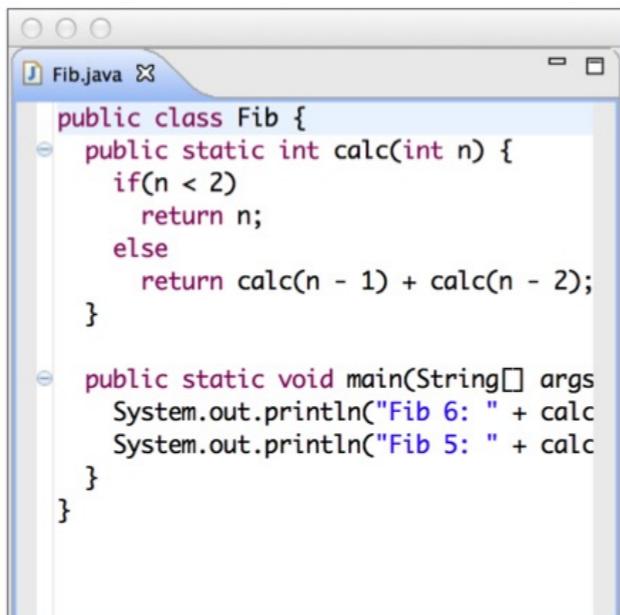
Name
Resolver

Type
Checker

Code
Generator



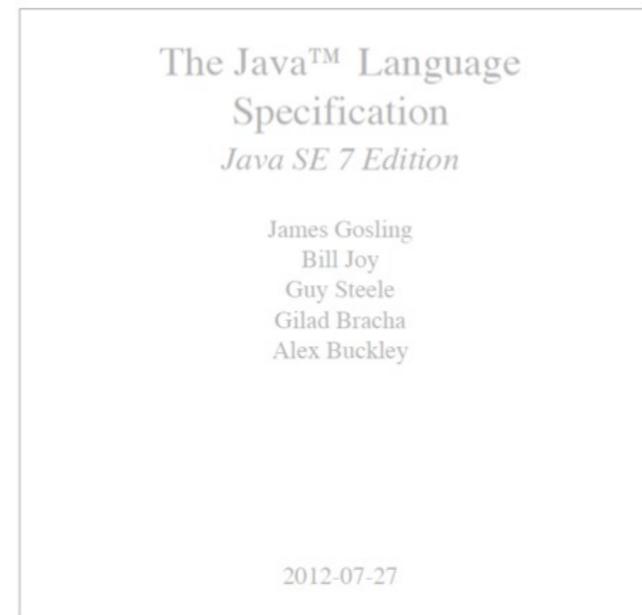
```
Desktop — bash — 37x16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$
```



A screenshot of a Java code editor showing the file `Fib.java`. The code defines a class `Fib` with a static method `calc` that calculates the nth Fibonacci number using recursion. It also contains a `main` method that prints the 6th and 5th Fibonacci numbers.

```
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));
    }
}
```



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$
```

A screenshot of a Java code editor showing a file named Fib.java. The code defines a public class Fib with a static method calc that returns the nth Fibonacci number using a recursive formula.

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

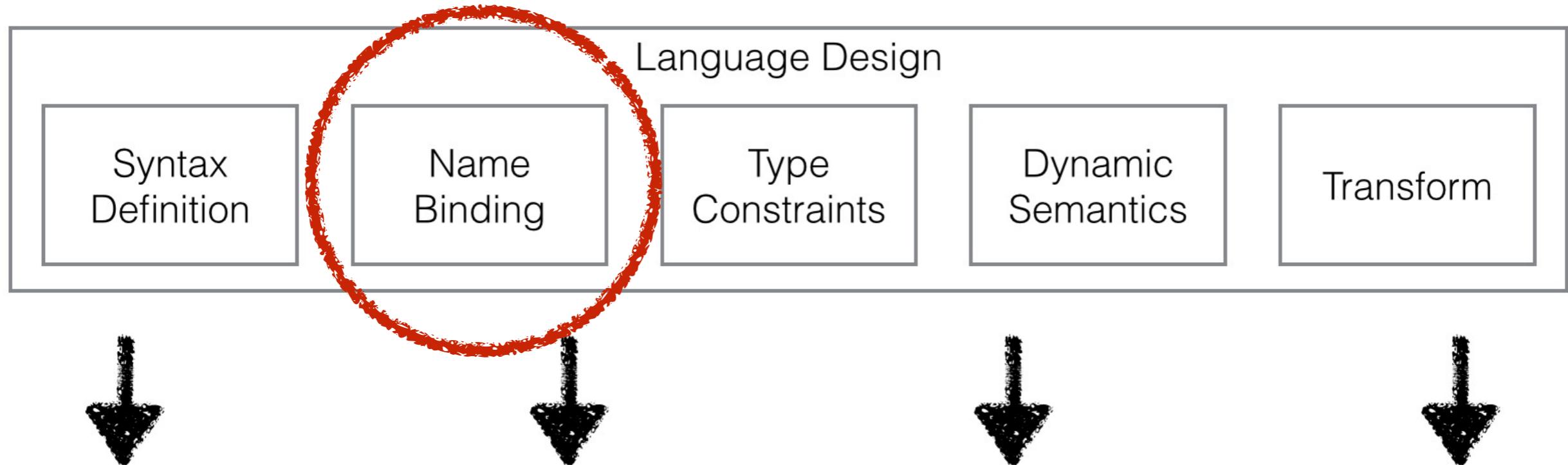
1 Introduction

A Language Designer's Workbench

2012-07-27

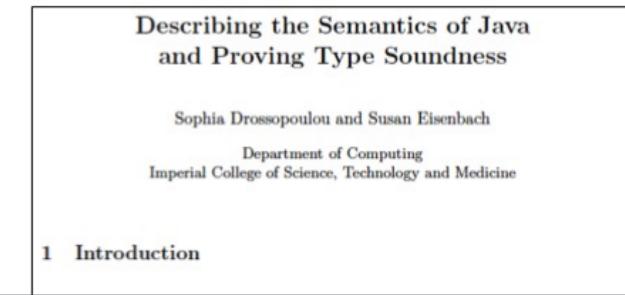
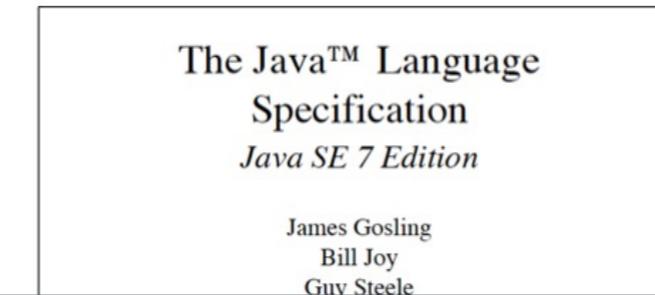
languages [4], [5], [6], there have not been many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.

A Theory of Name Resolution



