Static Name Resolution

static analysis

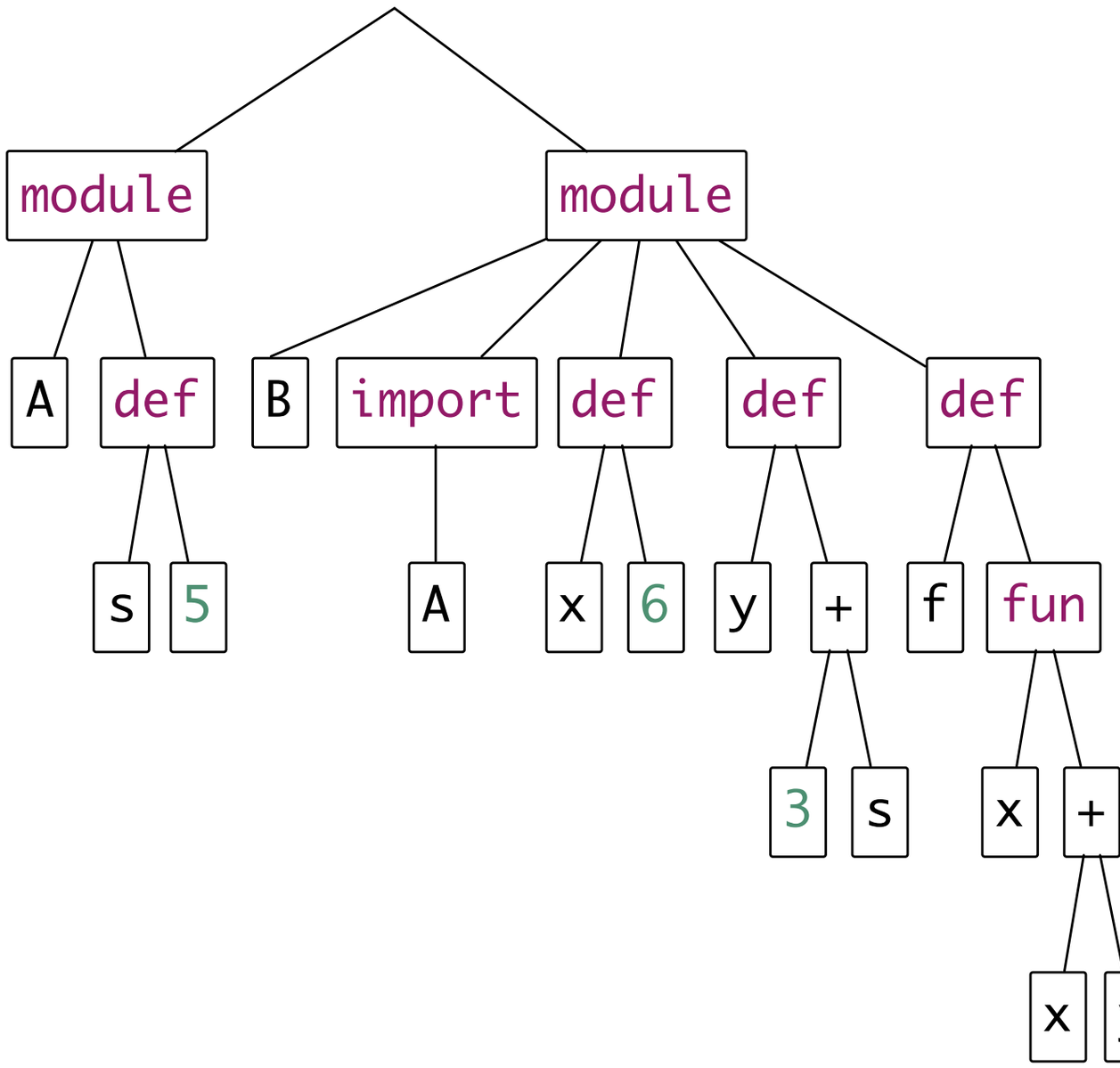
Eelco Visser

Static Name Resolution

```
module A {
    def s = 5
}

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

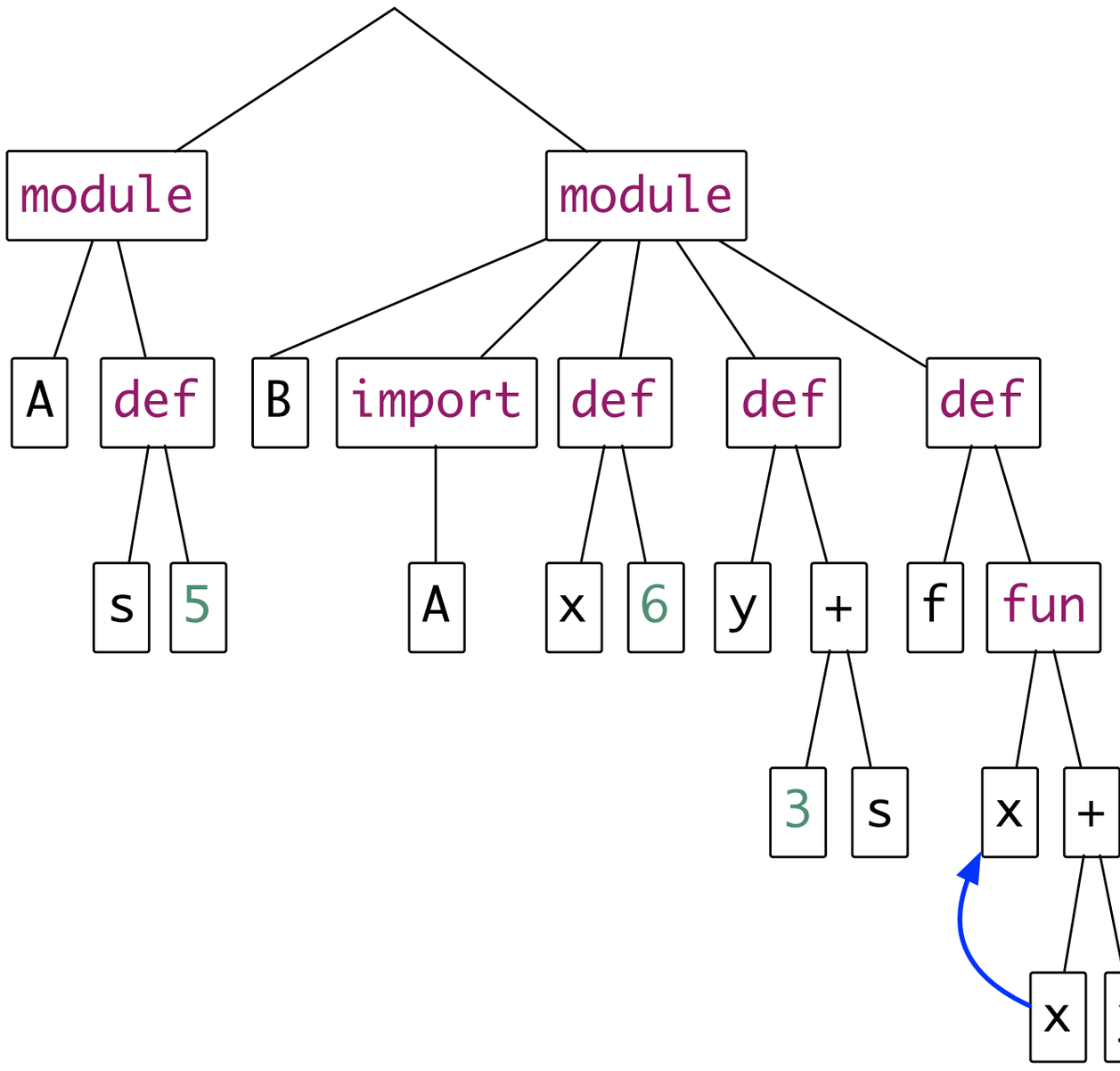
Static Name Resolution

```
module A {
    def s = 5
}

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



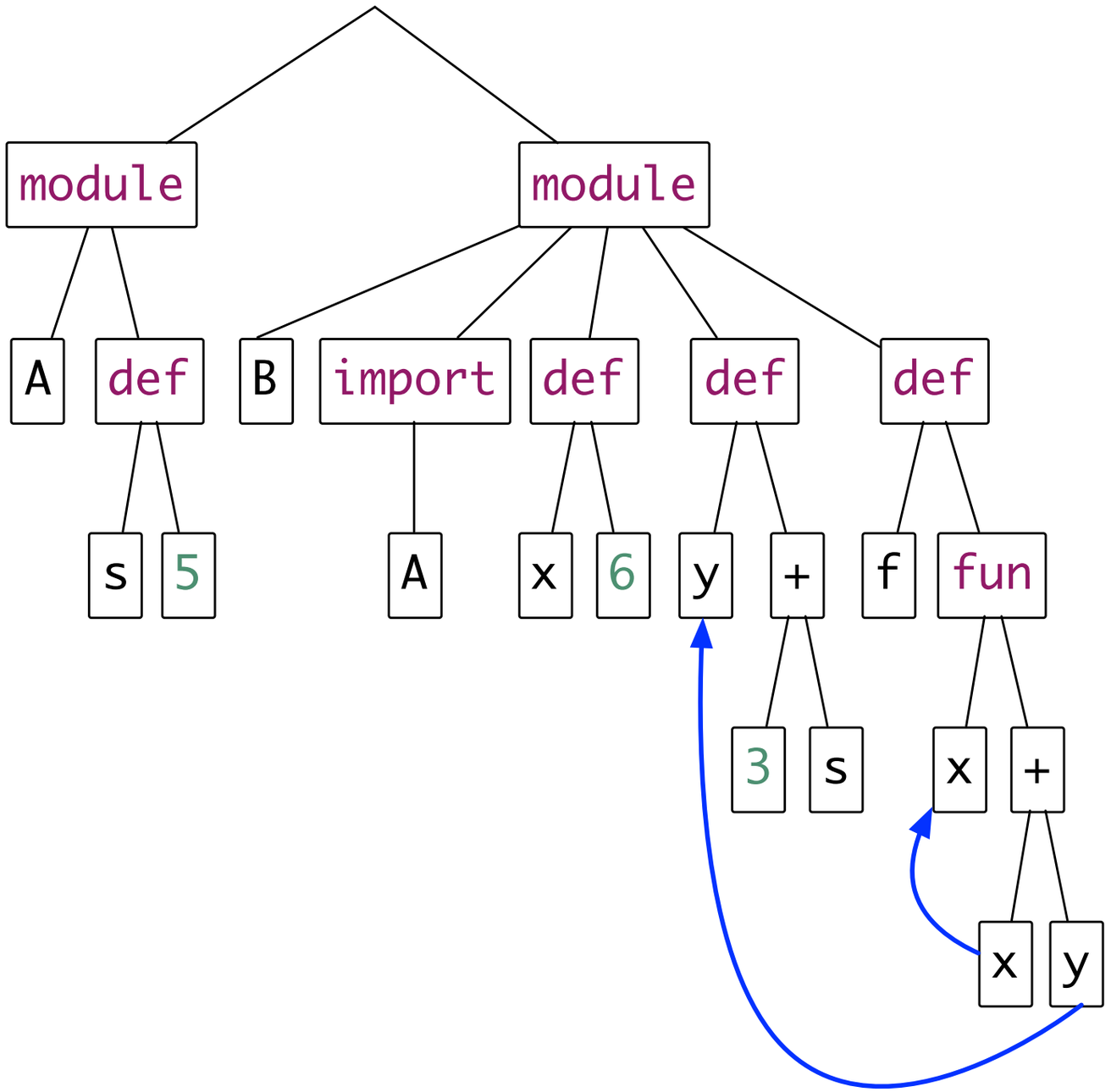
Static Name Resolution

```
module A {
    def s = 5
}

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



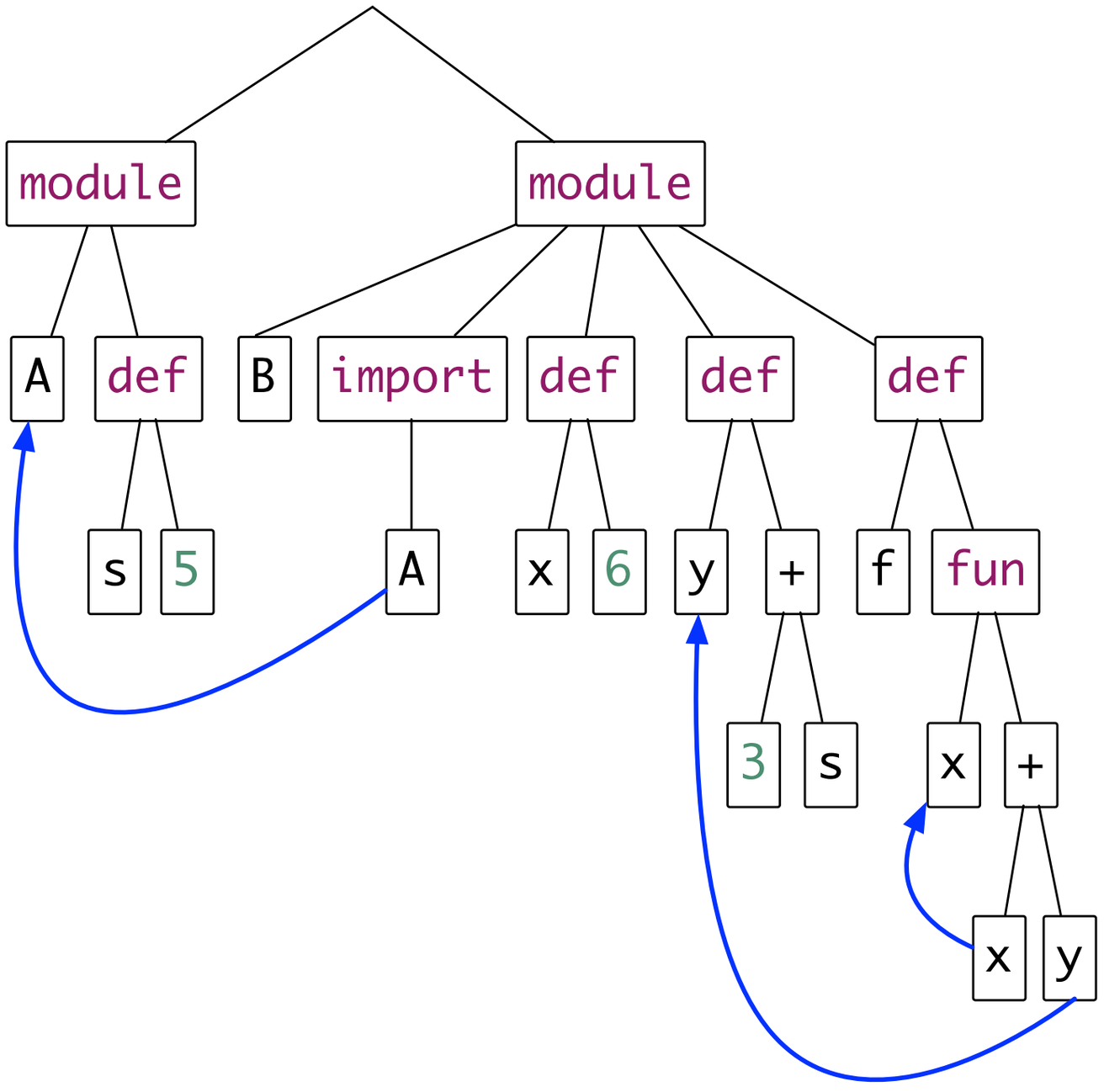
Static Name Resolution

```
module A {
    def s = 5
}

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



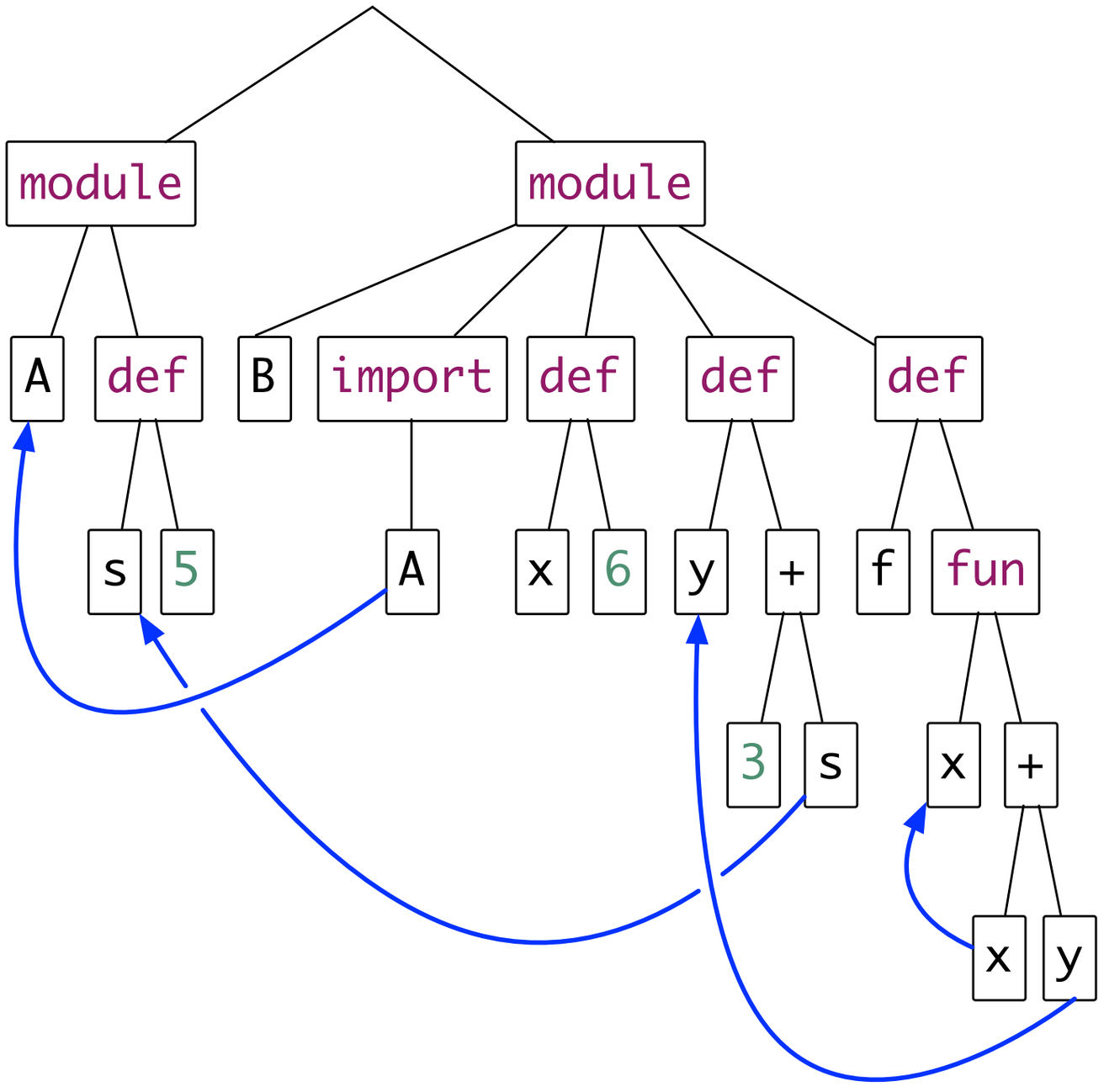
Static Name Resolution

```
module A {
     def s = 5
}

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



Static Name Resolution

```
module A {
    def s = 5
}

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



Based On …

- **Declarative Name Binding and Scope Rules**
  - NaBL name binding language
  - Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser
  - SLE 2012
- **A Language Designer's Workbench**
  - A one-stop-shop for implementation and verification of language designs
  - Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, Gabriël D. P. Konat
  - Onward 2014
- **A Theory of Name Resolution**
  - Pierre Néron, Andrew Tolmach, Eelco Visser, Guido Wachsmuth
  - ESOP 2015
- **A Constraint Language for Static Semantic Analysis based on Scope Graphs**
  - Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, Guido Wachsmuth
  - PEPM 2016

Detour:

Declare Your Syntax

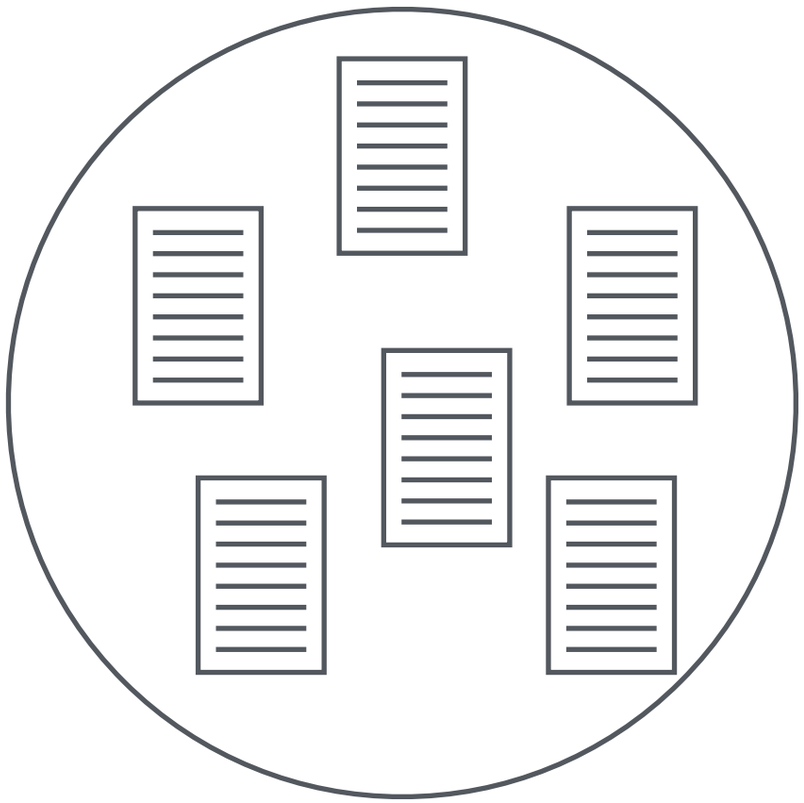
(an analogy for formalizing declarative name binding)