```
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$
```

```
public class Fib {
    public static int calc(int n) {
        if(n < 2)
            return n;
        else
            return calc(n - 1) + calc(n - 2);
    }
}
```

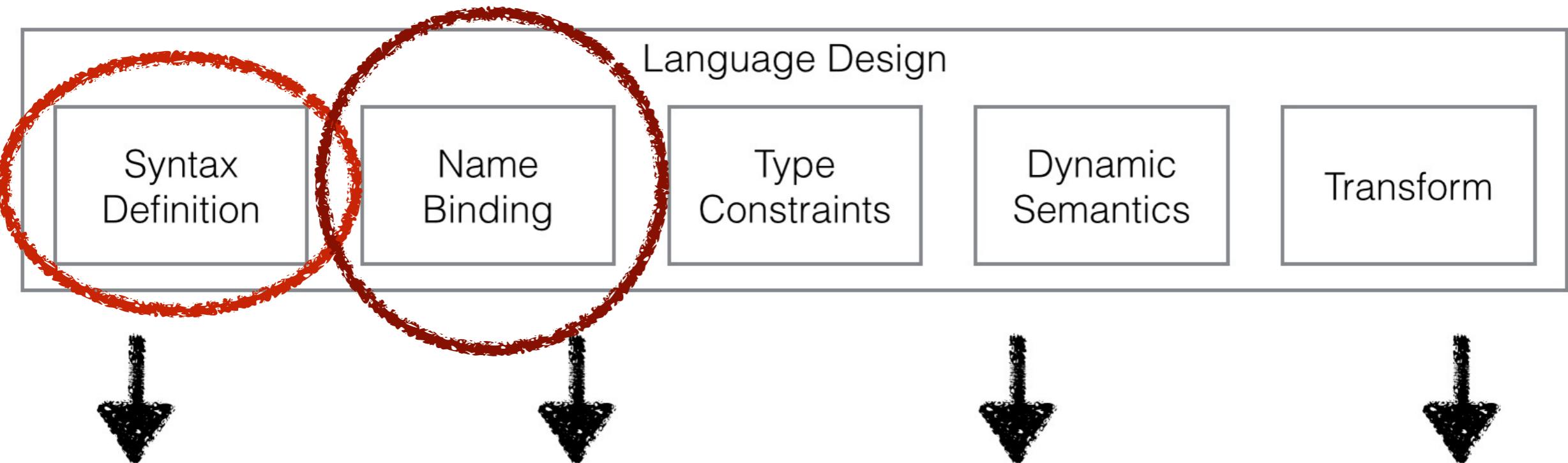


A Language Designer's Workbench

2012-07-27

languages [4], [5], [6], 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.

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 X
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

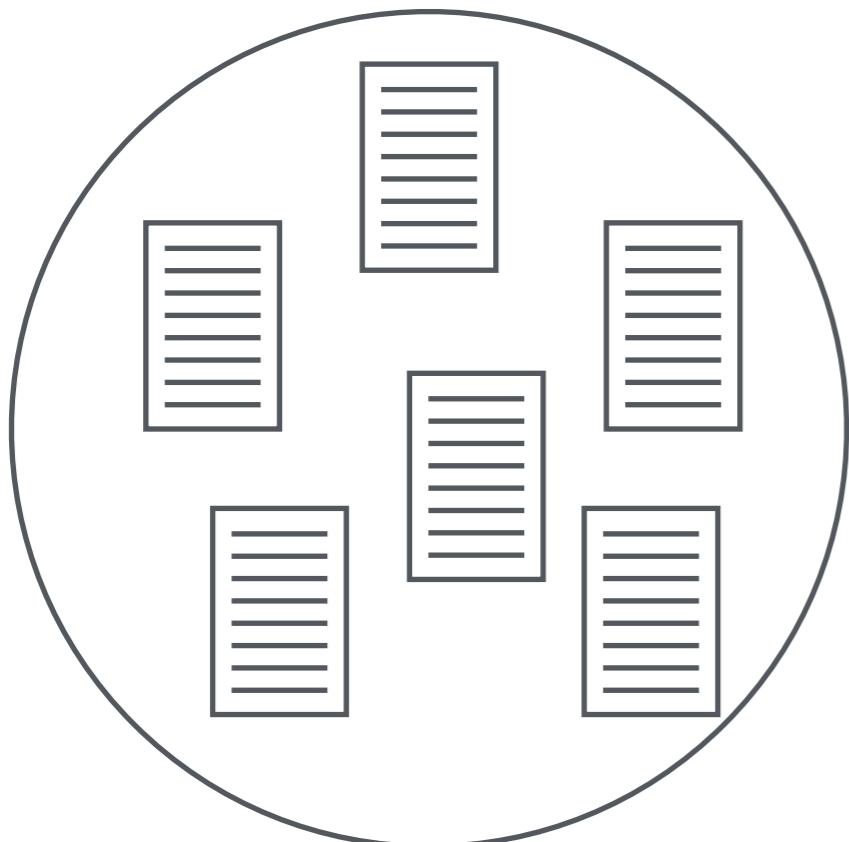
languages [4], [5], [6], there have not been many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.

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

Declare Your Syntax

[Kats, Visser, Wachsmuth; Onward 2010]

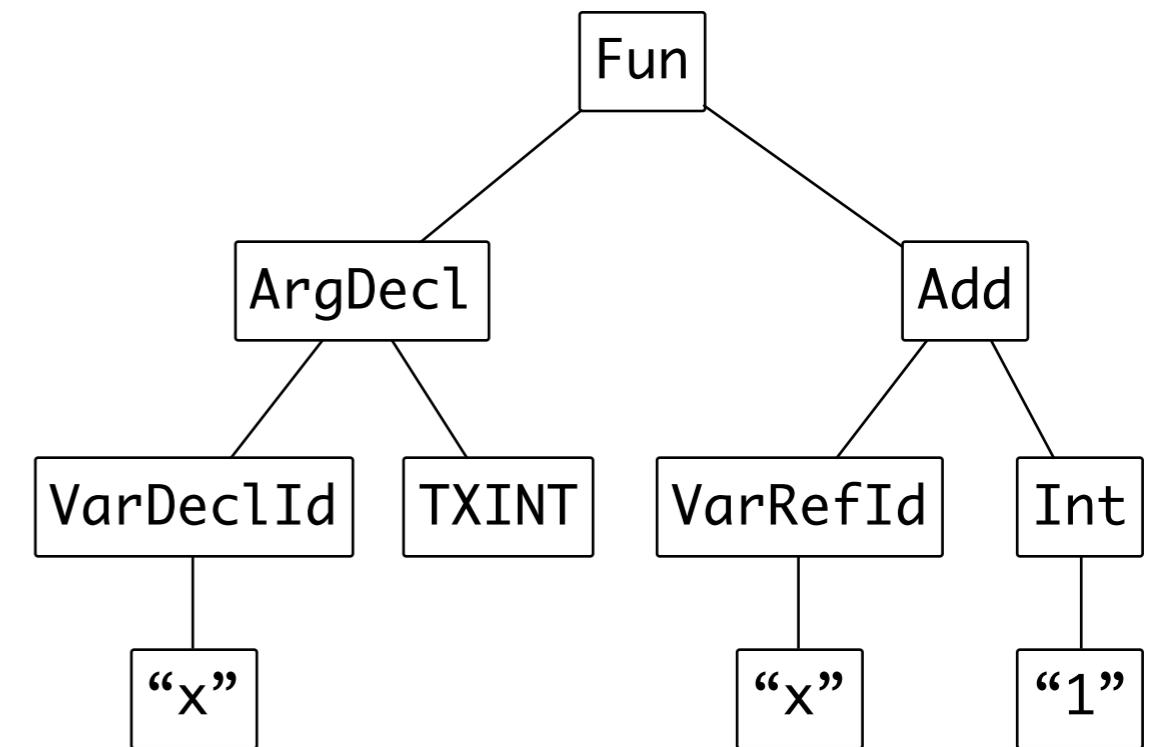
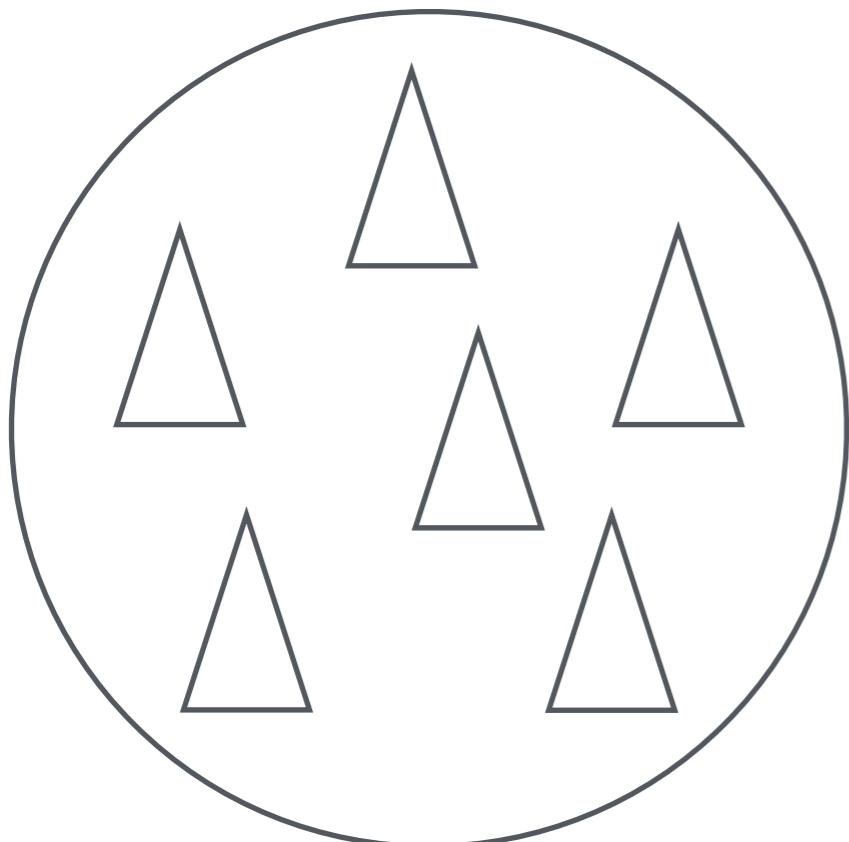
Language = Set of Sentences



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

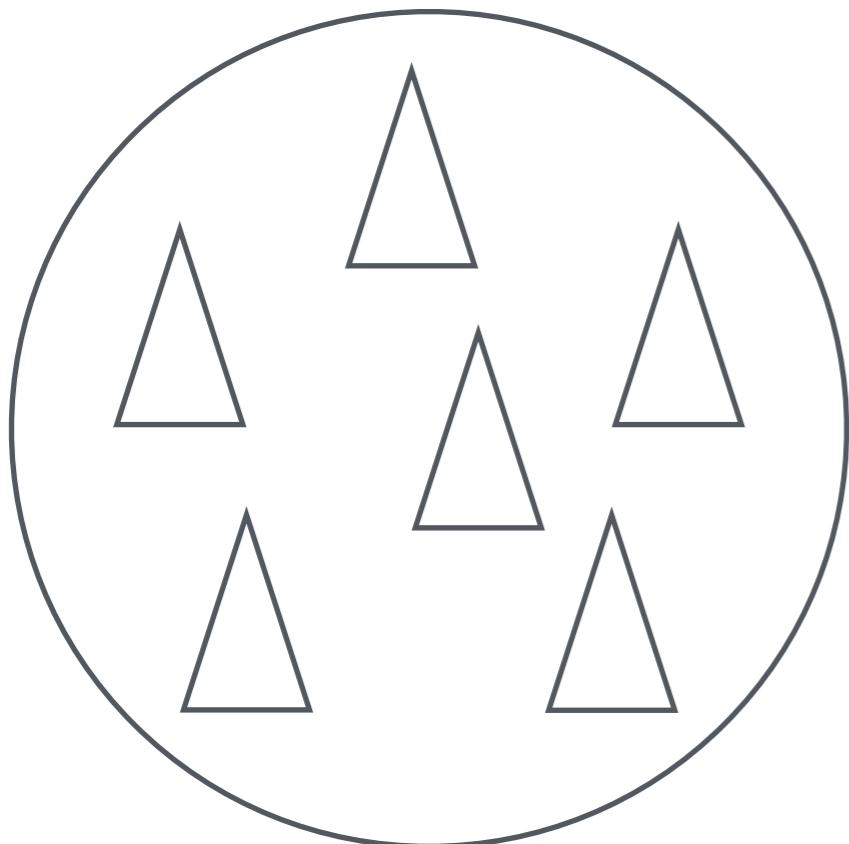
text is a convenient interface for writing and reading programs

Language = Set of Trees



tree is a convenient interface for transforming programs

Tree Transformation

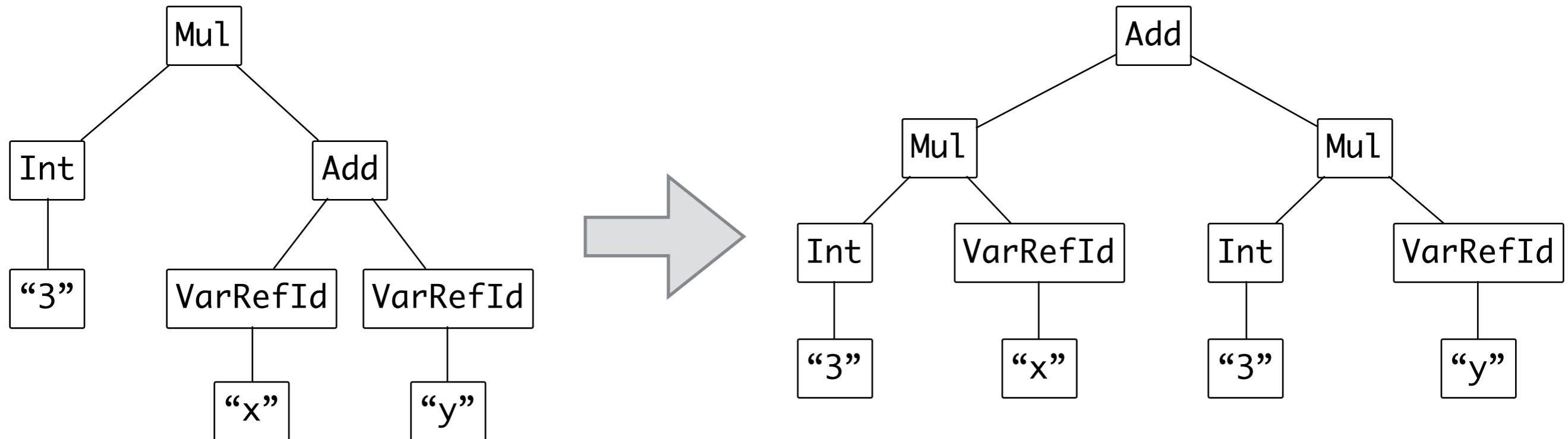


Syntactic
coloring
outline view
completion

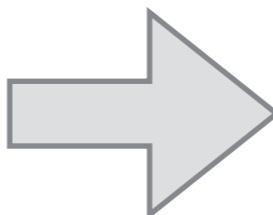
Semantic
transform
translate
eval
analyze
refactor
type check

tree is a convenient interface for transforming programs

Tree Transformation



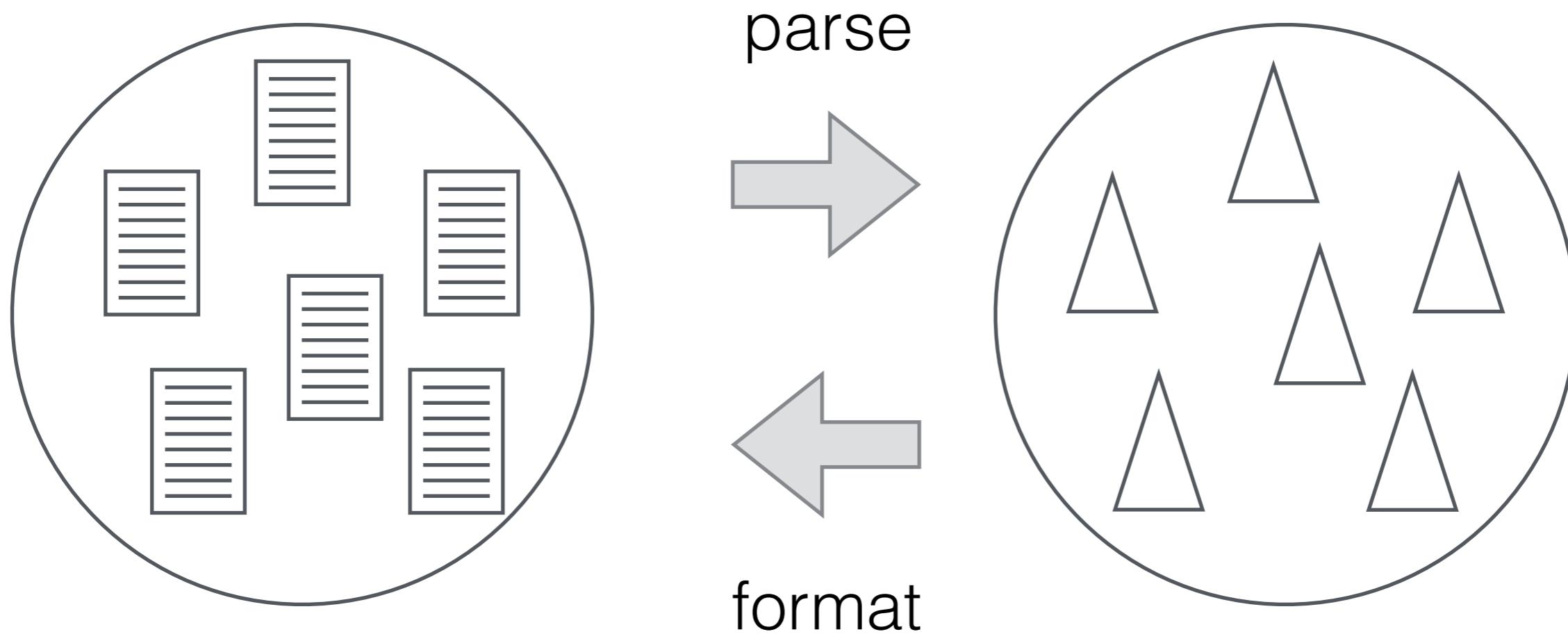
```
Mul(Int("3"),
     Add(VarRefId("x"),
          VarRefId("y"))))
```



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

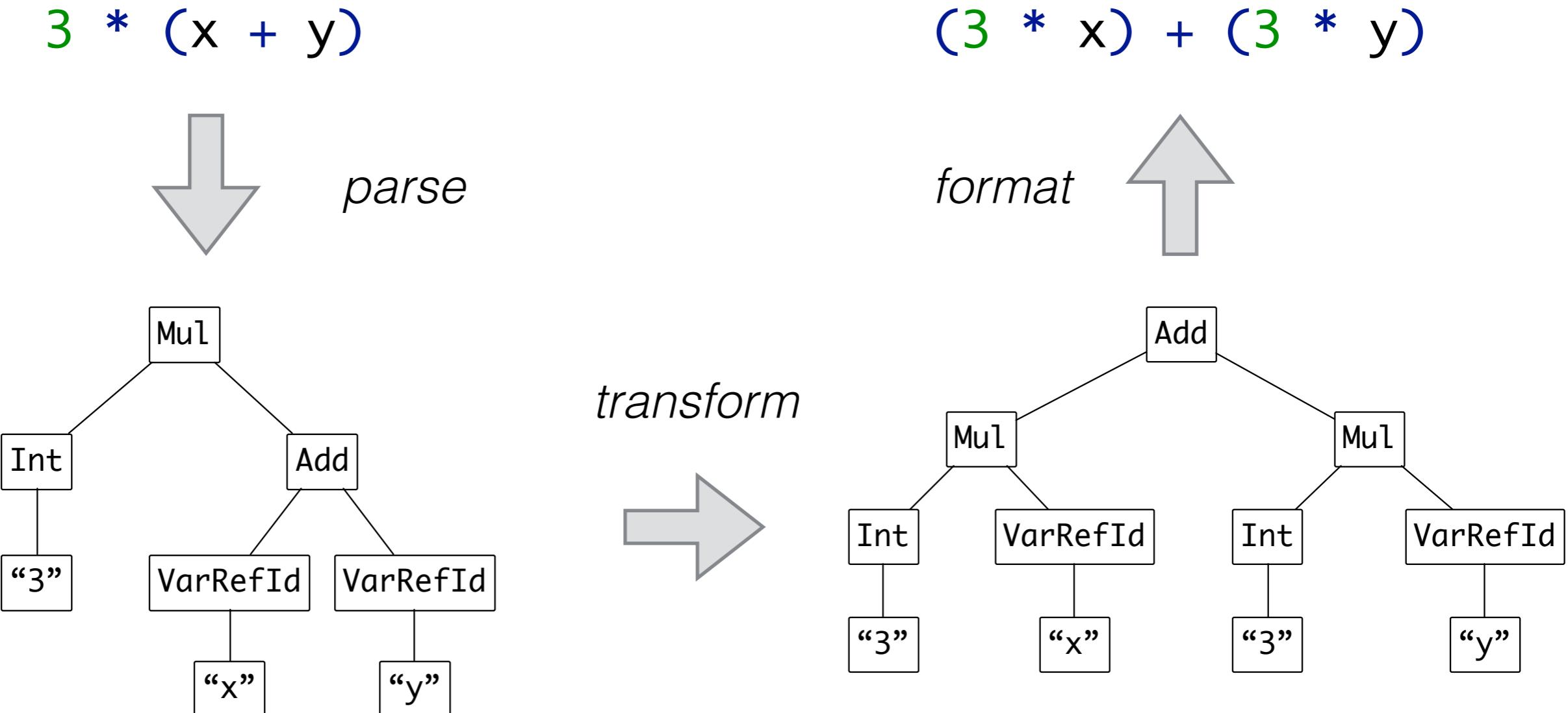
```
Mul(e1, Add(e2, e3)) -> Add(Mul(e1, e2), Mul(e1, e3))
```

Language = Sentences *and* Trees



different representations convenient for different purposes

From Text to Tree and Back



SDF3 defines Trees *and* Sentences

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



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

Grammar Engineering in Spooftax

The screenshot shows the Eclipse IDE interface with the following windows:

- Package Explorer:** Shows the project structure with files like `record01.partition.index`, `test00.aterm`, `test00.lmr`, `test00.partition.index`, `test00.pp.lmr`, `test01.aterm`, `test01.lmr`, `icons`, `include`, `lib`, `META-INF`, `src-gen`, and `syntax` containing grammar files.
- Syntax View:** Displays the contents of `ExpressionsAmb.sdf3`. It shows various Expr rules such as `Expr.True`, `Expr.False`, `Expr.VarRef`, `Expr.Add`, `Expr.Sub`, `Expr.Mul`, `Expr.Div`, `Expr.And`, `Expr.Or`, `Expr.Eq`, `Expr.App`, `Expr.If`, `Expr.Fun`, `ArgDecl.ArgDecl`, `Expr.Let`, `Expr.LetRec`, `Expr.LetPar`, `DefBind.DefBind`, and `DefBind.DefBindTyped`.
- Analysis View:** Displays the contents of `amb01.lmr`. It contains a program named `test01` with a module `A` defining a variable `x` and a module `B` importing `A` and defining `y` as `x + 1`.
- Generation View:** Displays the contents of `test01.aterm`. It shows the generated Aterm representation of the program, including a `Program` node with `test01`, a `Module` node for `A` with a `DefBind` for `x`, and a `Module` node for `B` with an `Import` of `A` and a `DefBind` for `y`.

The status bar at the bottom indicates the file is `Writable`, has `5 : 2` errors, and is `Analyzing files (legacy)`.

Ambiguity

The screenshot shows the Eclipse IDE interface with several open windows and toolbars.

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

Toolbars: Syntax, Analysis, Generation

Left View (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
```