Pure and declarative syntax definition: paradise lost and regained

Lennart C. L. Kats, Eelco Visser, Guido Wachsmuth

Onward 2010

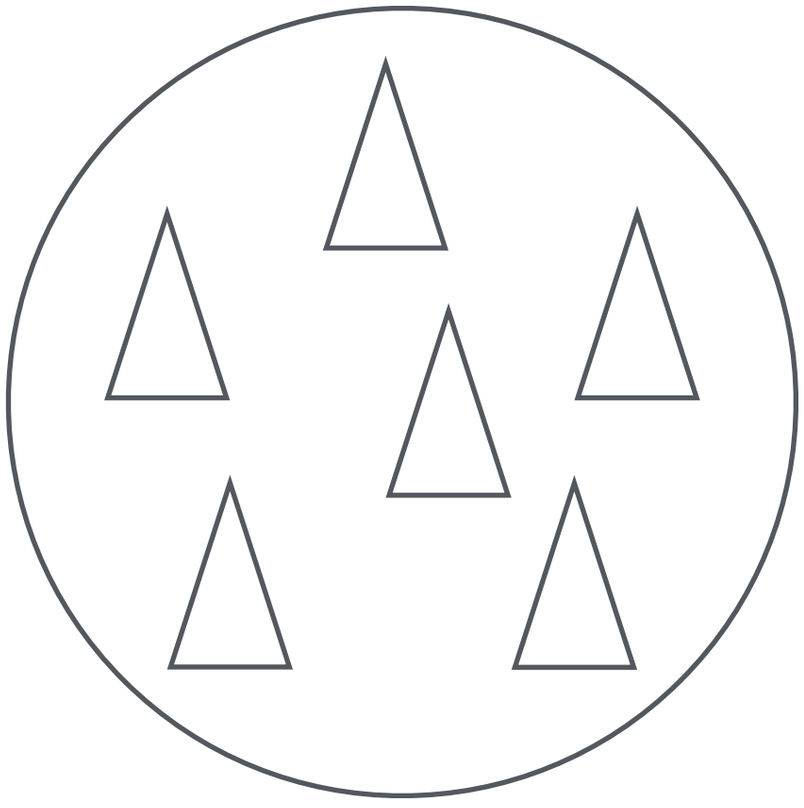
Language = Set of Sentences

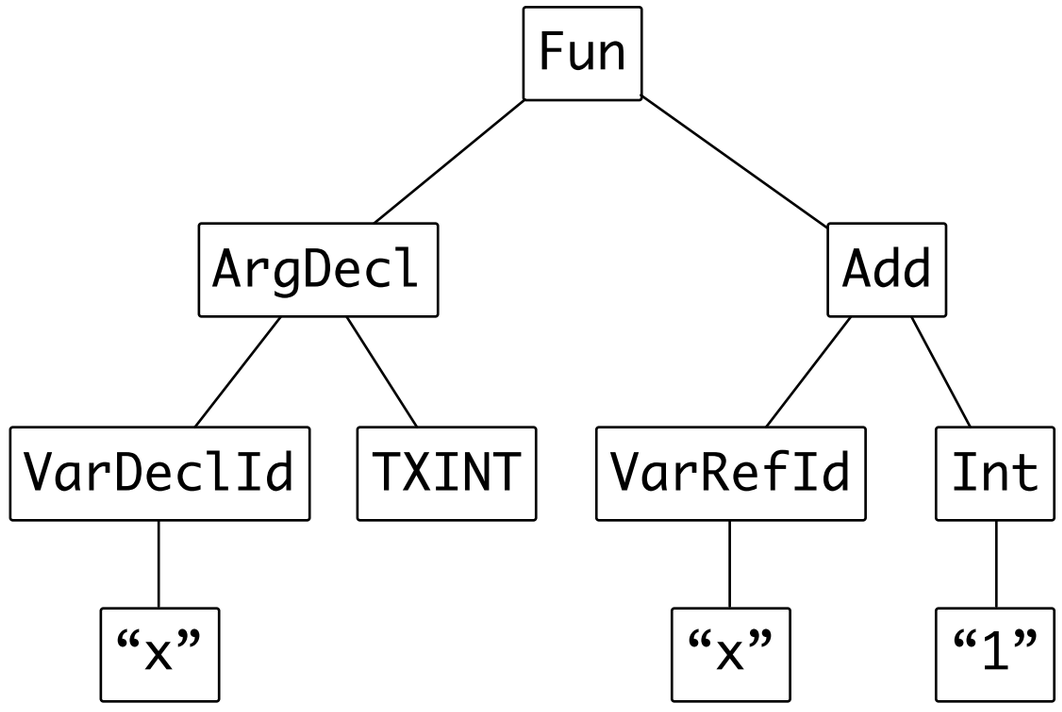


```
fun (x : Int) { x + 1 }
```

Text is a convenient interface for writing and reading programs

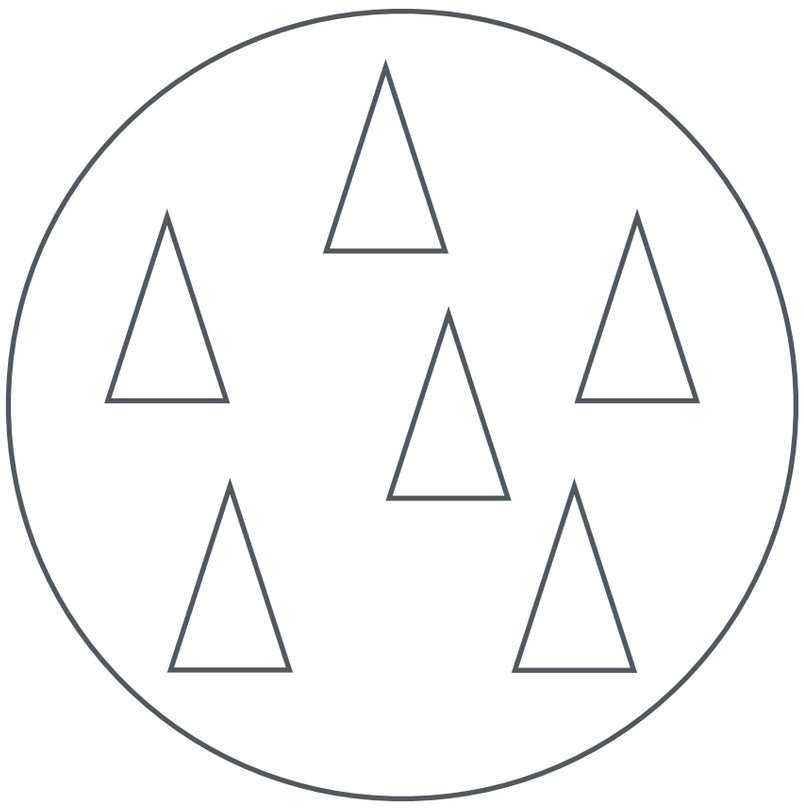
Language = Set of Trees





Tree is a convenient interface for transforming programs

Tree Transformation



Syntactic

coloring

outline view

completion

Semantic

transform

translate

eval

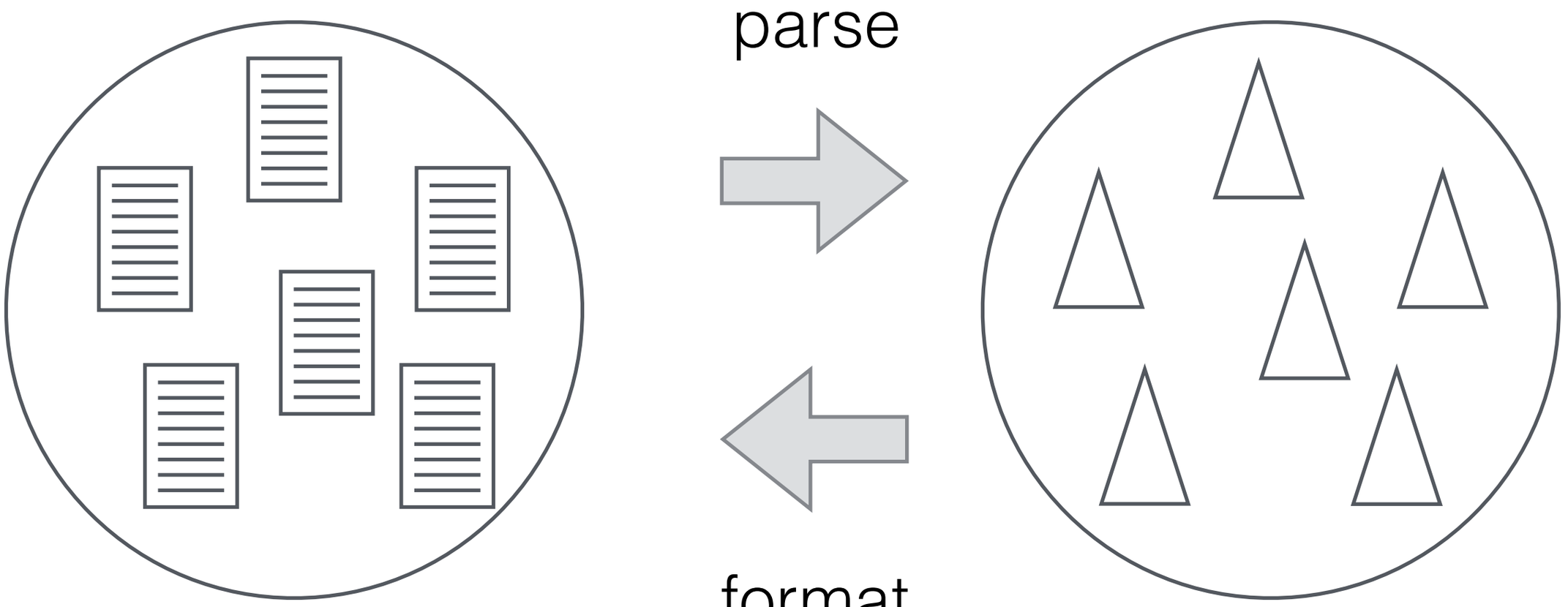
analyze

refactor

type check

Tree is a convenient interface for transforming programs

Language = Sentences *and* Trees



different representations convenient for different purposes

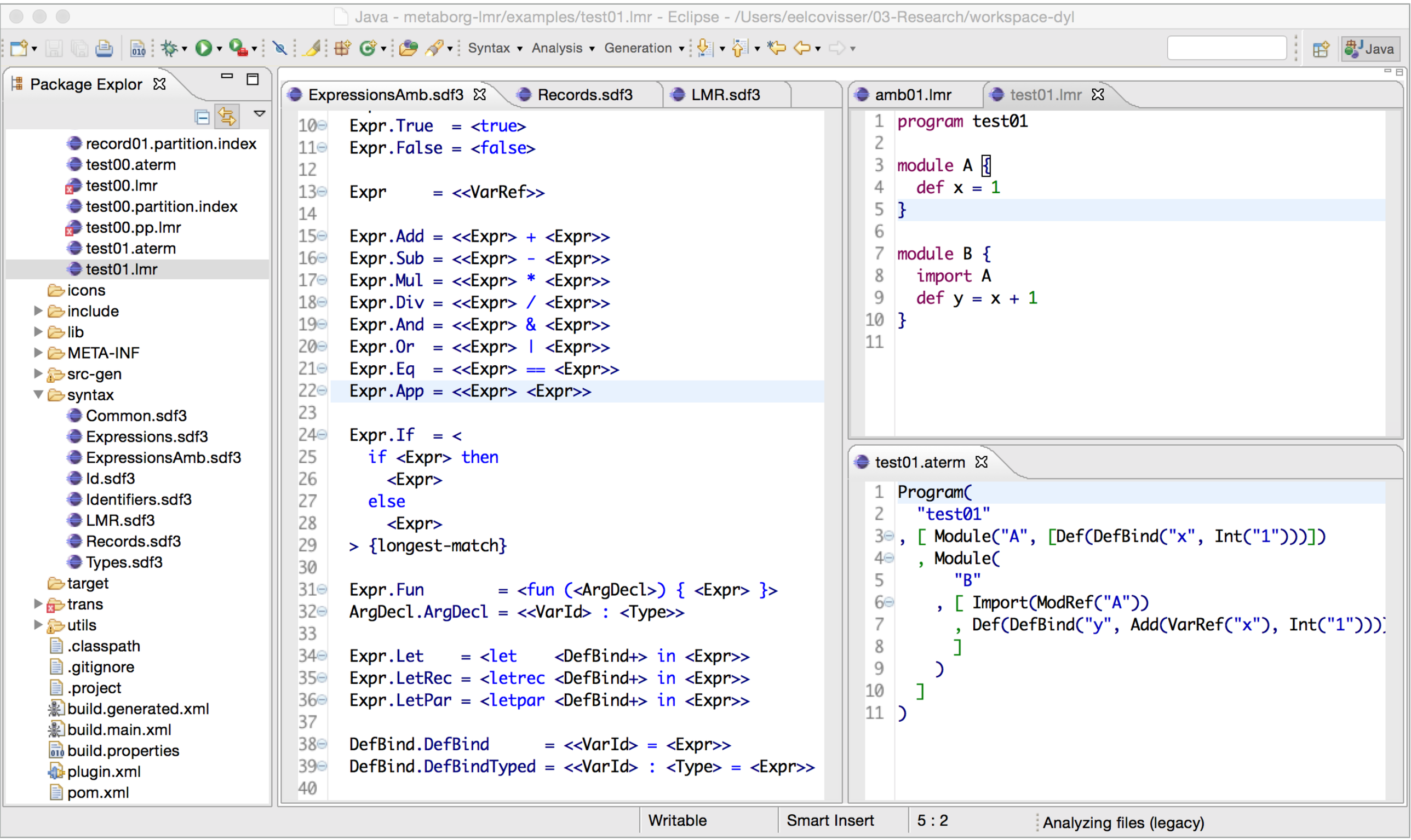
SDF3 defines Trees *and* Sentences

```
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

```
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 | test01.lmr

```
1  a + b * x - 1
```

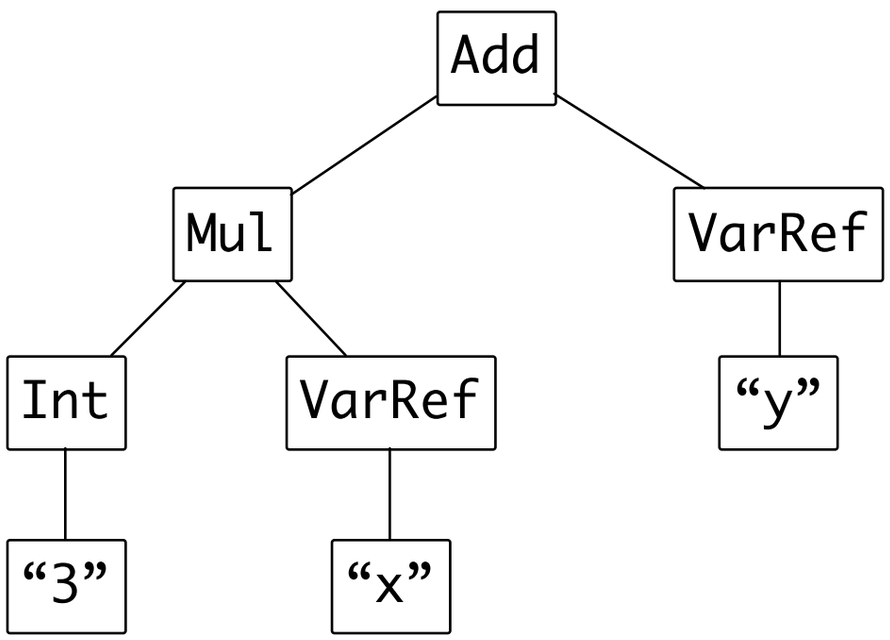
test01.aterm | amb01.aterm

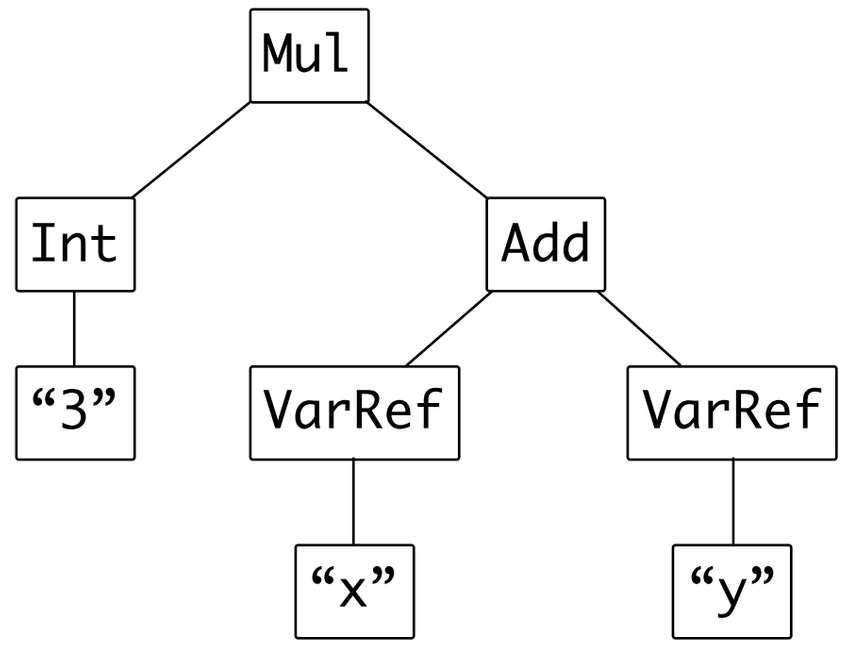
```
1  amb(
2    [ amb(
3        [ Sub(
4            amb(
5              [ Mul(Add(VarRef("a"), VarRef("b")), VarRef("x"))
6              , Add(VarRef("a"), Mul(VarRef("b"), VarRef("x")))
7              ]
8            )
9          , Int("1")
10         )
11       , Add(
12           VarRef("a")
13  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

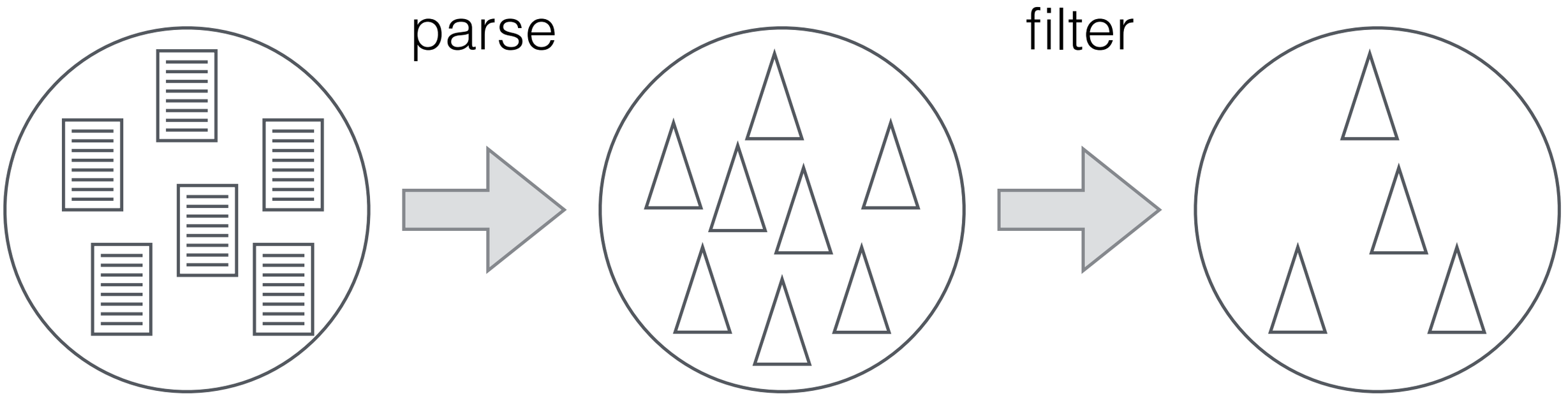
```
3 * x + y
```






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

Declarative Disambiguation

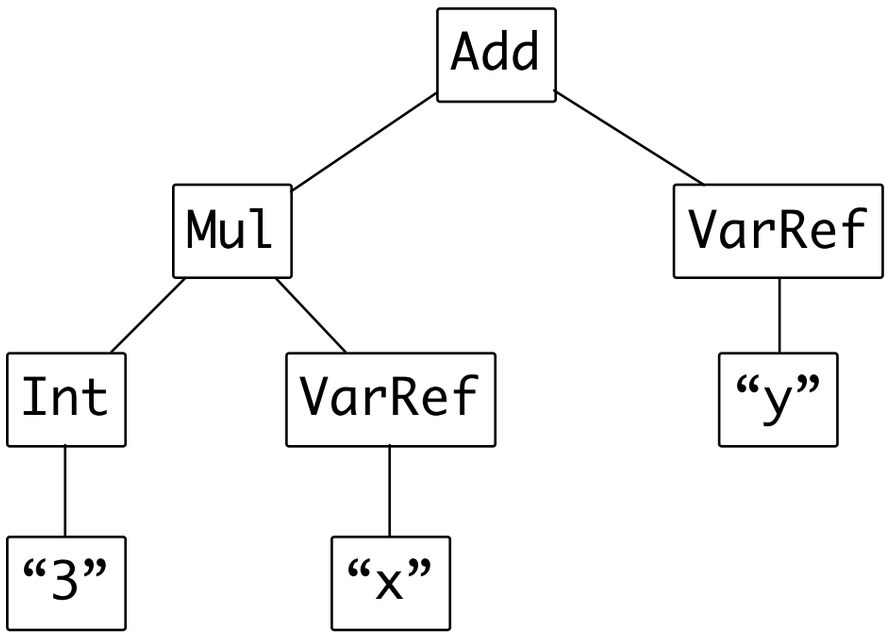


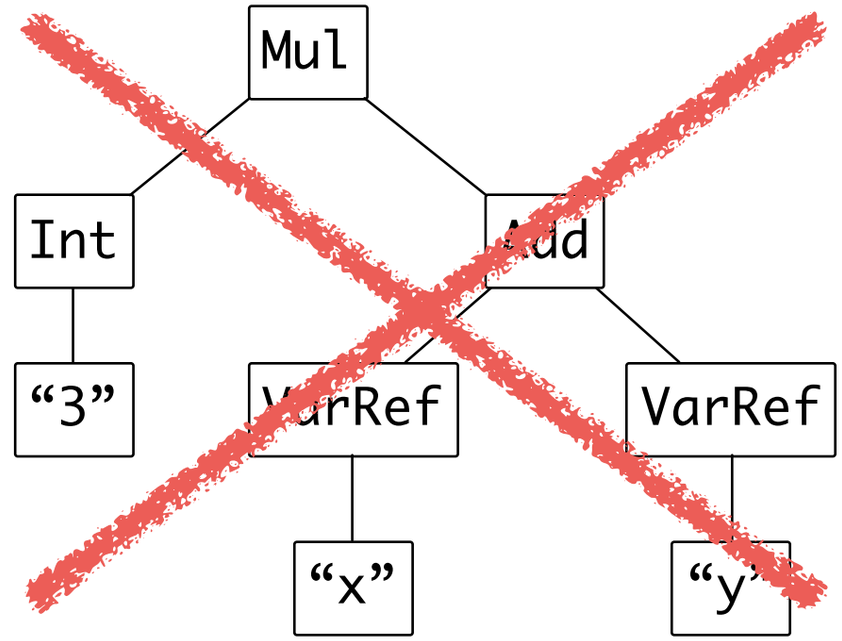
Disambiguation Filters

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. (SLE 2013, Onward 2015)

Declarative Syntax Definition

- **Representation: (Abstract Syntax) Trees**
  - Standardized representation for structure of programs
  - Basis for syntactic and semantic operations
- **Formalism: Syntax Definition**
  - Productions + Constructors + Templates + Disambiguation
  - Language-specific rules: structure of each language construct
- **Language-Independent Interpretation**
  - Well-formedness of abstract syntax trees
    - provides declarative correctness criterion for parsing
  - Parsing algorithm
    - No need to understand parsing algorithm
    - Debugging in terms of representation
  - Formatting based on layout hints in grammar

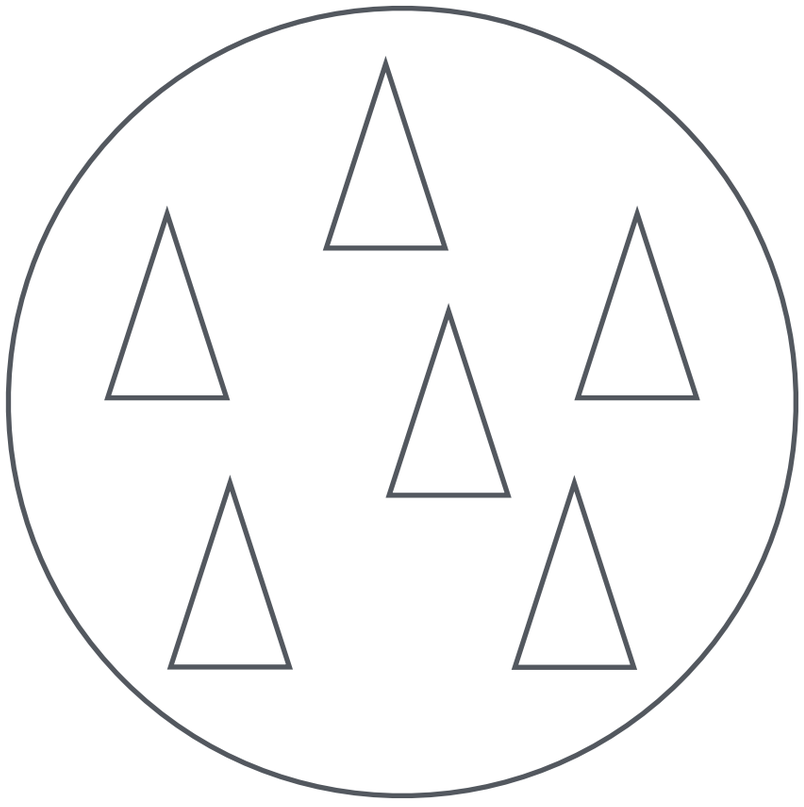
Declare Your Names

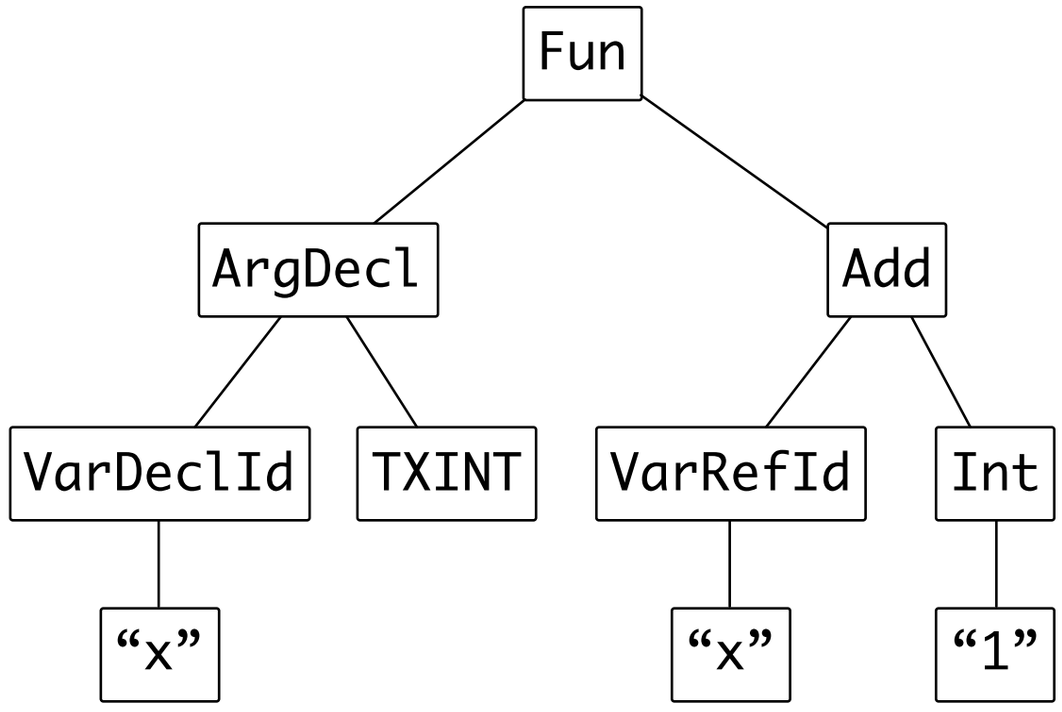
A Theory of Name Resolution

Pierre Néron, Andrew Tolmach, Eelco Visser, Guido Wachsmuth

ESOP 2015

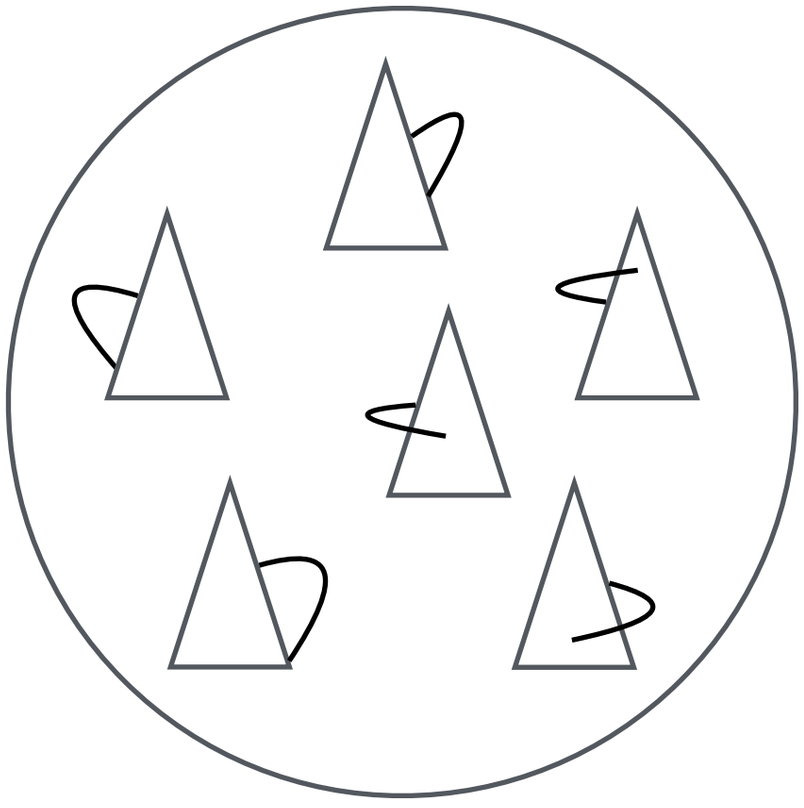
Language = Set of Trees

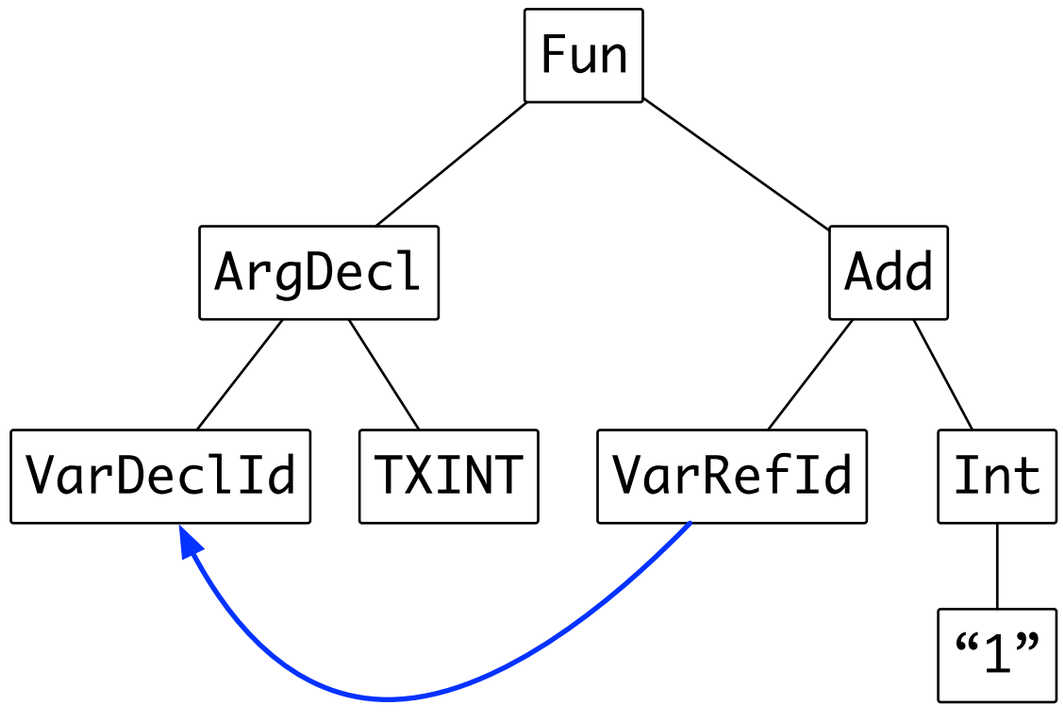




Tree is a convenient interface for transforming programs

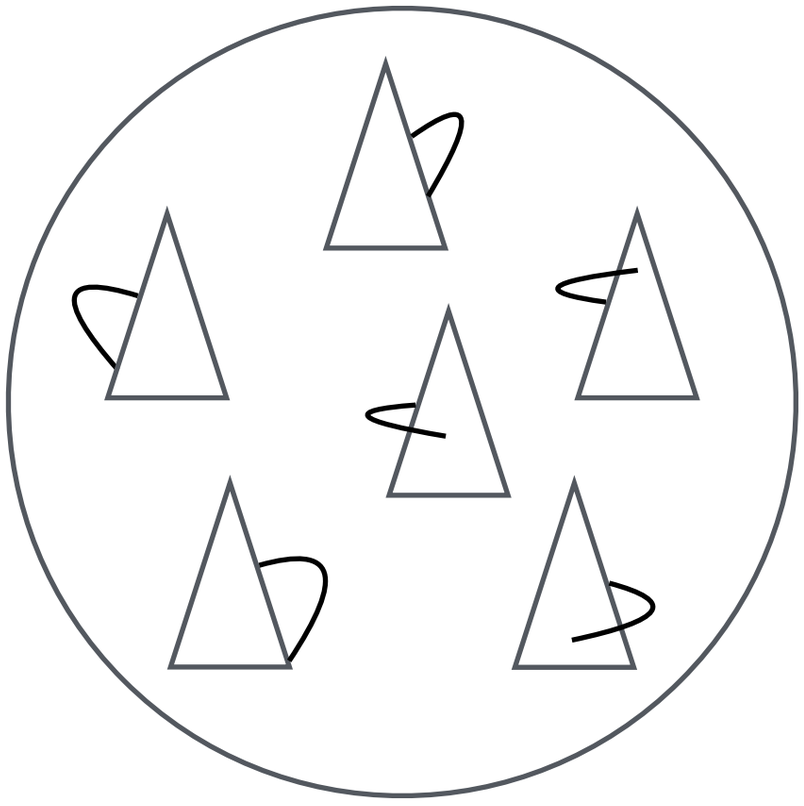
Language = Set of *Graphs*

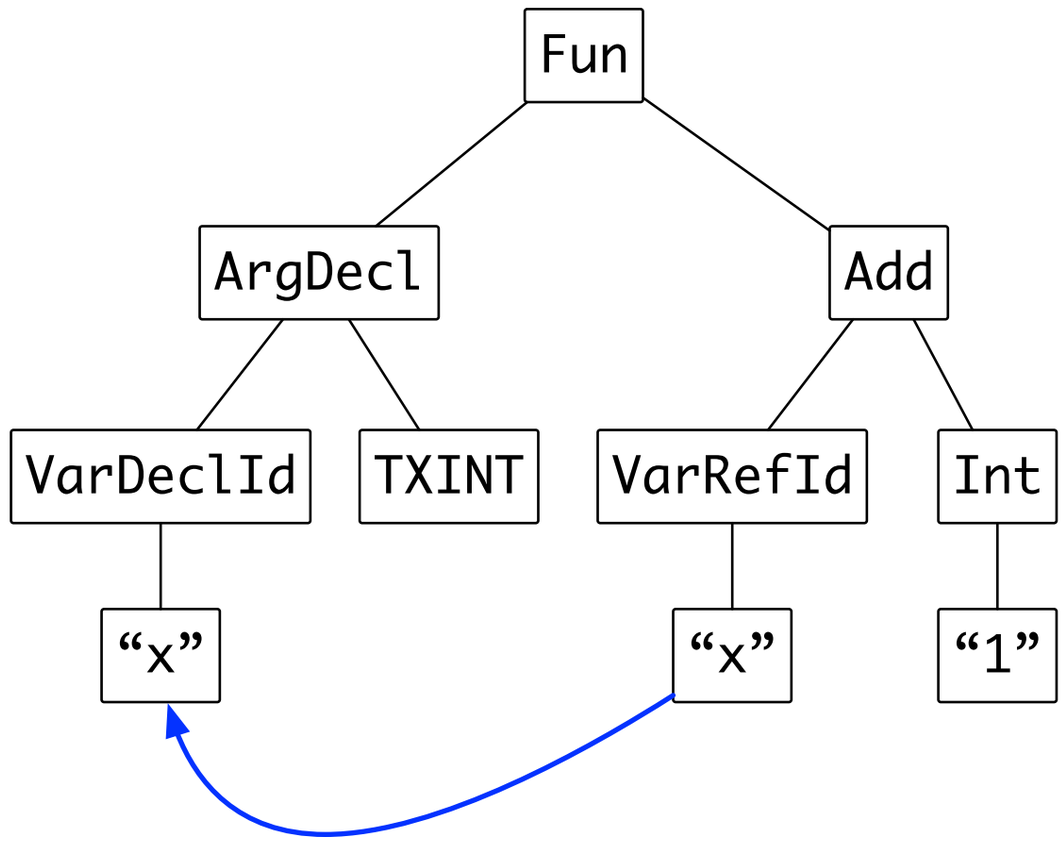




Edges from references to declarations

Language = Set of *Graphs*





Names are placeholders for edges in linear / tree representation

Name Resolution is Pervasive

- **Used in many different language artifacts**
  - compiler
  - interpreter
  - semantics
  - IDE
  - refactoring
- **Binding rules encoded in many different and ad-hoc ways**
  - symbol tables
  - environments
  - substitutions
- **No standard approach to formalization**
  - there is no BNF for name binding
- **No reuse of binding rules between artifacts**
  - how do we know substitution respects binding rules?

NaBL Name Binding Language