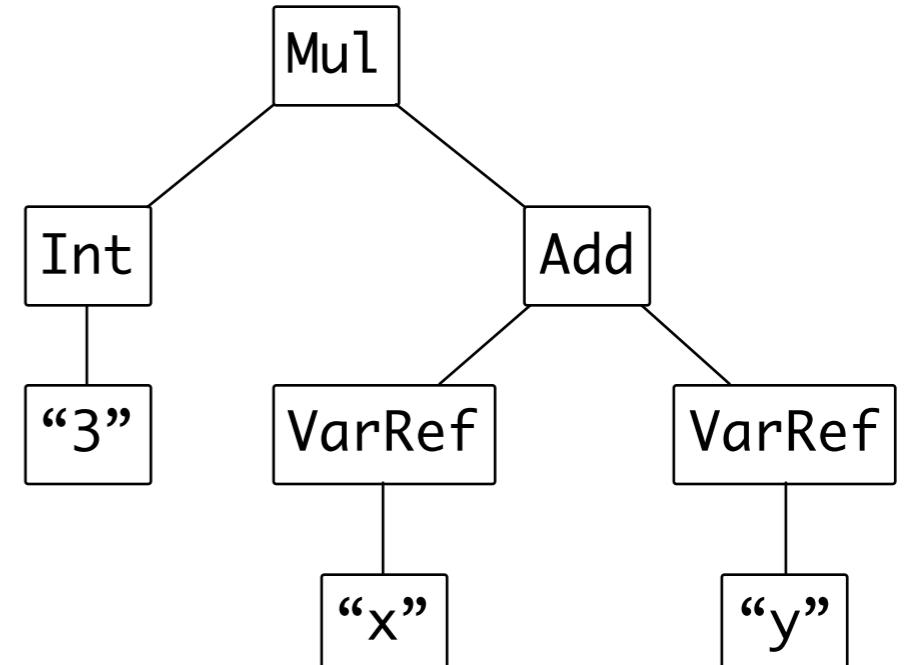
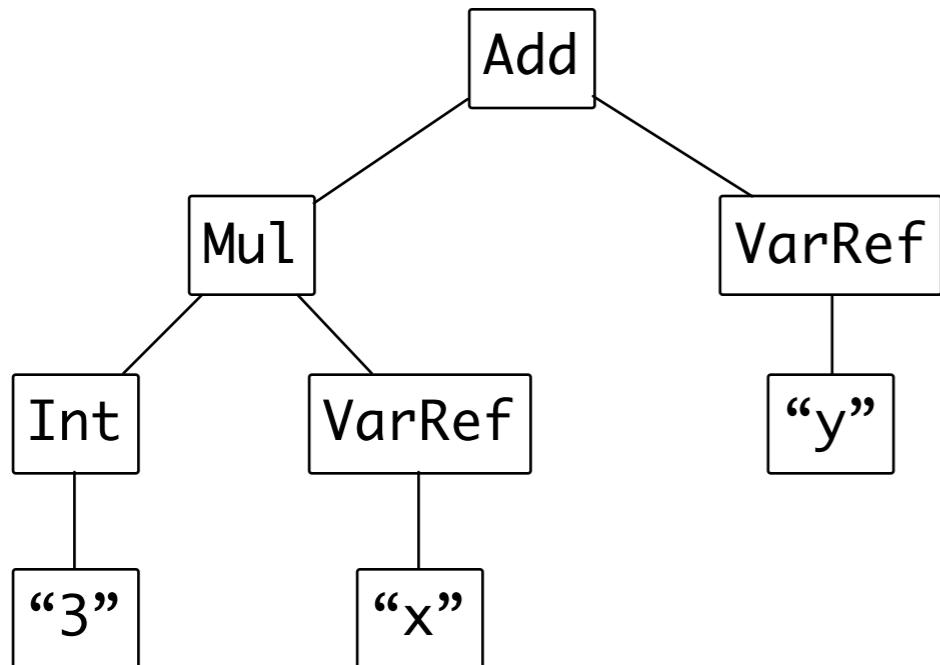
Middle View (test01.aterm):

```
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    , Mul(
21      Add(VarRef("a"), VarRef("b"))
22      , Sub(VarRef("x"), Int("1")))
23    )
24  )
25  ]
26 )
```

Status Bar: Writable, Smart Insert, 10 : 10, Analyzing files (legacy)

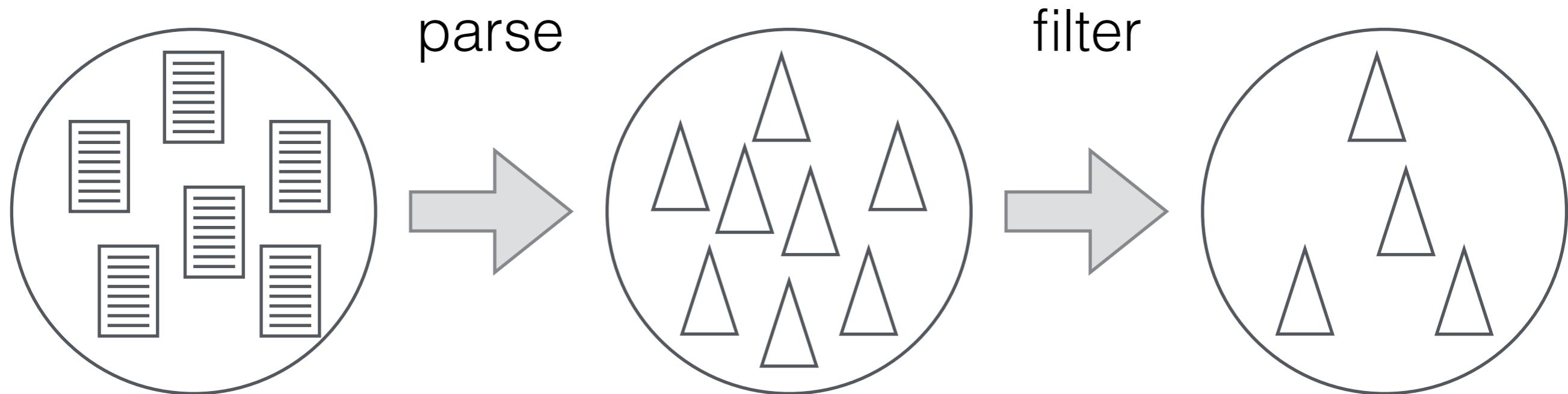
Ambiguity

3 * x + y



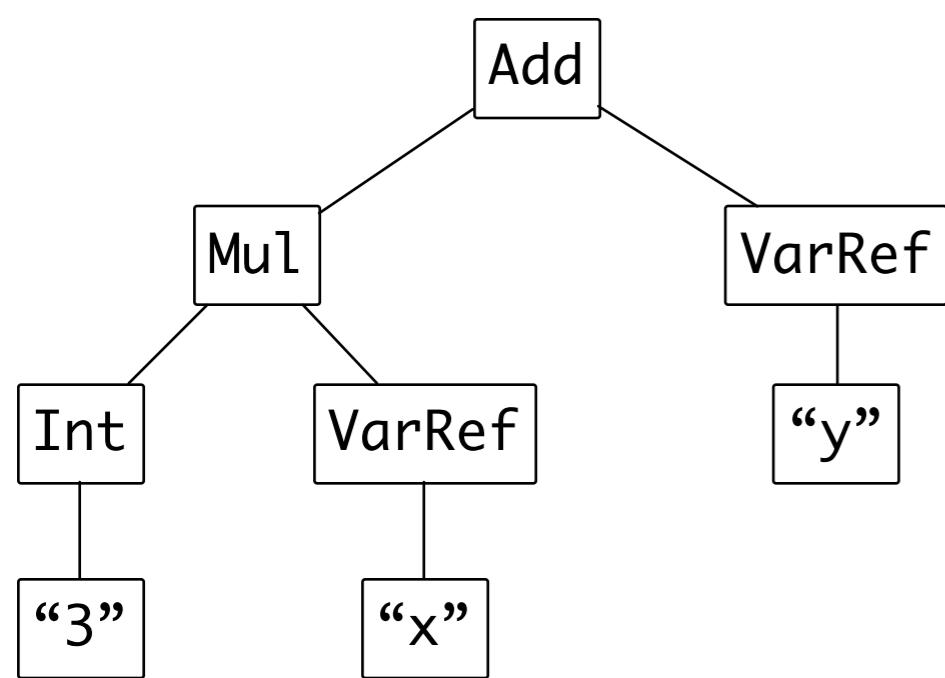
$t1 \neq t2 \wedge \text{format}(t1) = \text{format}(t2)$

Declarative Disambiguation

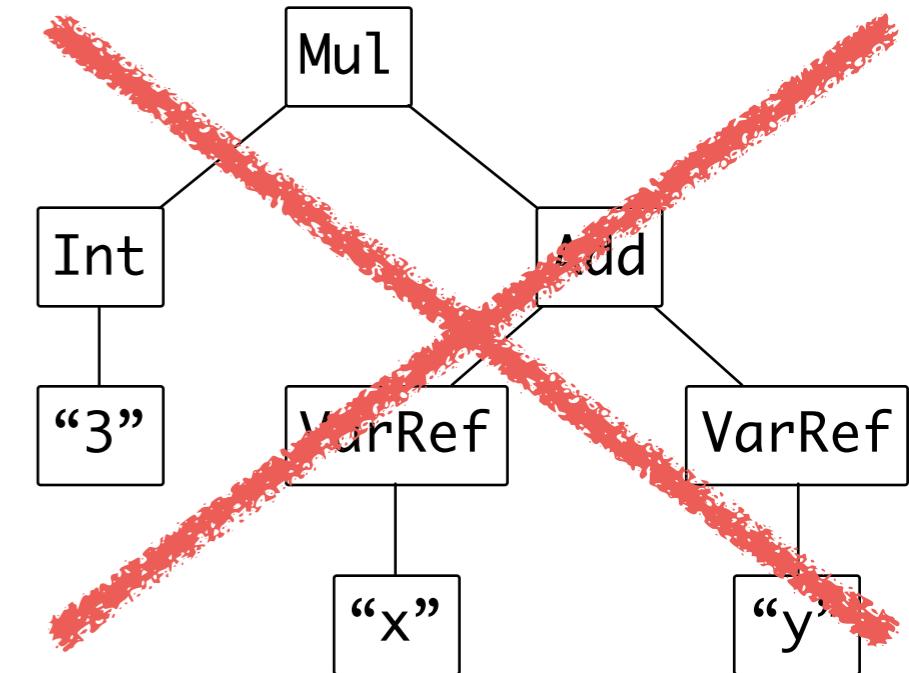


Disambiguation Filters [Klint & Visser; 1994], [Van den Brand, Scheerder, Vinju, Visser; CC 2002]

Priority and Associativity



3 * x + y



context-free syntax

`Expr.Int = INT`

`Expr.Add = <<Expr> + <Expr>> {left}`

`Expr.Mul = <<Expr> * <Expr>> {left}`

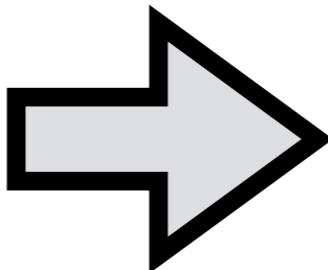
context-free priorities

`Expr.Mul > Expr.Add`

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

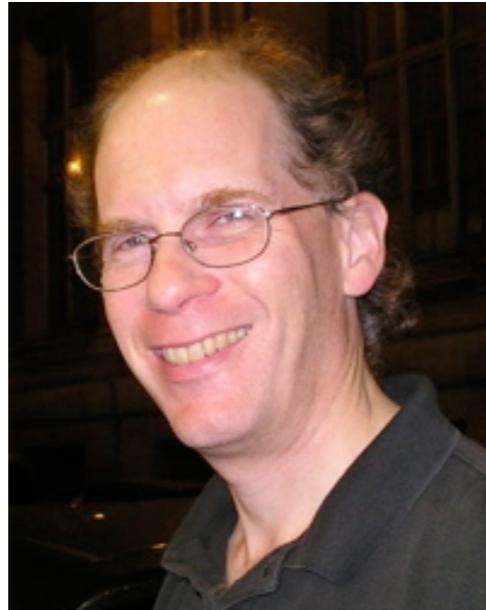
- (1) language-specific grammar + disambiguation rules
- (2) language-independent spec of well-formed trees for grammar
- (3) formatting based on layout hints in grammar
- (4) parser generated automatically
- (4') no need to understand parsing algorithm
- (4'') debugging in terms of representation
- (5) syntactic and semantic operations abstract from parsing

Declare Your Names

A Theory of Name Resolution



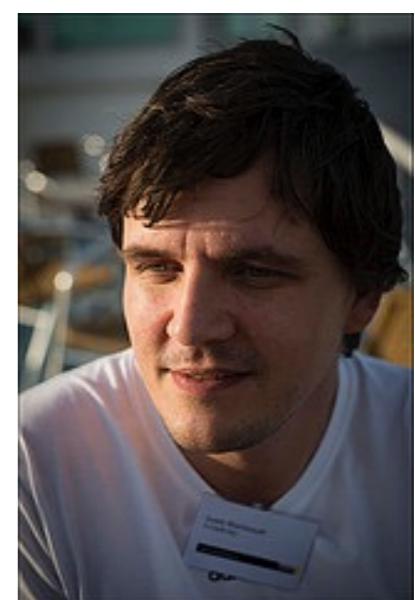
Pierre
Neron¹



Andrew
Tolmach²



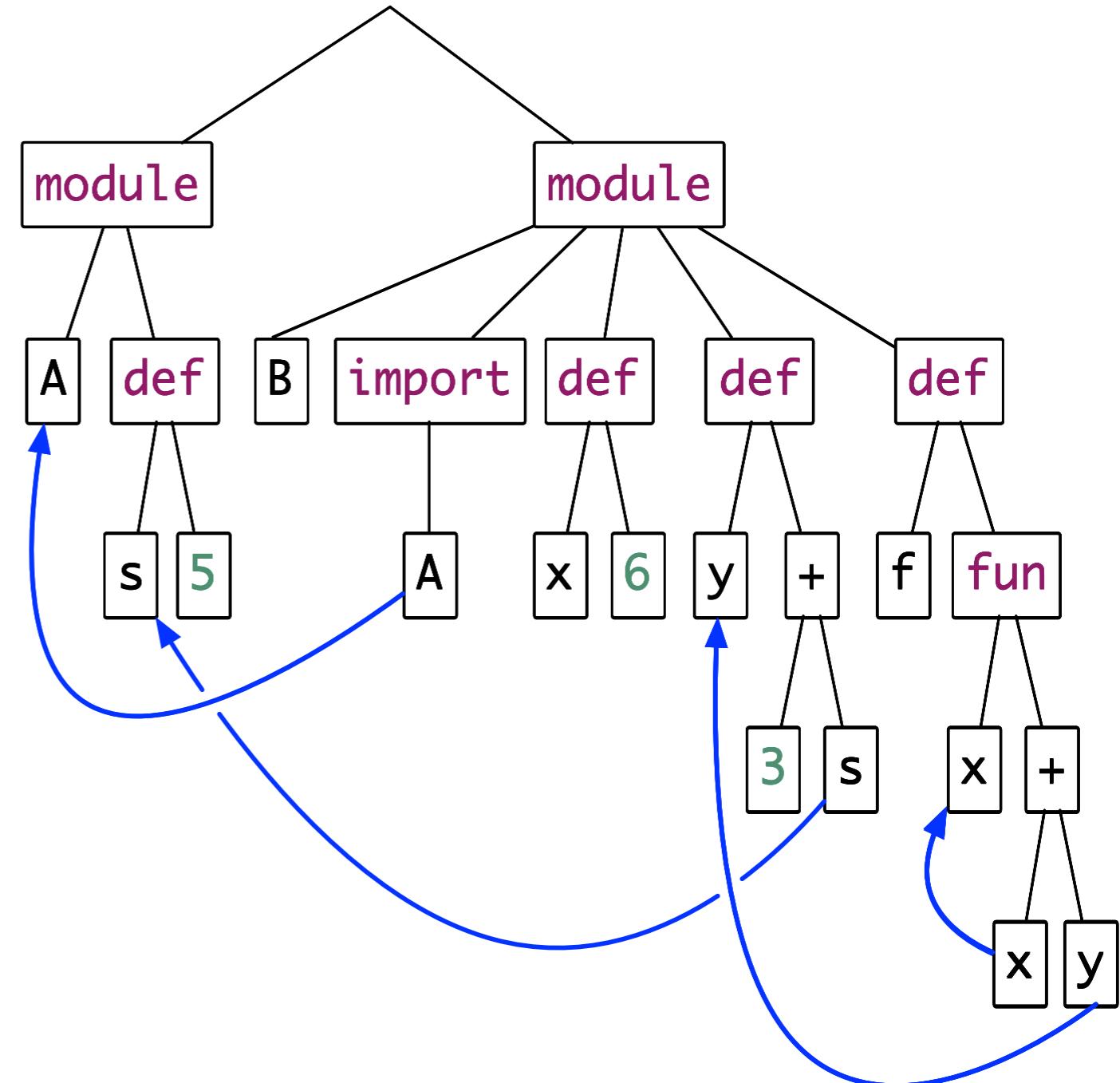
Eelco
Visser¹



Guido 1
Wachsmuth

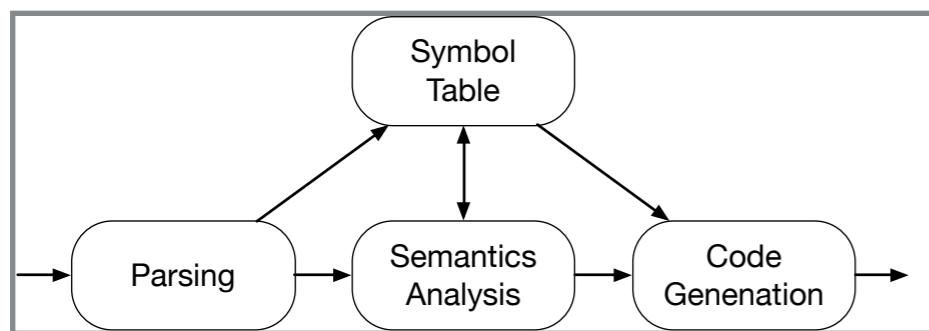
Language = Set of Graphs

```
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

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

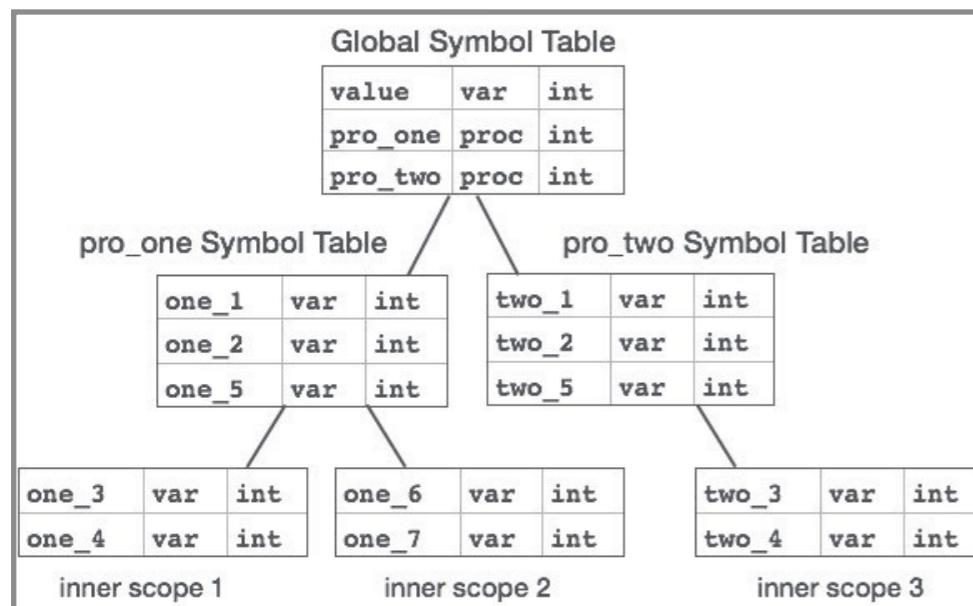
Semantics

A screenshot of an IDE showing Java code. The code defines a class 'A' with a static integer 'x' and a method 'plus' that returns 'y + x'. The variable 'x' is highlighted in yellow, corresponding to the variable in the semantic rule.

```
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

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

The NaBL Name Binding Language

The screenshot shows the Eclipse IDE interface with several open files:

- Expressions.sdf**: A file containing binding rules for variables. It includes definitions for ArgDecl, DefBind, DefBindTyped, VarRef, FldAccess, New, and FldBind.
- name-binding.na**: A file containing the NaBL source code. It defines a program record01 with records Point, ColorPoint, and Line, and various def statements.
- amb01.lmr**: A file containing a record01.lmr file, which is a concrete representation of the NaBL code.
- test01.lmr**: A file containing a test01.lmr file.
- record01.lmr**: A file containing a record01.lmr file.
- record01.aterm**: A file containing a record01.aterm file, showing the abstract syntax tree (AST) representation of the NaBL code.

The code in **name-binding.na** is as follows:

```
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
```

The code in **record01.aterm** is as follows:

```
1 , Decls
22 , DefBind(
33 "p"
24 , New(
45 TypeRef("Point")
56 , [FldBind("x", Int("1")), FldBind("y", Int("2"))]
67 )
78 )
89 )
90
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 )
```

Bottom status bar: Writable, Smart Insert, 54 : 21, Analyzing files (legacy)