```
binding rules // variables

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

  Let(bs, e) :
    scopes Variable

Bind(t, x, e) :
  defines Variable x of type t

Var(x) :
  refers to Variable x
```

Declarative specification

Abstracts from implementation

Incremental name resolution

But:

How to explain it to Coq?

What is the semantics of NaBL?

Declarative Name Binding and Scope Rules

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser

SLE 2012

NaBL Name Binding Language

```
binding rules // classes

  Class(c, _, _, _) :
   defines Class c of type ClassT(c)
   scopes Field, Method, Variable

  Extends(c) :
   imports Field, Method from Class c

ClassT(c) :
 refers to Class c

New(c) :
 refers to Class c
```

Especially:

What is the semantics of imports?

Declarative Name Binding and Scope Rules

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser

SLE 2012

A Theory of Name Resolution

- **Representation: Scope Graphs**
  - Standardized representation for lexical scoping structure of programs
  - Path in scope graph relates reference to declaration
  - Basis for syntactic and semantic operations
  - Supports ambiguous / erroneous programs
- **Formalism: Name Binding Specification**
  - References + Declarations + Scopes + Reachability + Visibility
  - Language-specific rules: mapping constructs to scope graph
- **Language-Independent Interpretation**
  - Resolution calculus
    - Correctness of path with respect to scope graph
    - Separation of reachability and visibility (disambiguation)
  - Name resolution algorithm
    - sound wrt calculus
  - Transformation
    - Alpha equivalence, substitution, refactoring, …

Outline

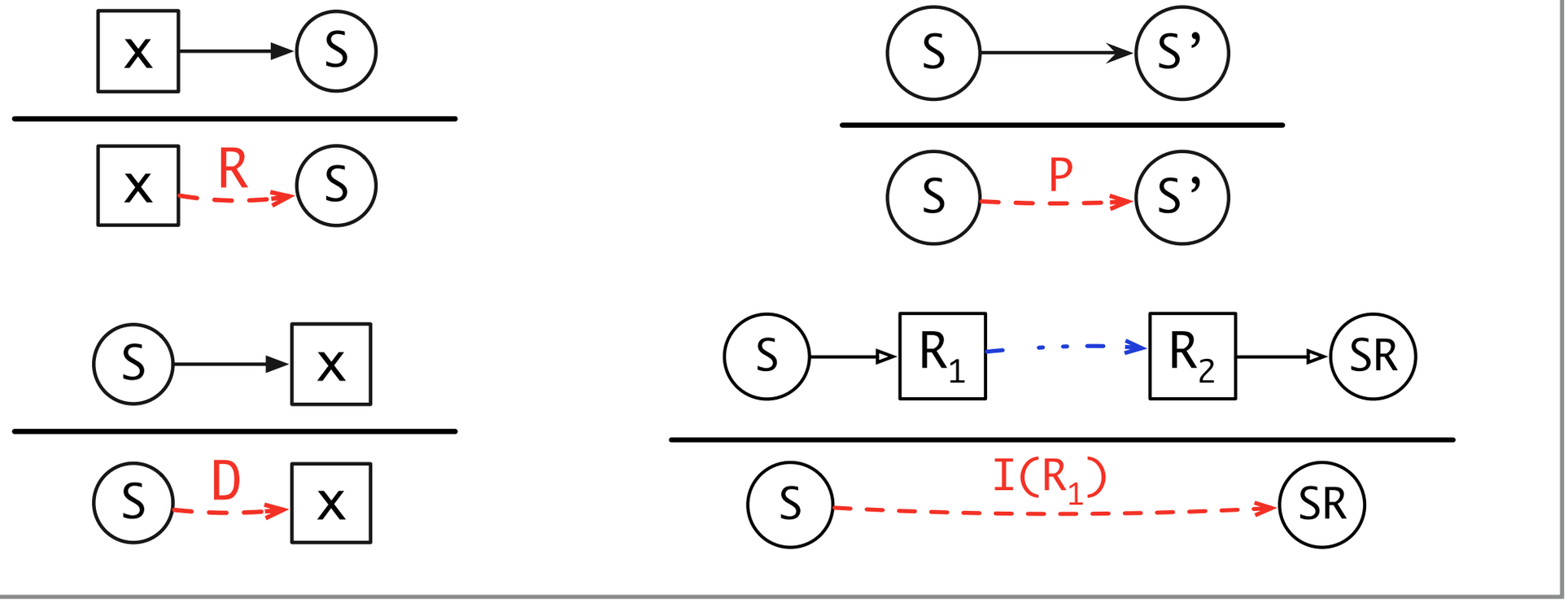
- **Reachability**
  - References, Declarations, Scopes, Resolution Paths
  - Lexical, Imports, Qualified Names
- **Visibility**
  - Ambiguous resolutions
  - Disambiguation: path specificity, path wellformedness
- **Examples**
  - let, def-before-use, inheritance, packages, imports, namespaces
- **Formal framework**
  - Generalizing reachability and visibility through labeled scope edges
  - Resolution algorithm
  - Alpha equivalence
- **Names and Types**
  - Type-dependent name resolution
- **Constraint Language**

Reachability

(Slides by Pierre Néron)

A Calculus for Name Resolution

Scopes, References, Declarations, Parents, Imports



Path in scope graph connects reference to declaration

Simple Scopes

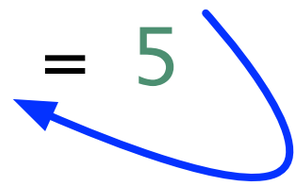
```
def y = x + 1
def x = 5
```

Simple Scopes

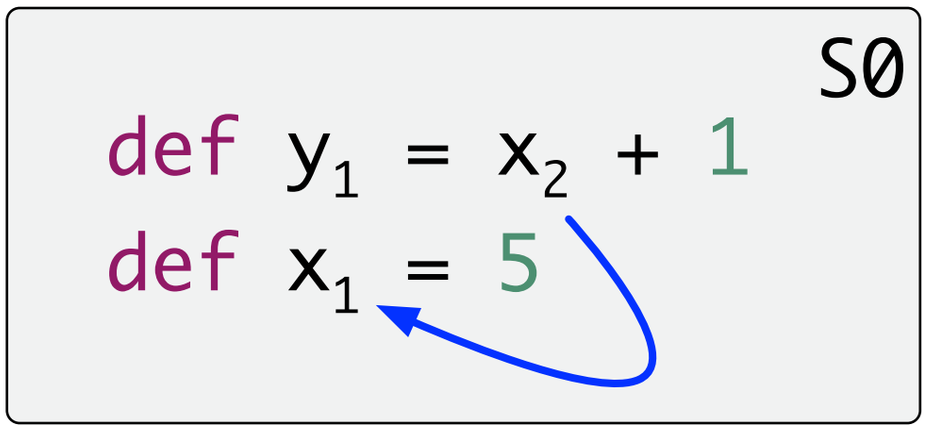
```
def y1 = x2 + 1
def x1 = 5
```

Simple Scopes

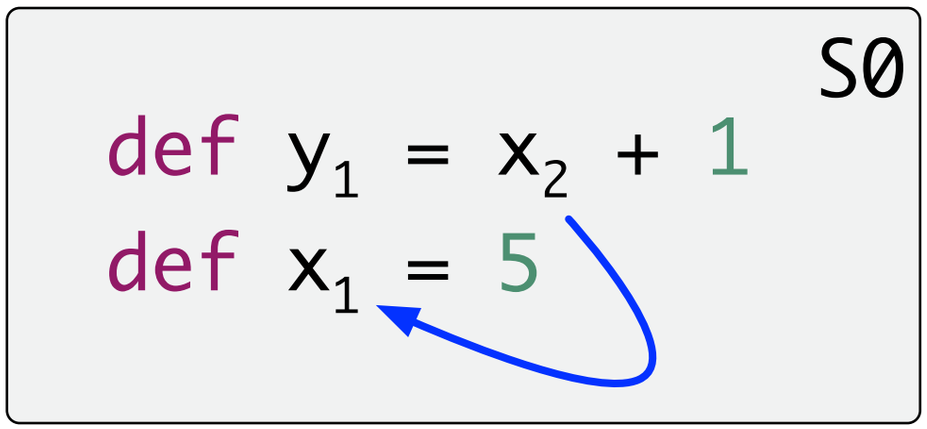
```
def y₁ = x₂ + 1
def x₁ = 5
```