Multi-Purpose Name Binding Rules

```
module names

namespaces Variable

binding rules

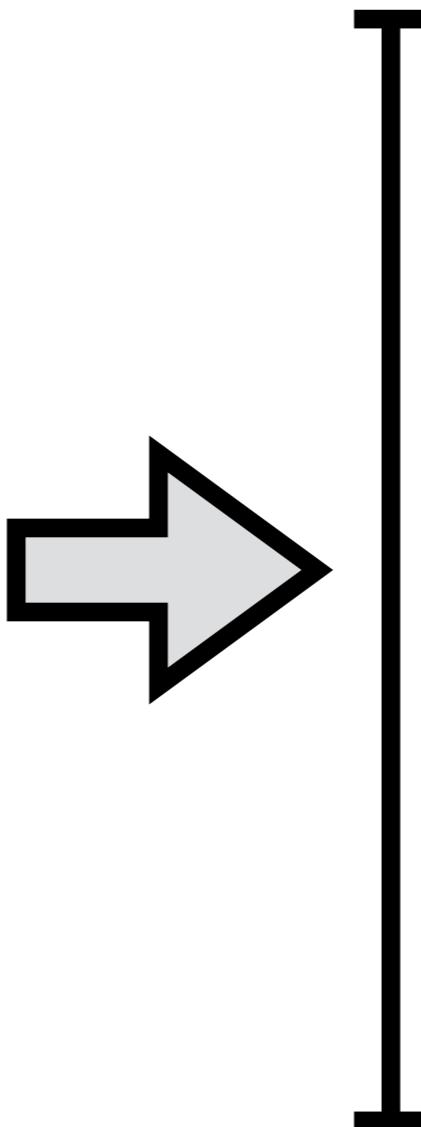
Var(x) :
  refers to Variable x

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)

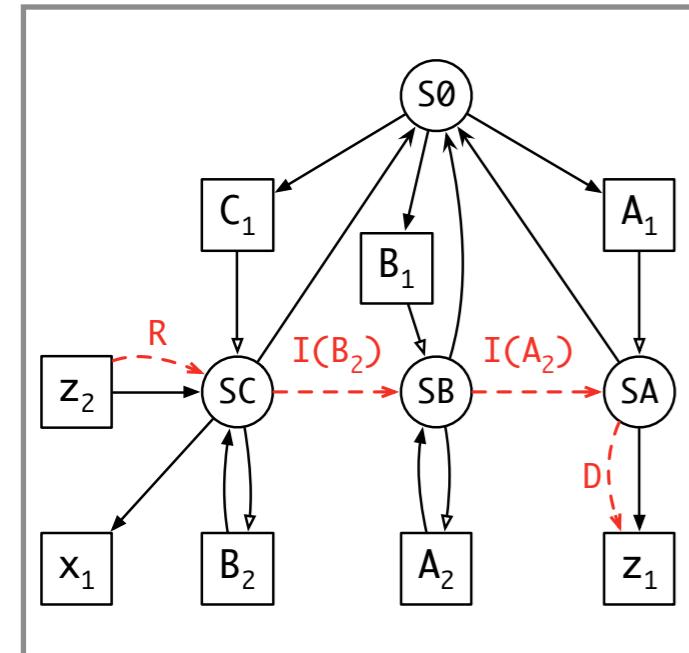
Design Goals

- 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*

**Scope
Graphs**

Supports

Resolution

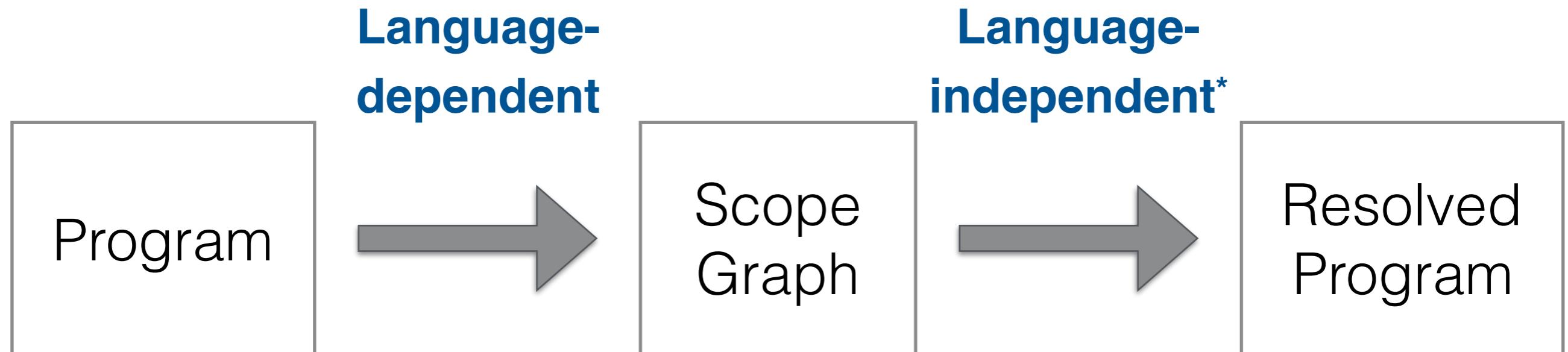
α -equivalence

IDE Navigation

Refactoring tools

Reasoning tools

Resolution Scheme



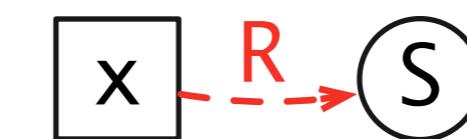
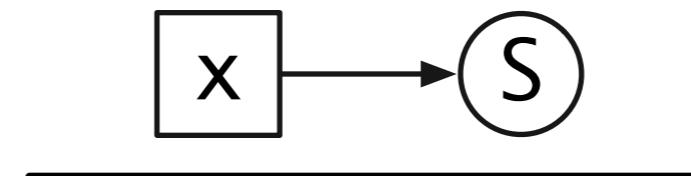
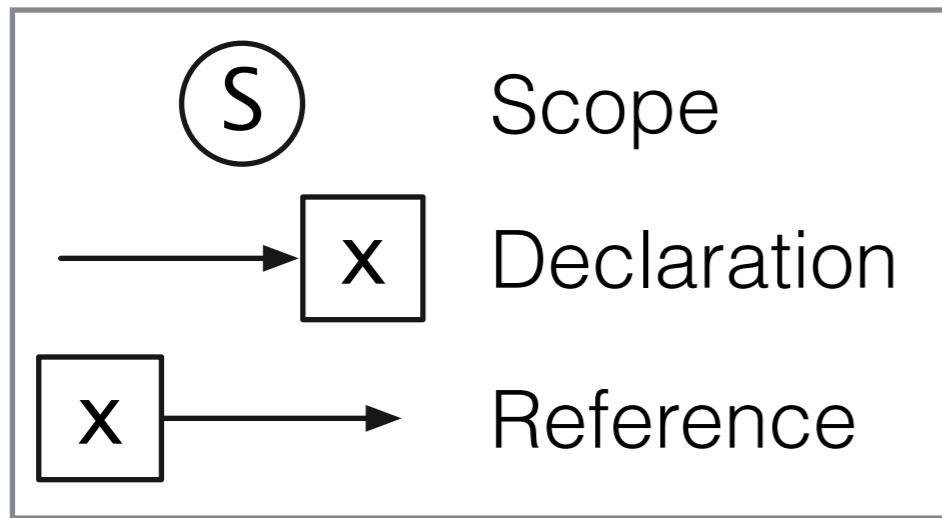
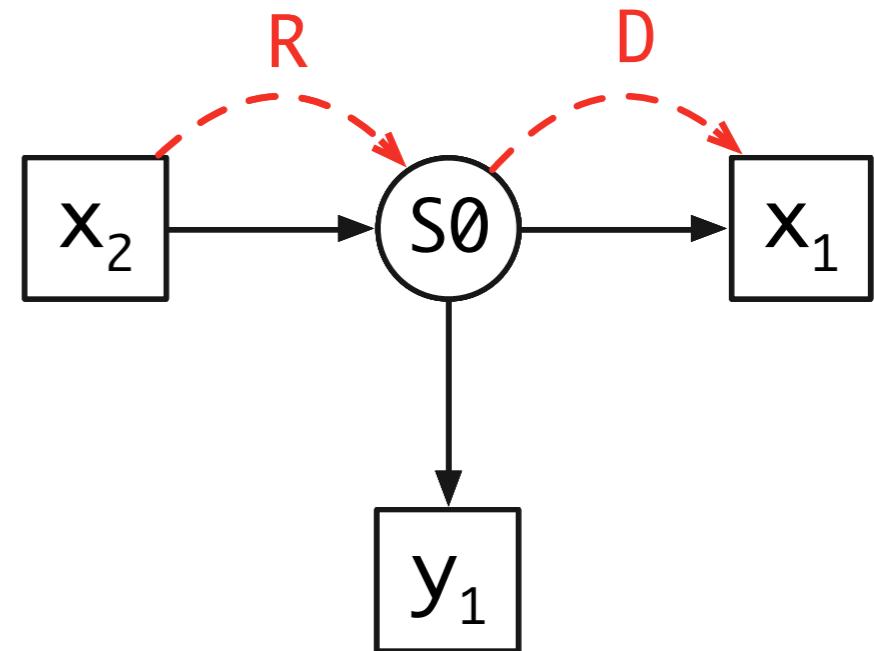
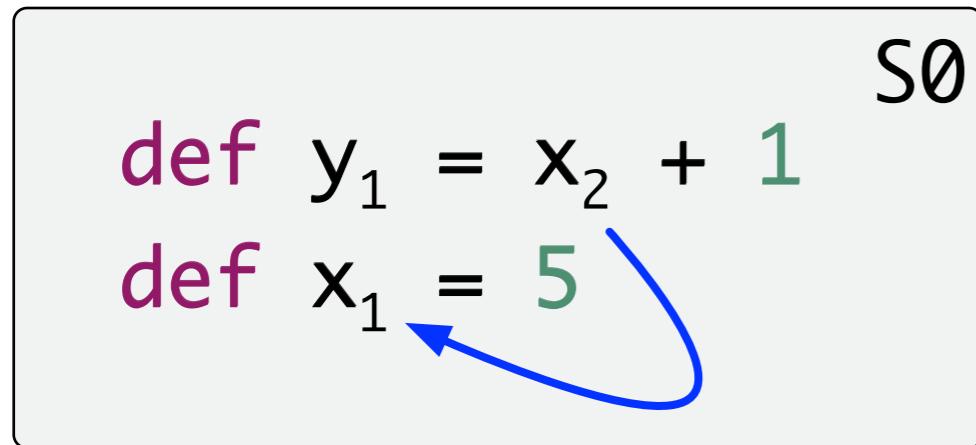
Resolution of a reference in a scope graph:

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

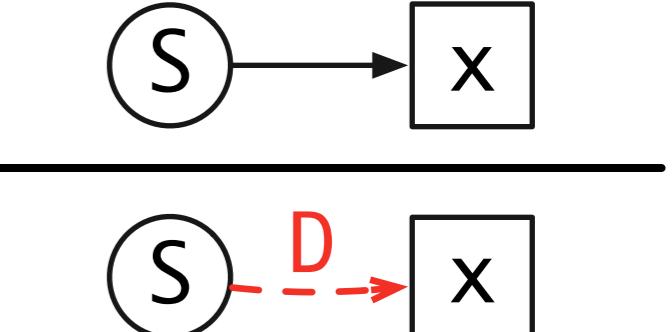
*Parameterized by notions of path **well-formedness**
and **ordering**

Scope Graphs by Example

Simple Scopes



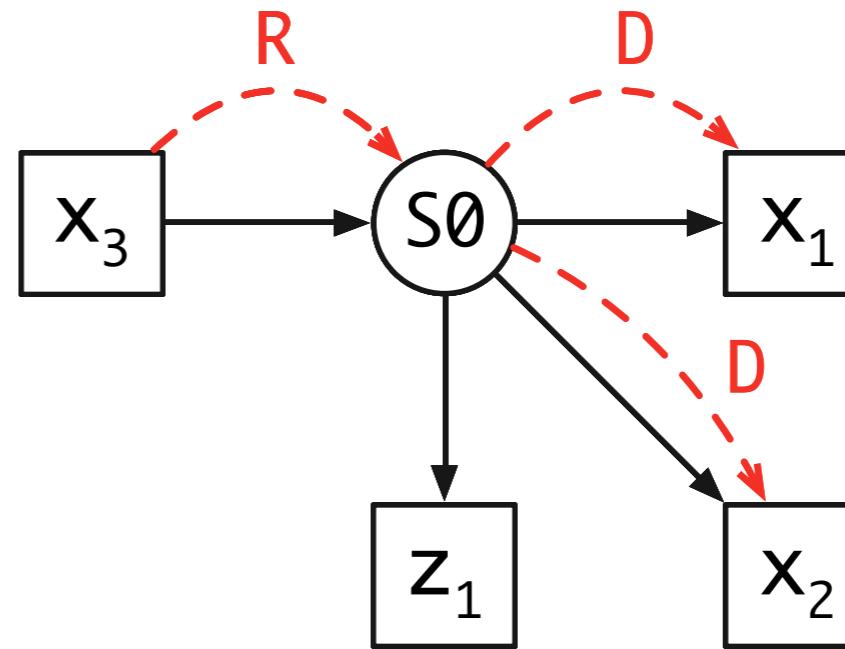
Reference Step



Declaration Step

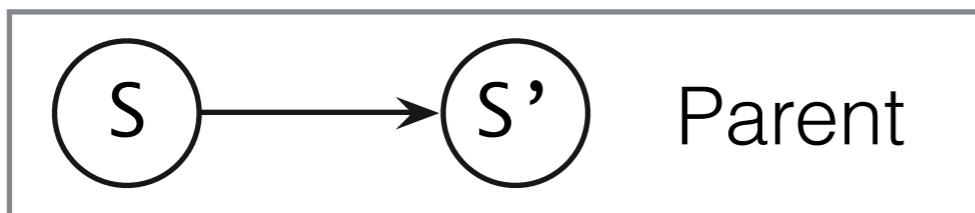
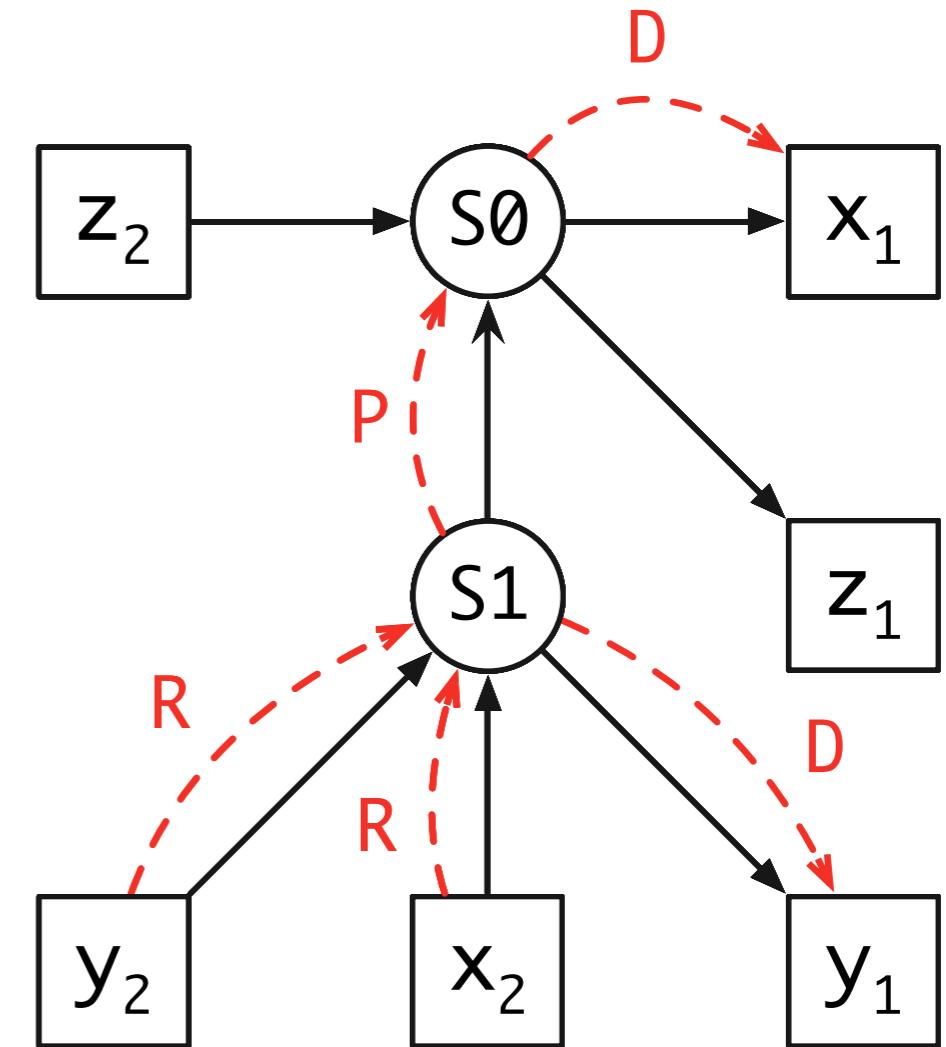
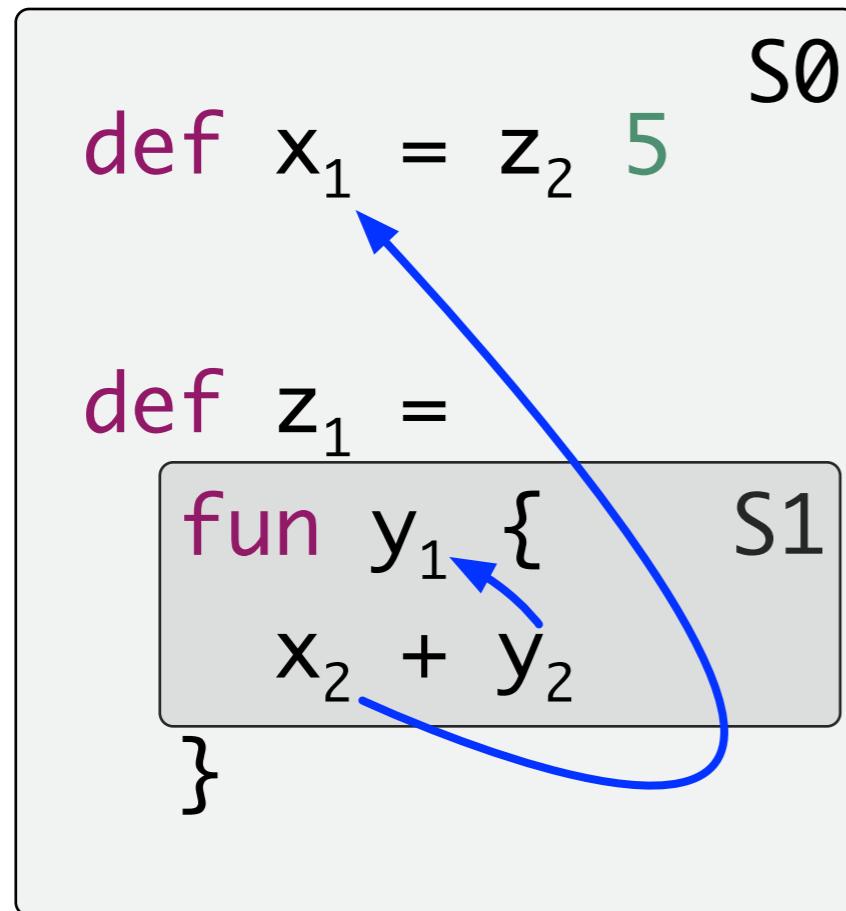
Ambiguous Resolutions

```
def x1 = 5      S0  
def x2 = 3  
def z1 = x3 + 1
```

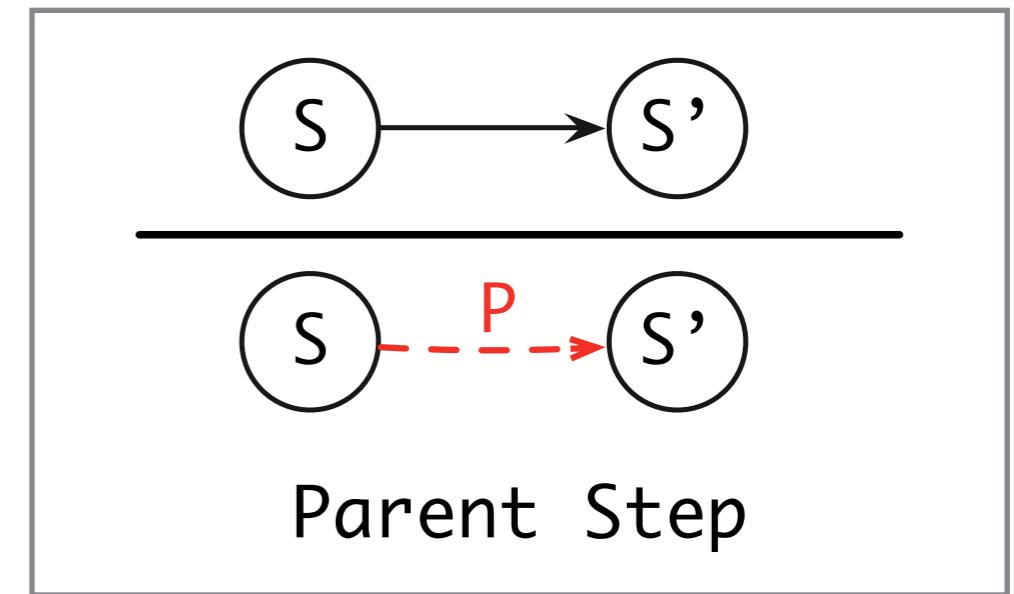