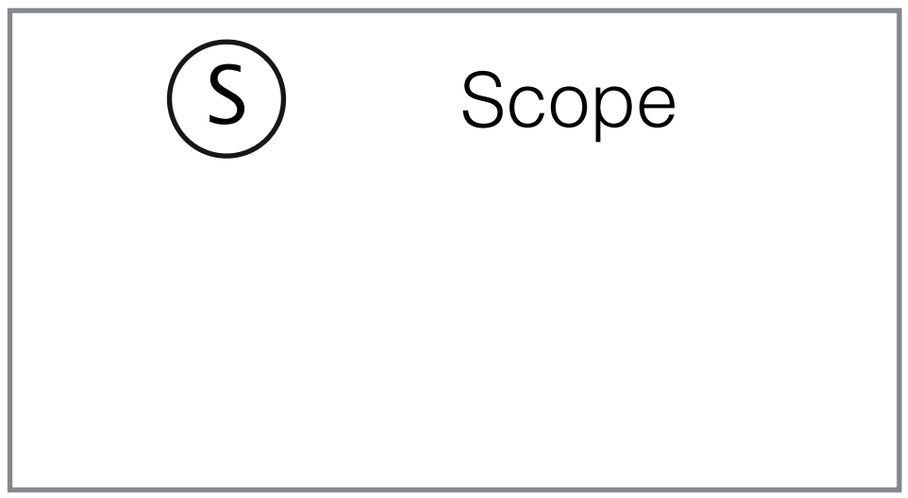


Simple Scopes



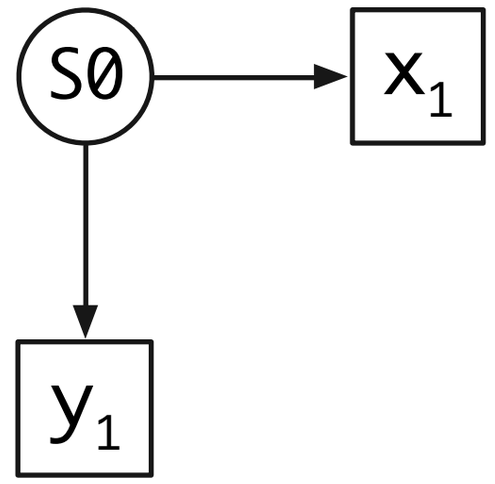
Simple Scopes

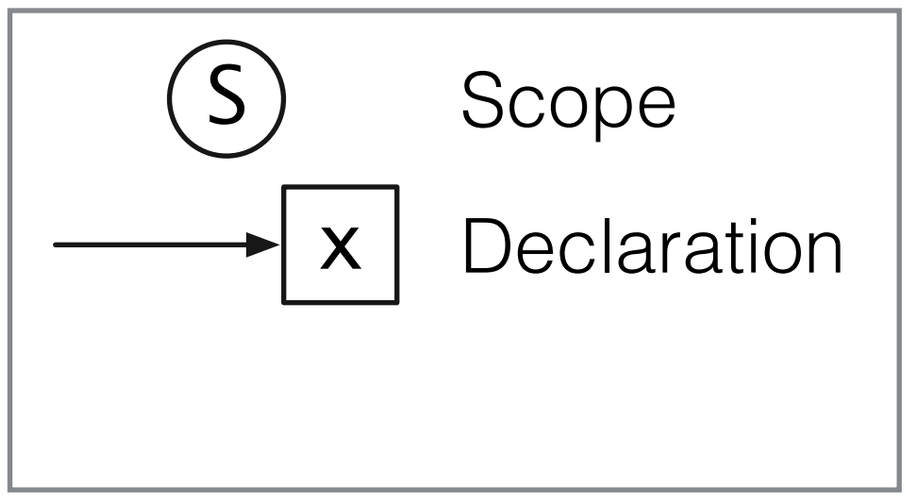




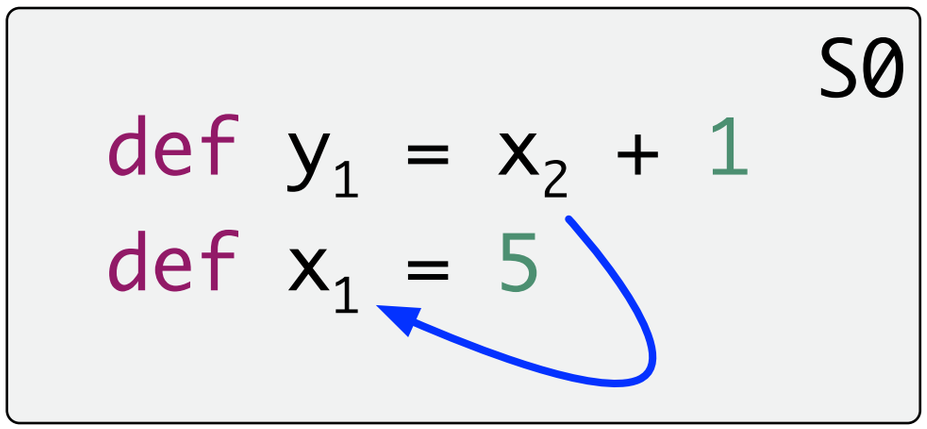
Simple Scopes

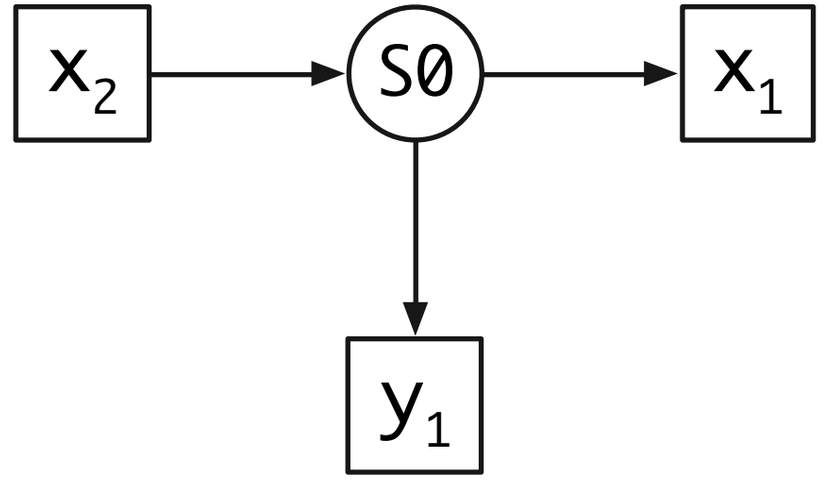
```
                                    S0
def y1 = x2 + 1
def x1 = 5
```

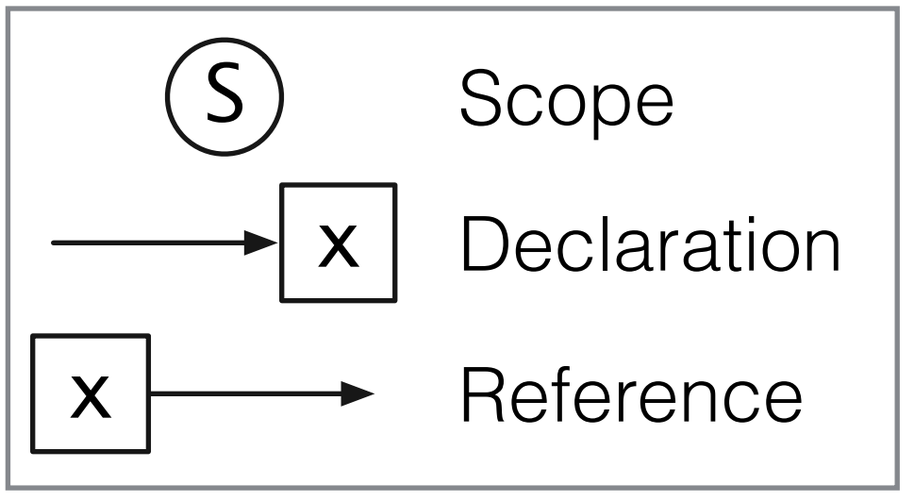




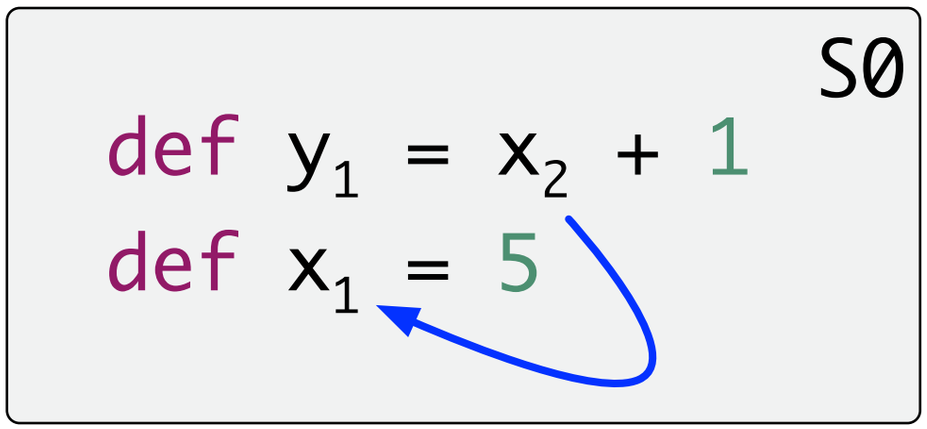
Simple Scopes

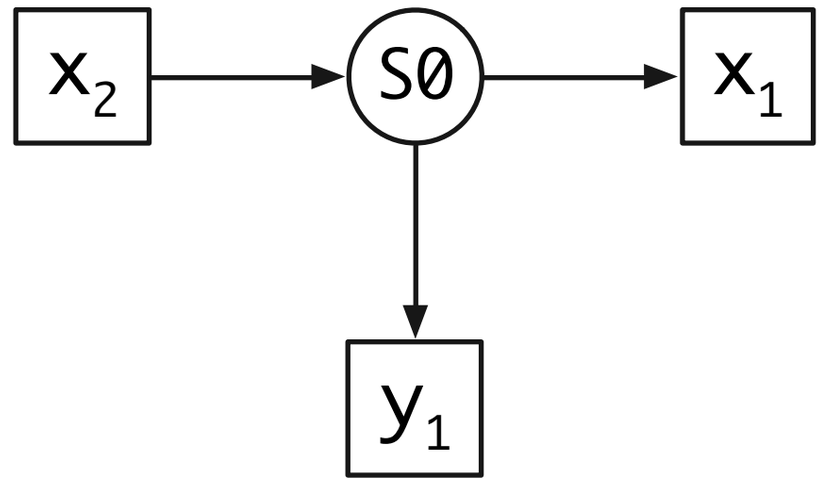


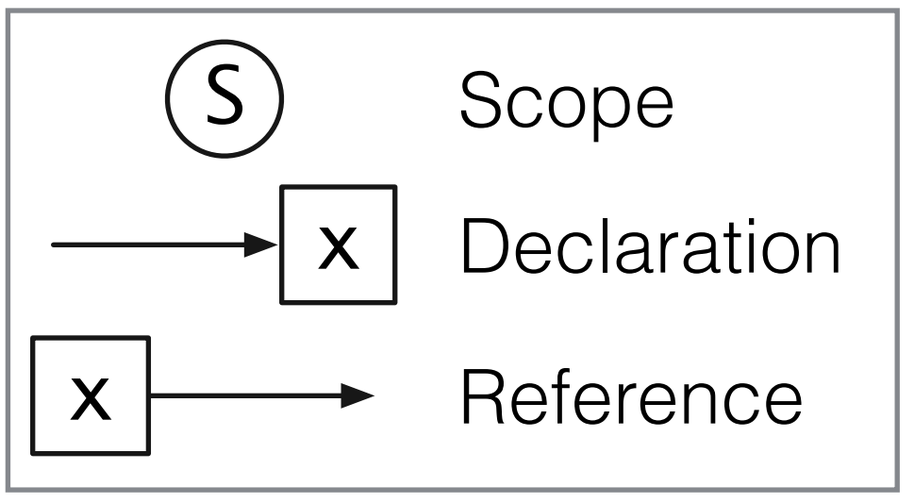


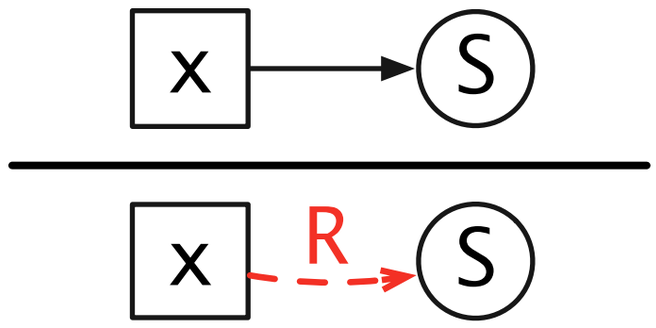


Simple Scopes



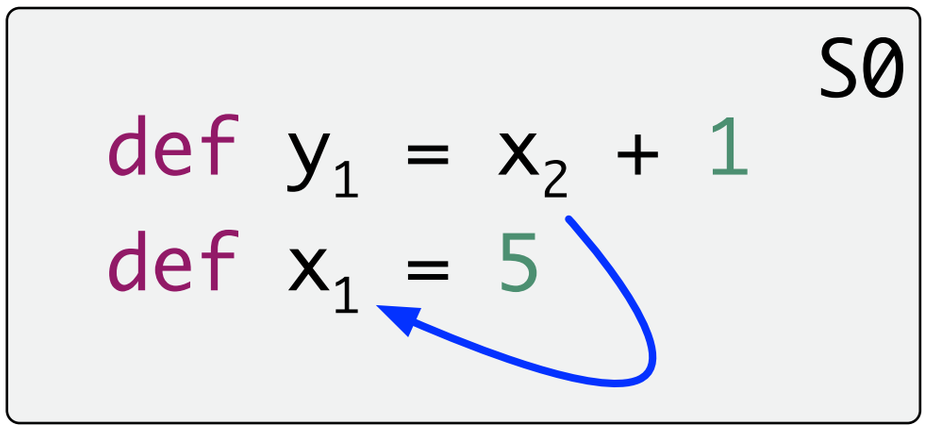


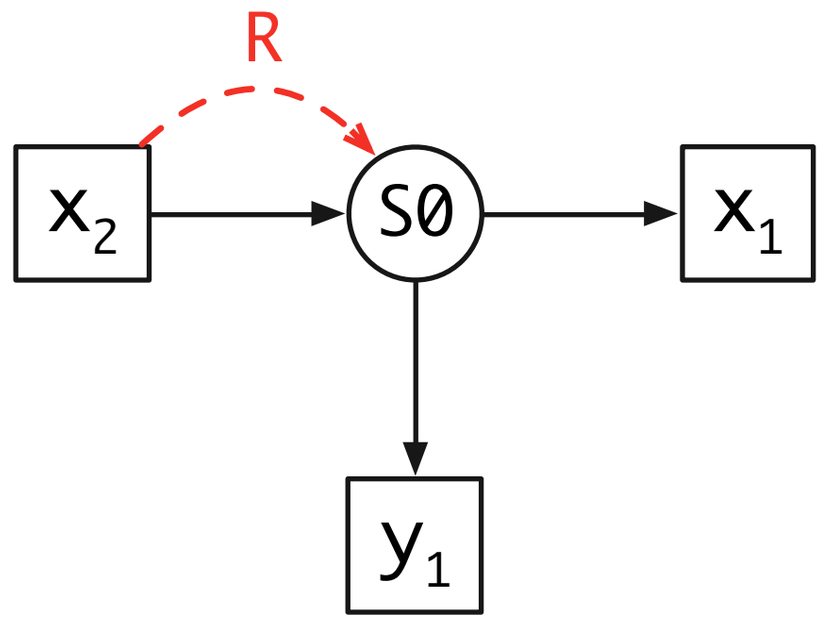


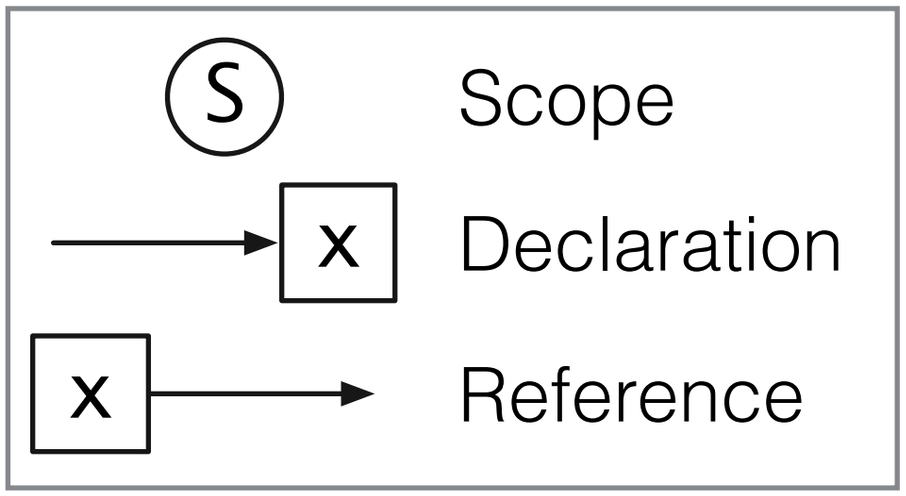


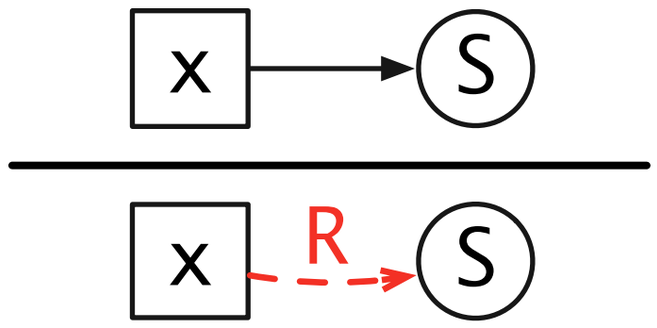
Reference Step

Simple Scopes



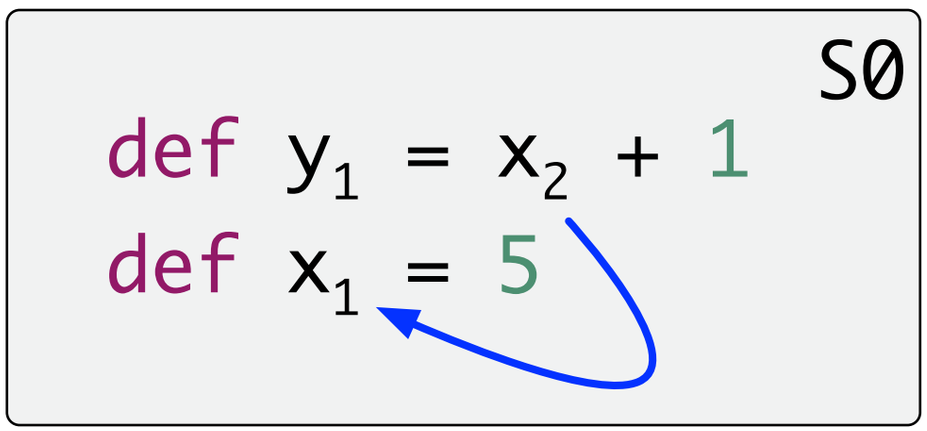


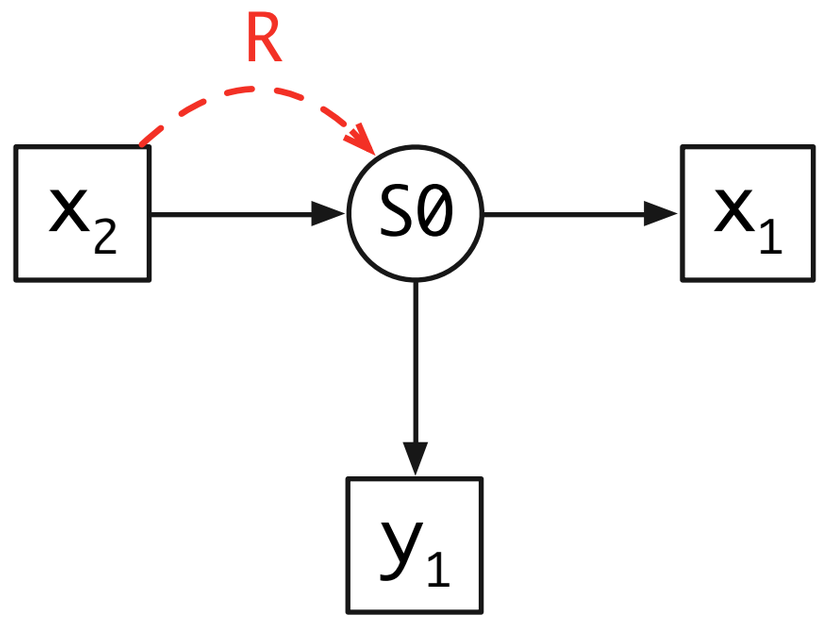


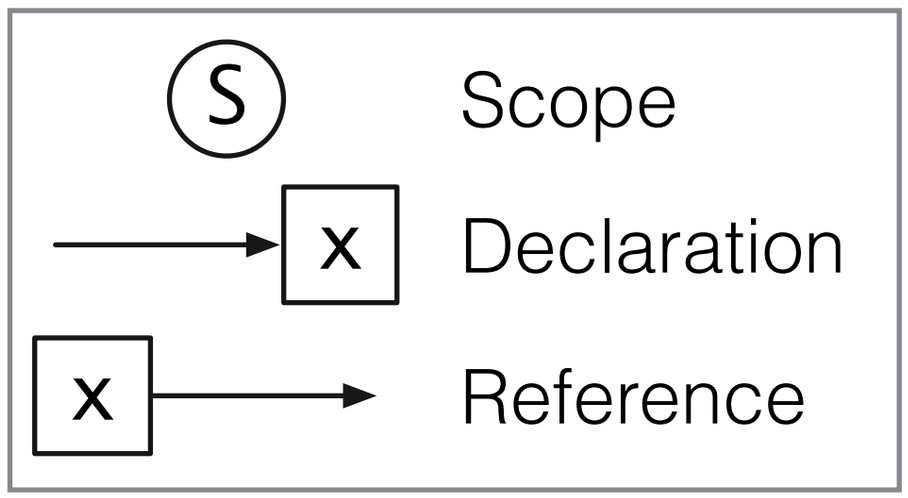


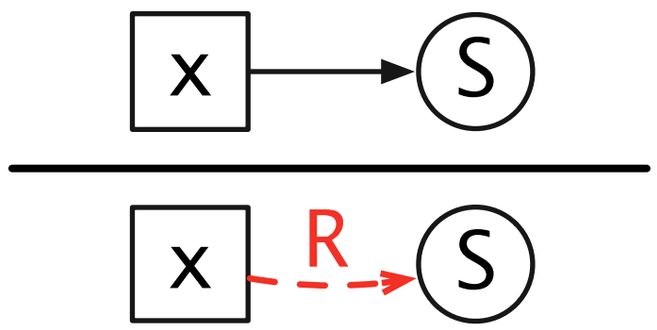
Reference Step

Simple Scopes

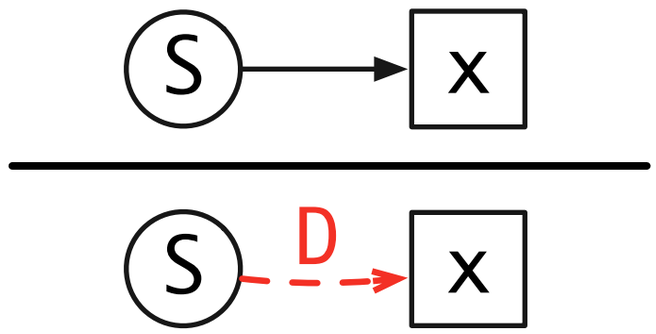






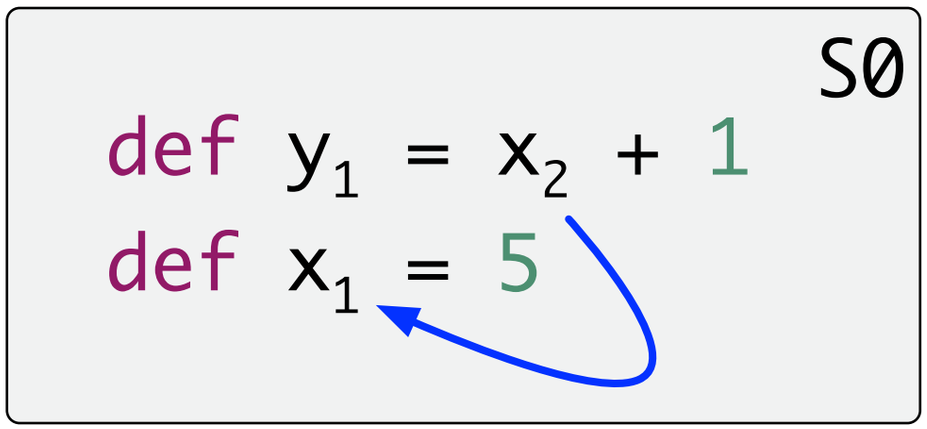


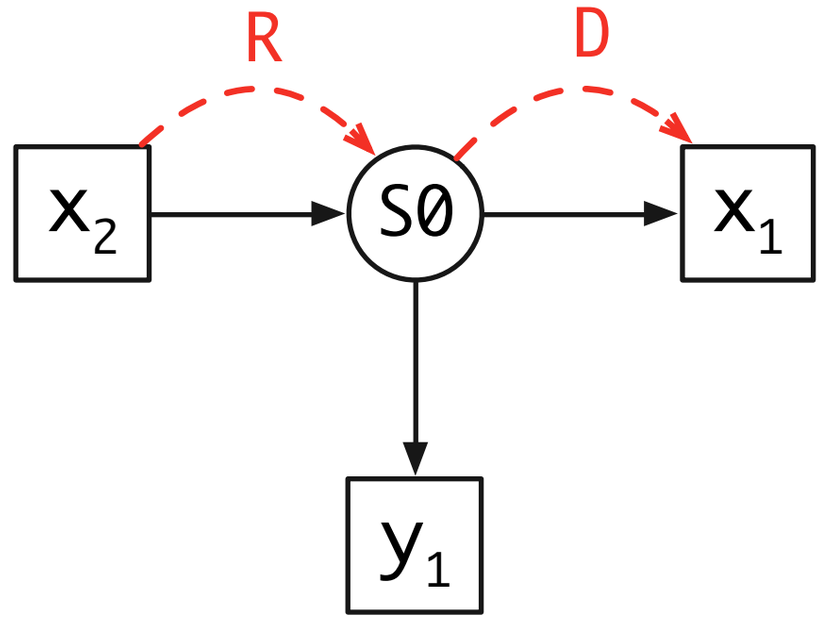
Reference Step

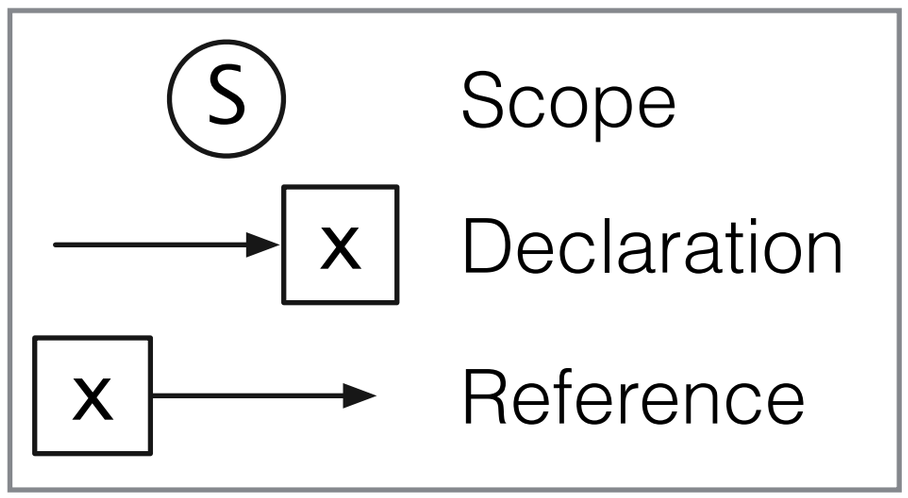


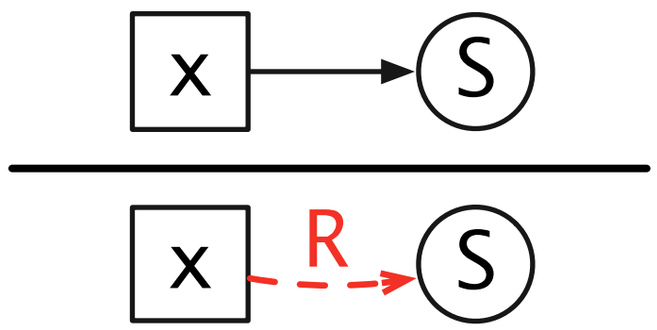
Declaration Step

Simple Scopes

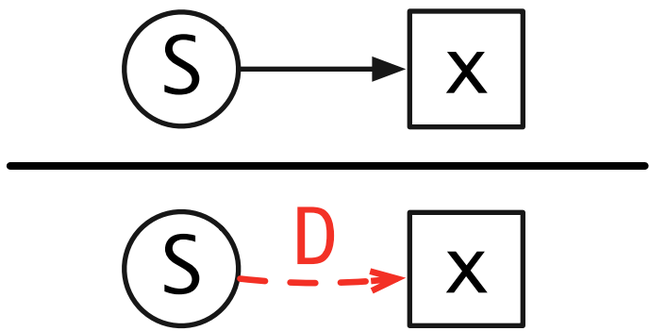








Reference Step



Declaration Step

Lexical Scoping