```
match t with  
| A x | B x => ...
```

Lexical Scoping



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



Shadowing

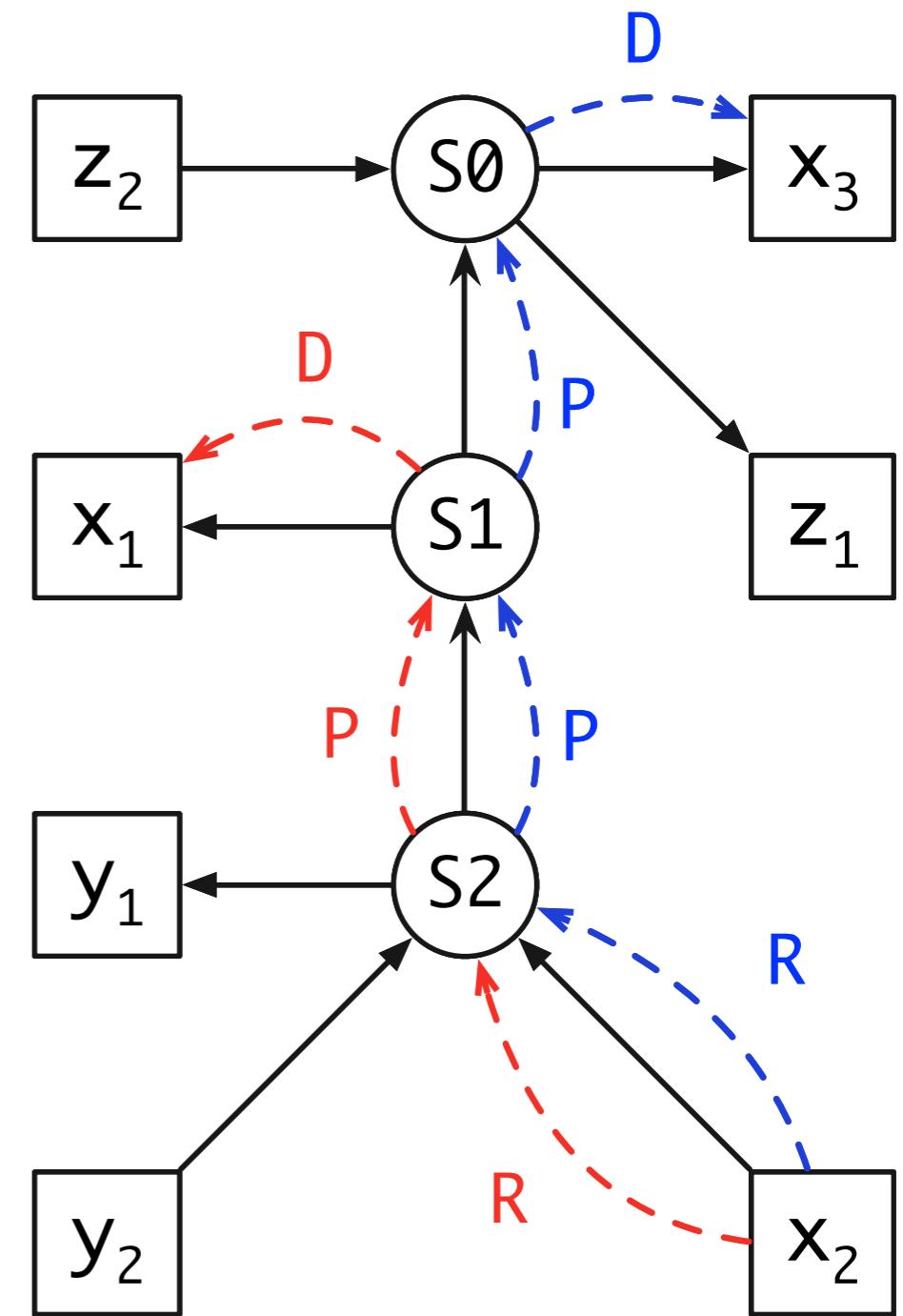
```

def x3 = z2 5 7 S0
def z1 =
  fun x1 {
    fun y1 {
      x2 + y2
    }
  }
}

```

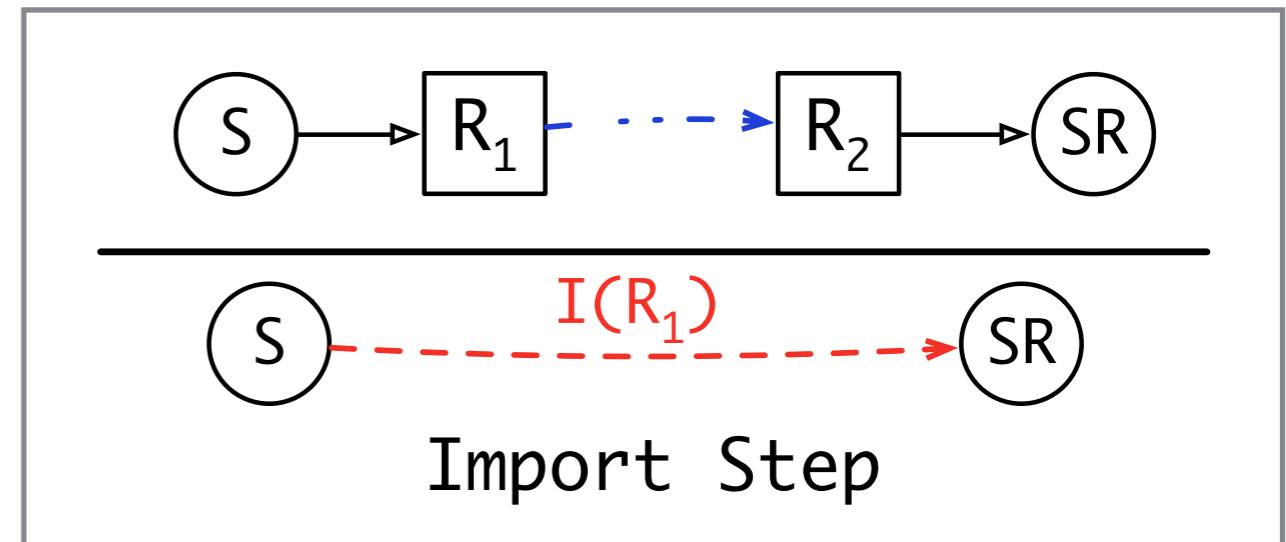
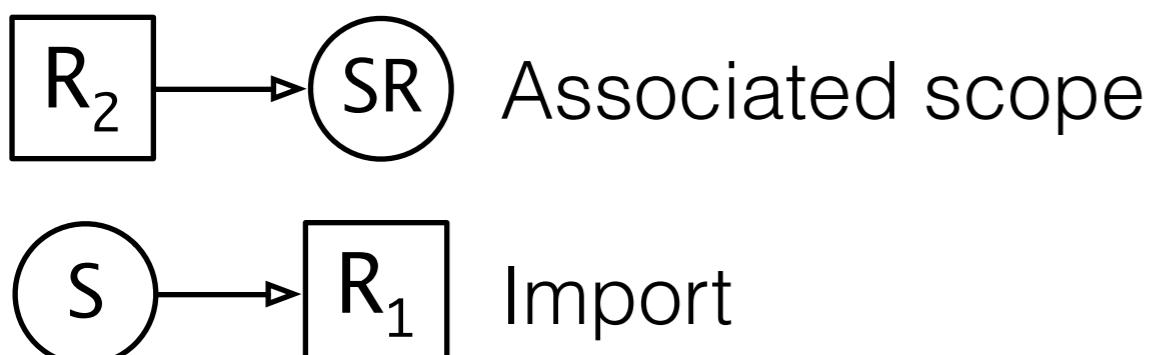
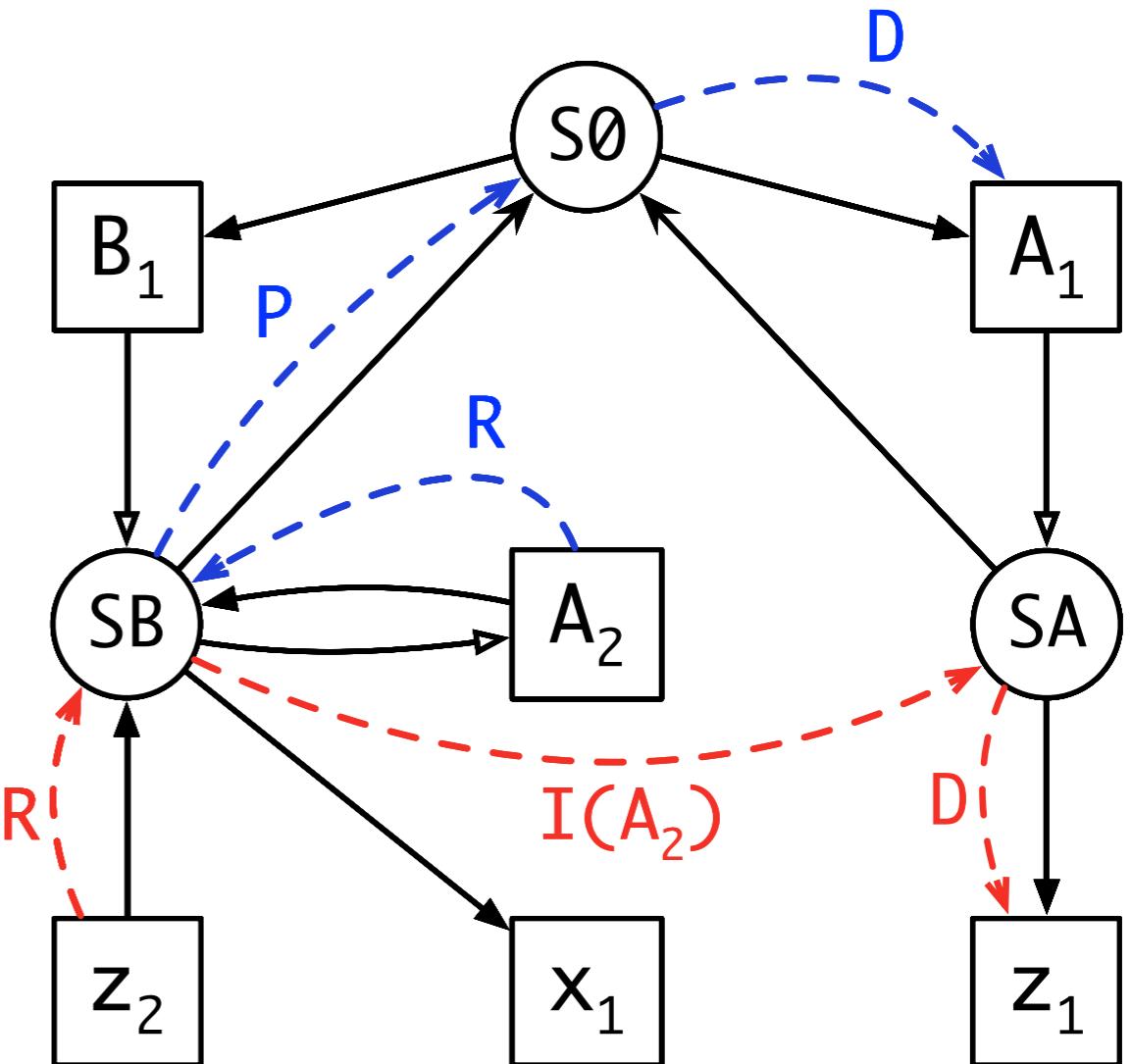
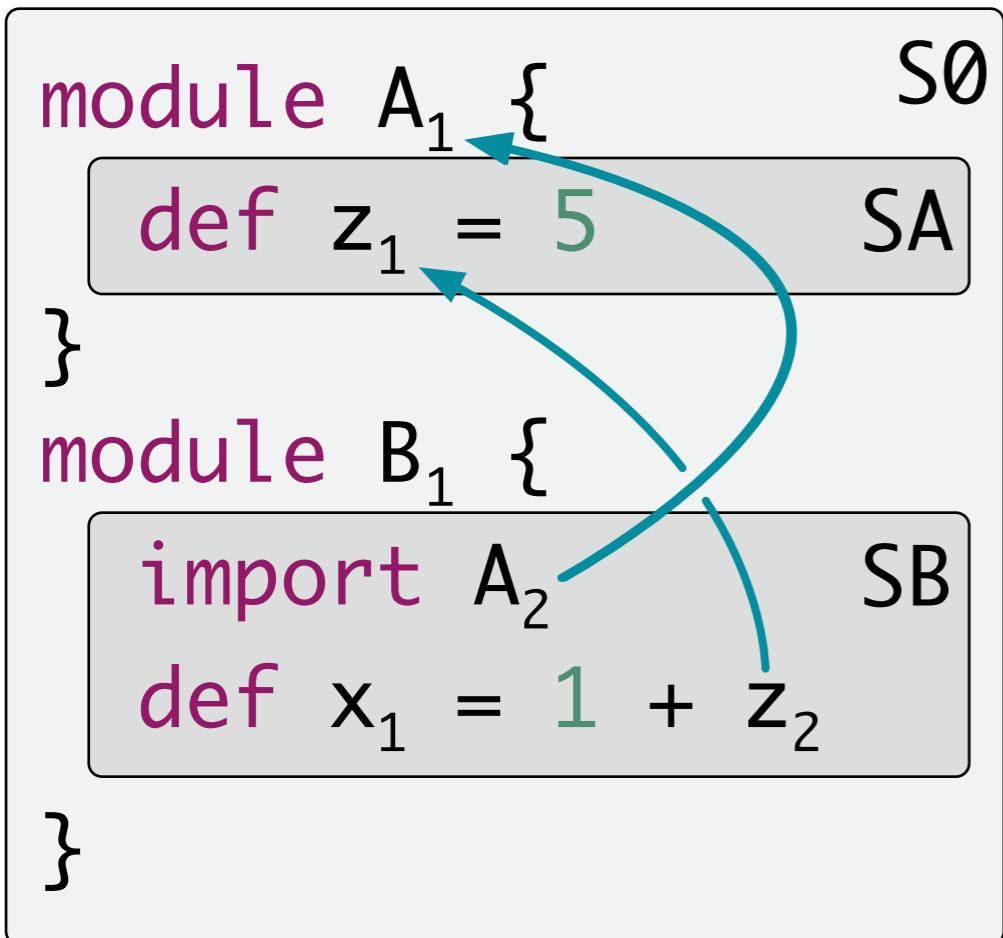
$$D < P.p$$

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



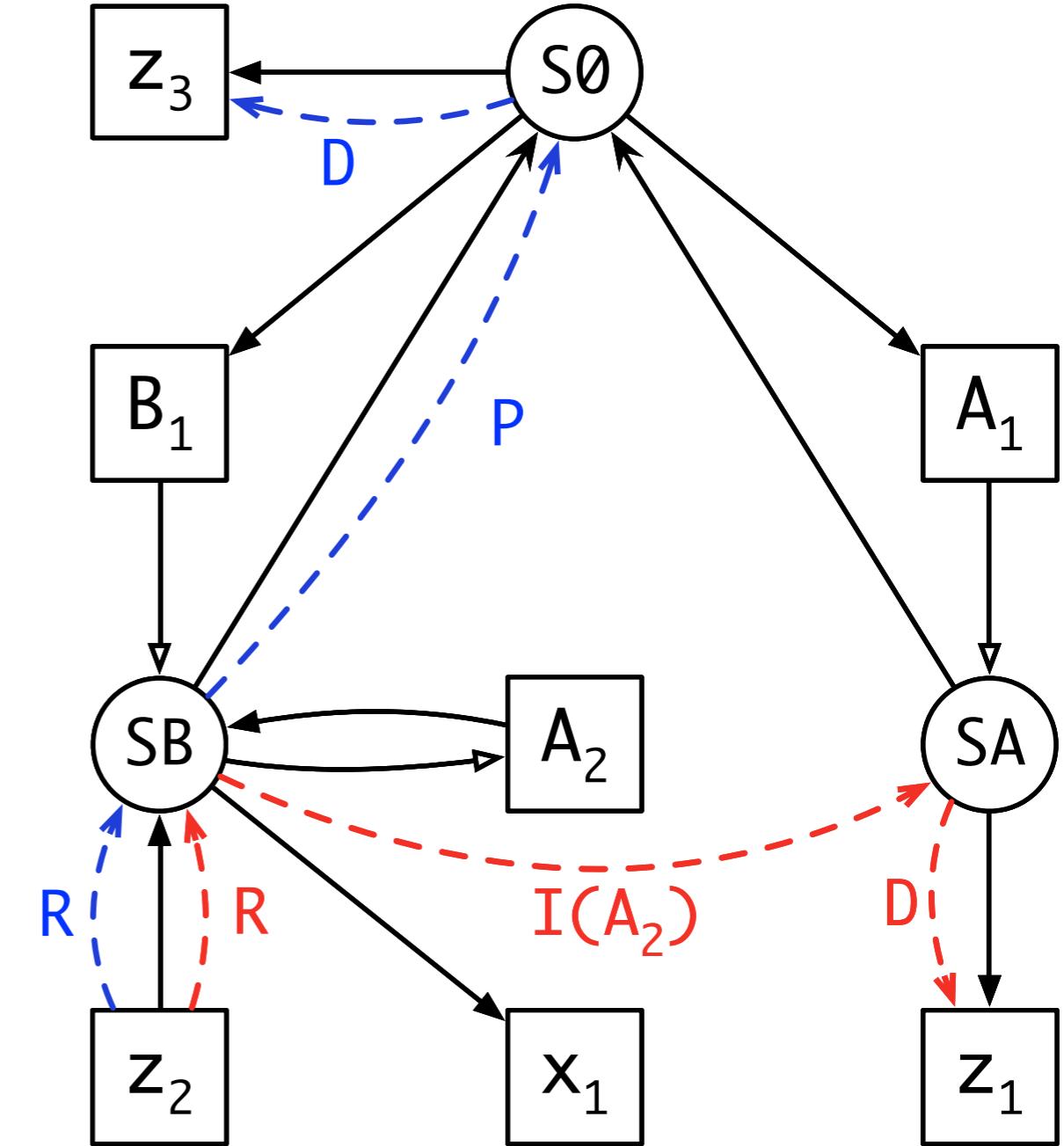
$$R.P.D < R.P.P.D$$

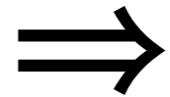
Imports



Imports shadow Parents

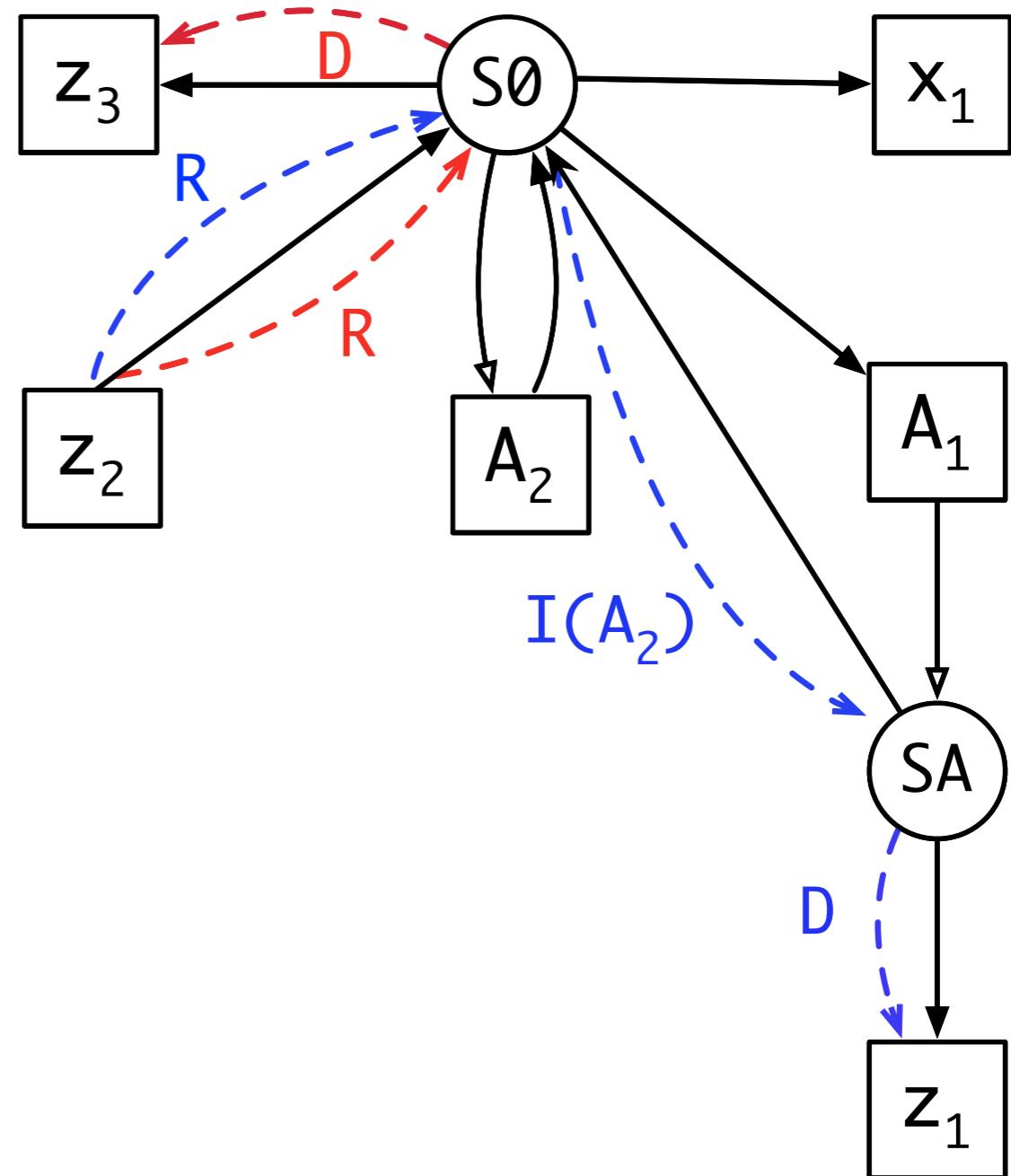
```
def z3 = 2 S0  
  
module A1 {  
    def z1 = 5 SA  
}  
  
module B1 {  
    import A2 SB  
    def x1 = 1 + z2  
}
```



 $I(_).p' < P.p$  $R.I(A_2).D < R.P.D$

Imports vs. Includes

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



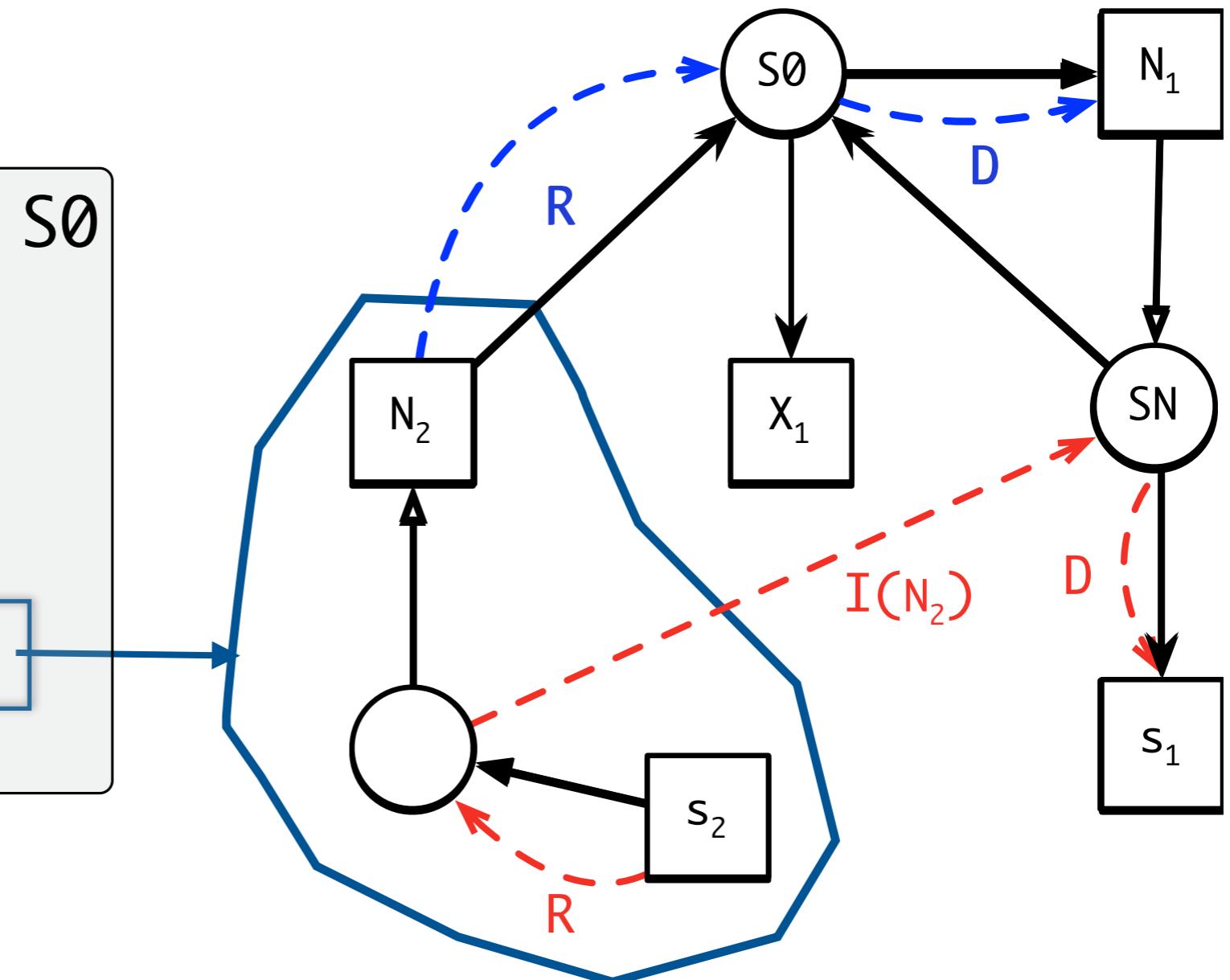
~~$D < I(A_2).p'$~~



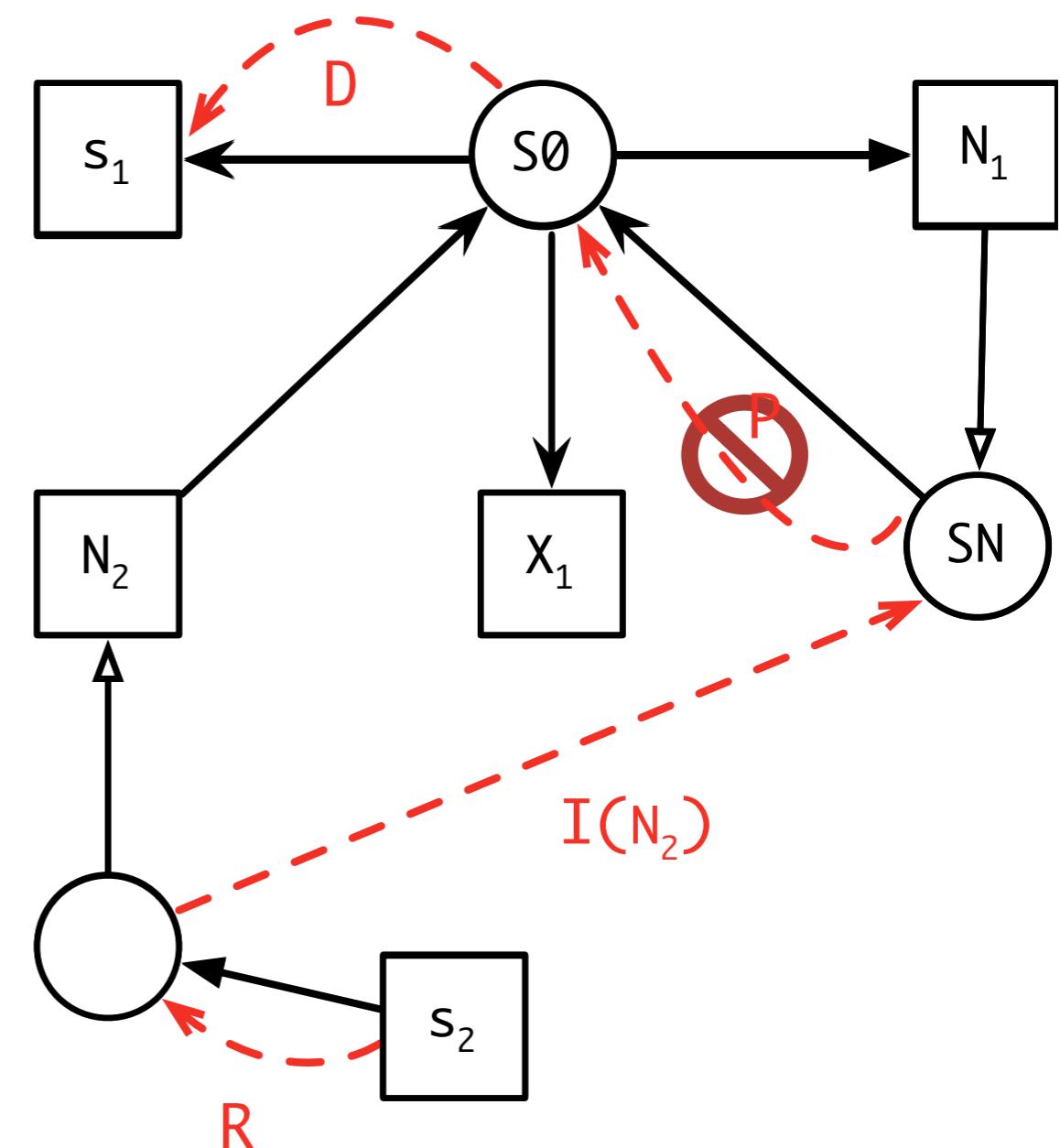
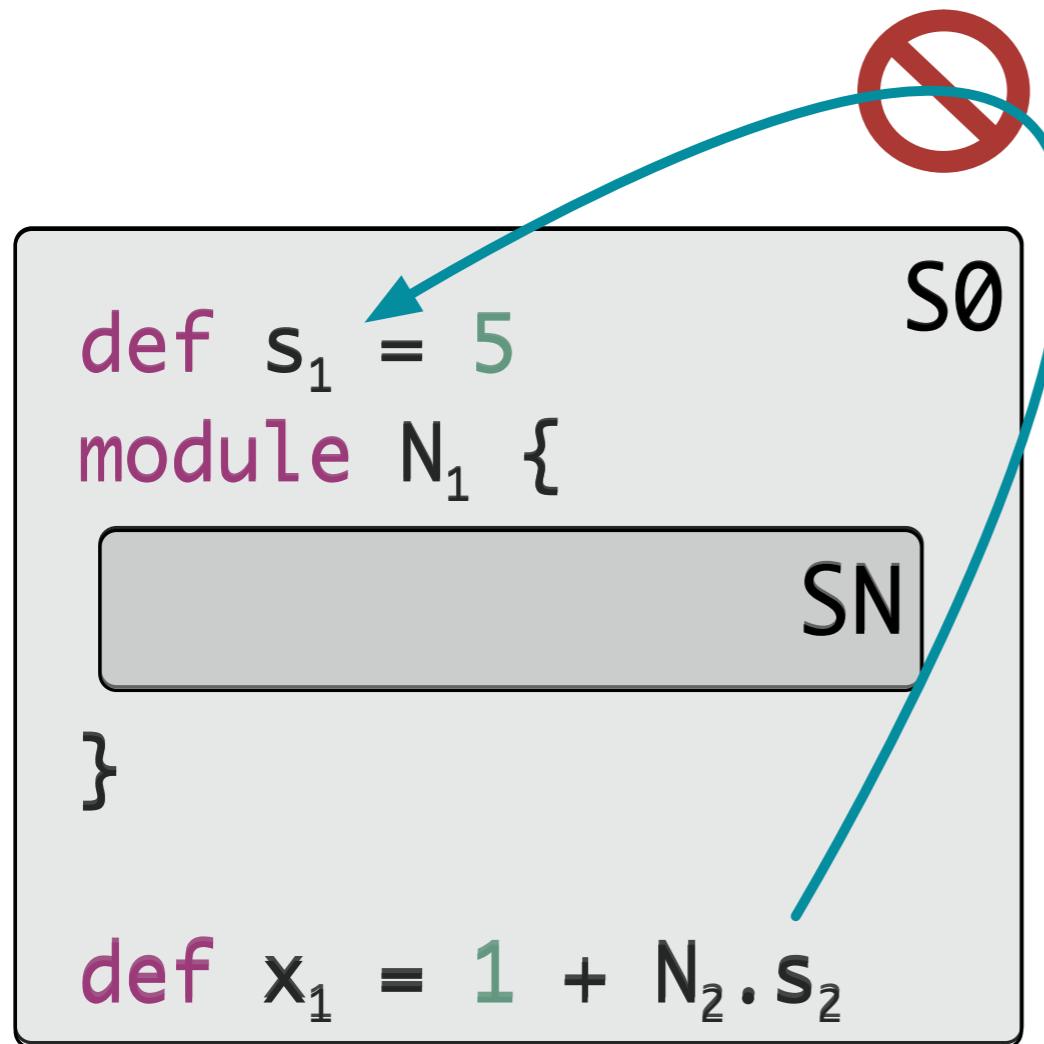
$R.D < R.I(A_2).D$

Qualified Names

```
module N1 {  
    def s1 = 5  
}  
  
module M1 {  
    def x1 = 1
```



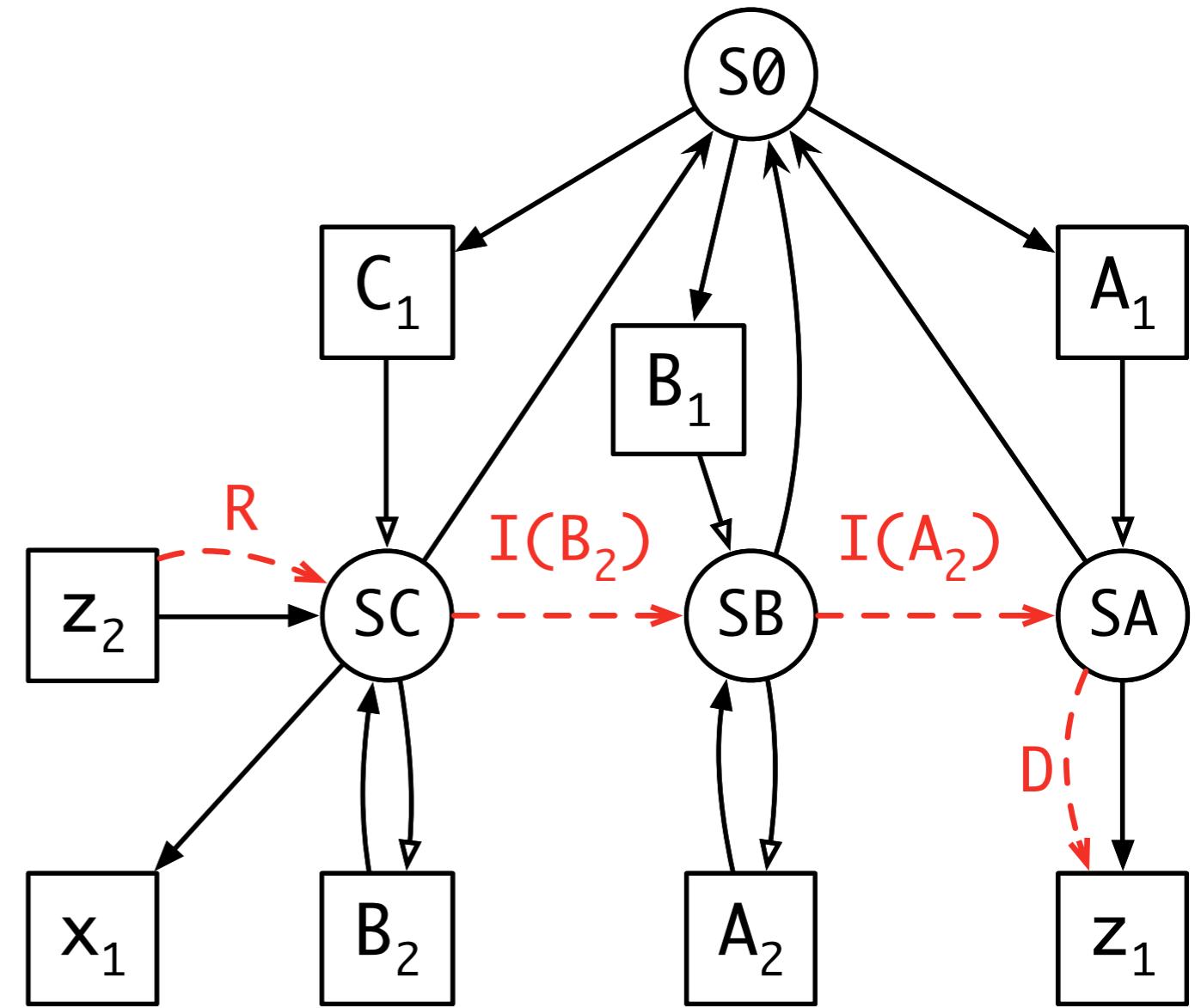
Import Parents



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

Transitive vs. Non-Transitive

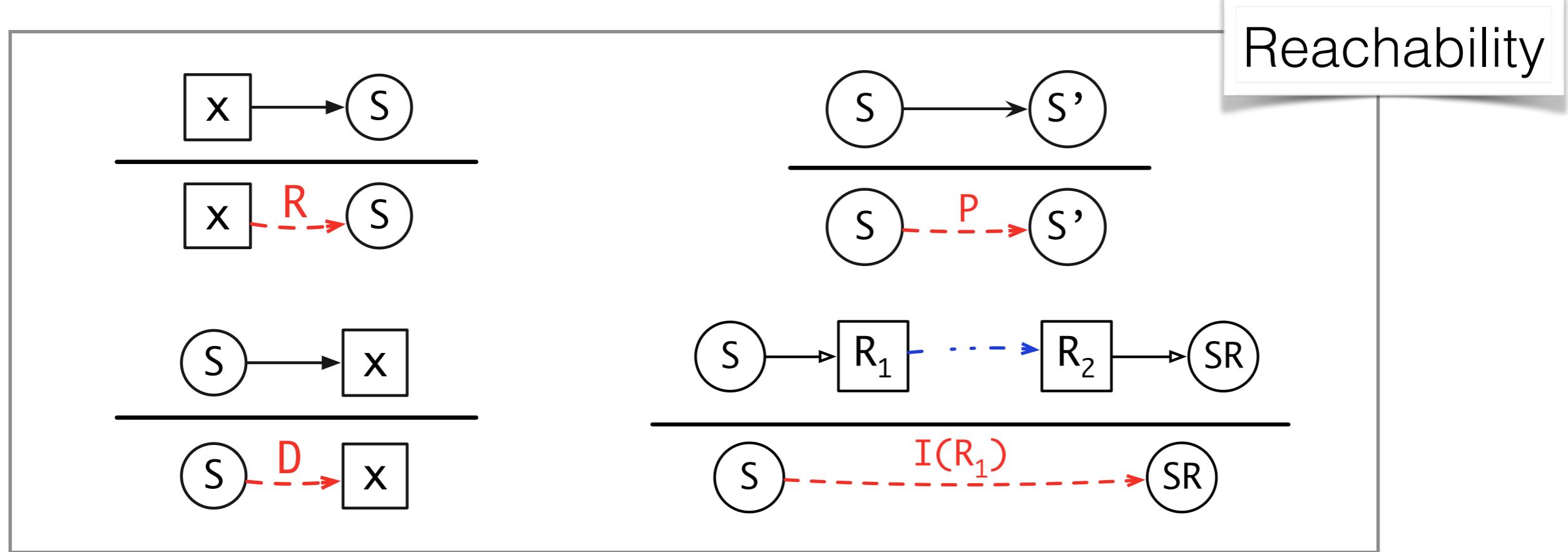
```
module A1 {  
    def z1 = 5 SA  
}  
module B1 {  
    import A2 SB  
}  
module C1 {  
    import B2  
    def x1 = 1 + z2 SC  
}
```



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

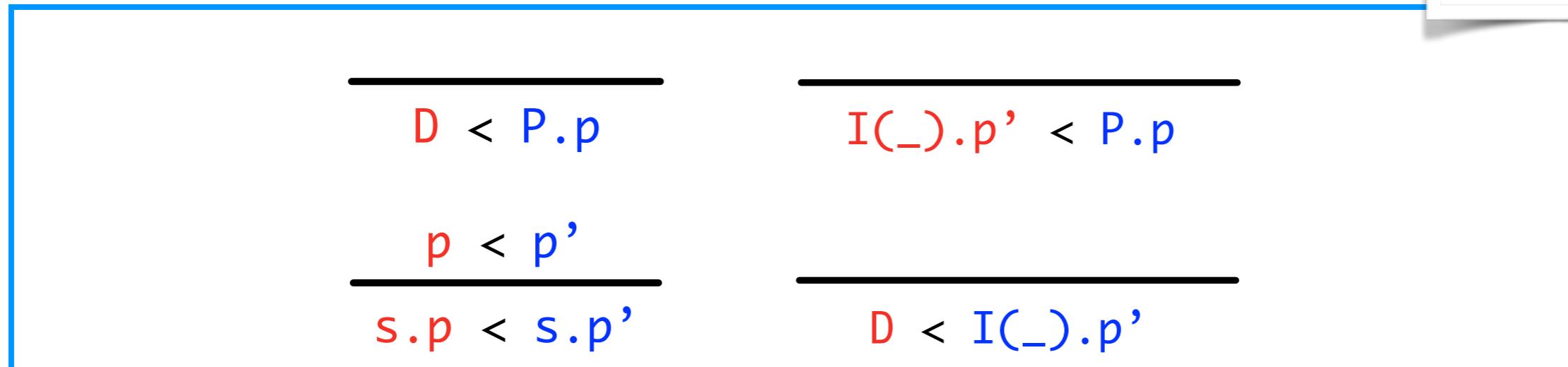
With non-transitive imports, a well formed path is **R.P*.I(_)?.D**

A Calculus for Name Resolution



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

Visibility

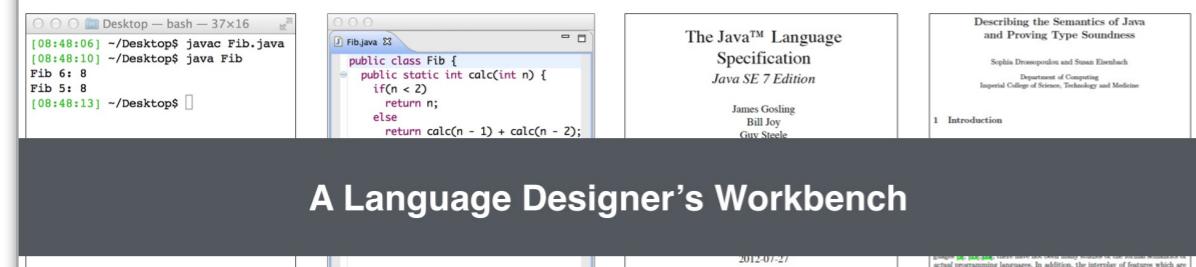
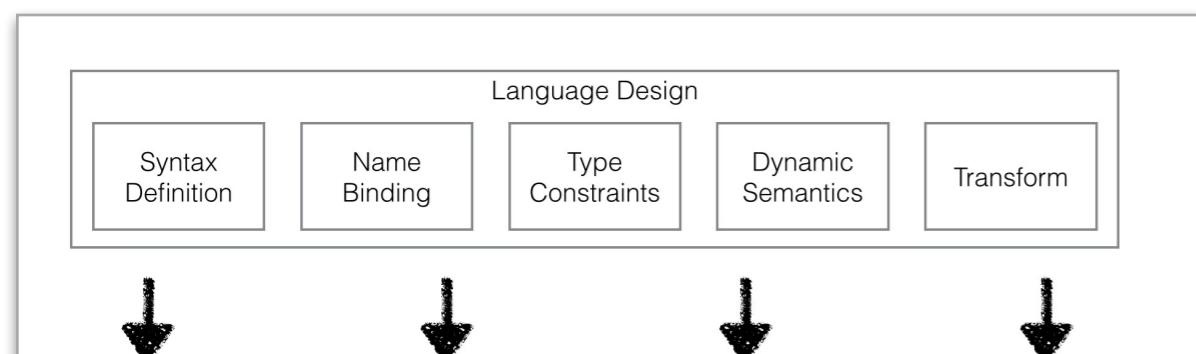
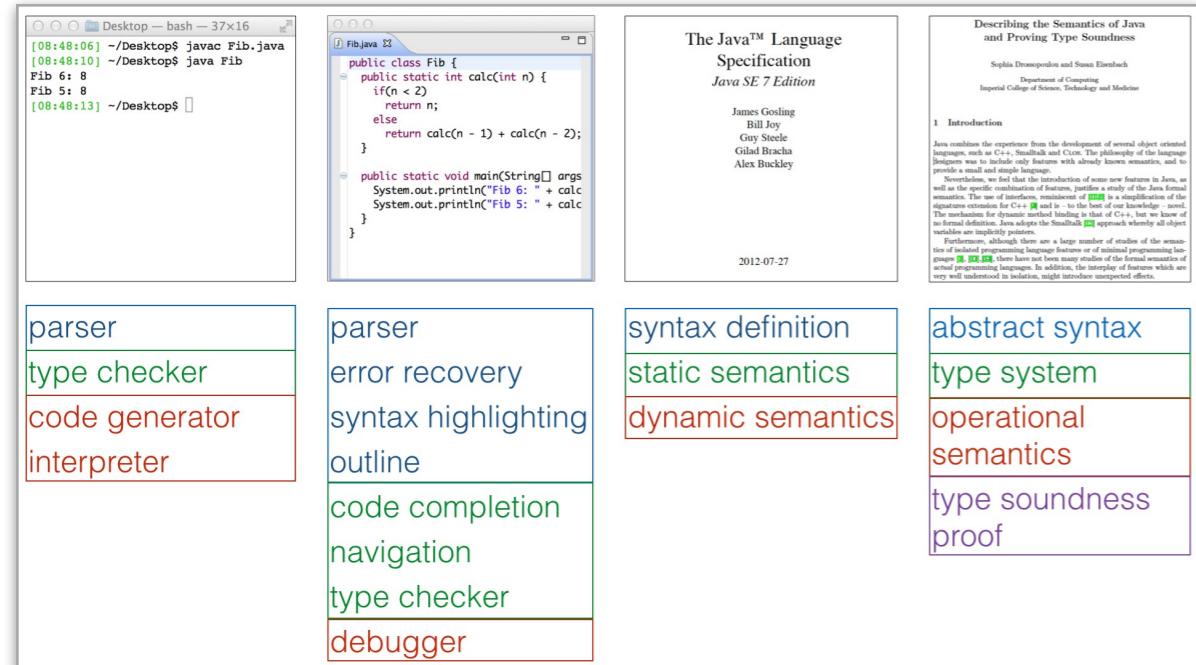


Multi-Purpose Declarative Syntax Definition

```
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



A Language Designer's Workbench

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