```
def x1 = z2 5

def z1 =
  fun y1 {
    x2 + y2
  }
```

Lexical Scoping

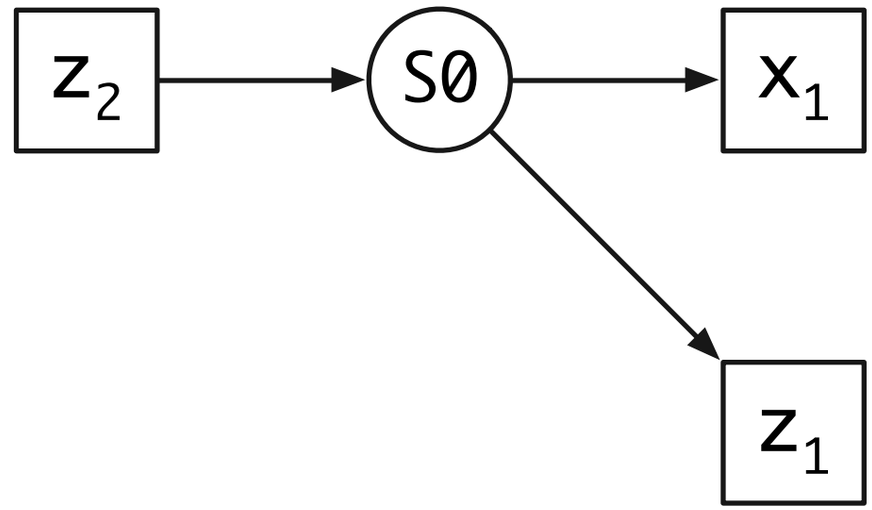
```
                                S0
def x1 = z2 5


def z1 =
  fun y1 {
    x2 + y2
  }
```

Lexical Scoping

```
                                S0
def x1 = z2 5

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

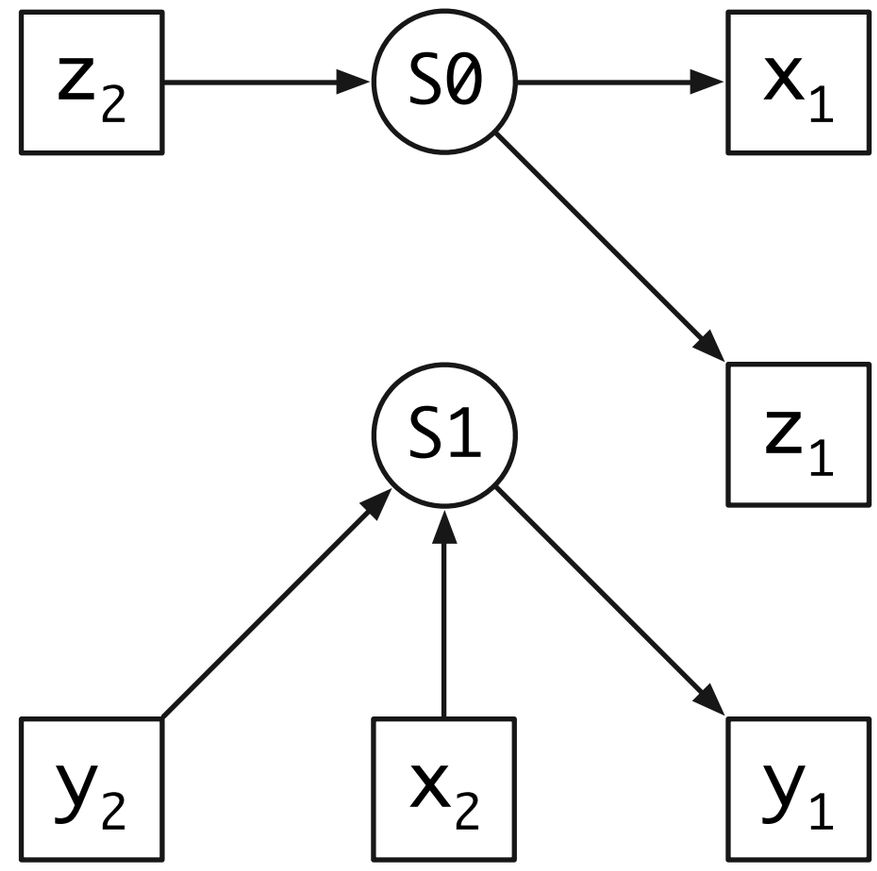
Lexical Scoping

```
                                  S0
def x1 = z2 5

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



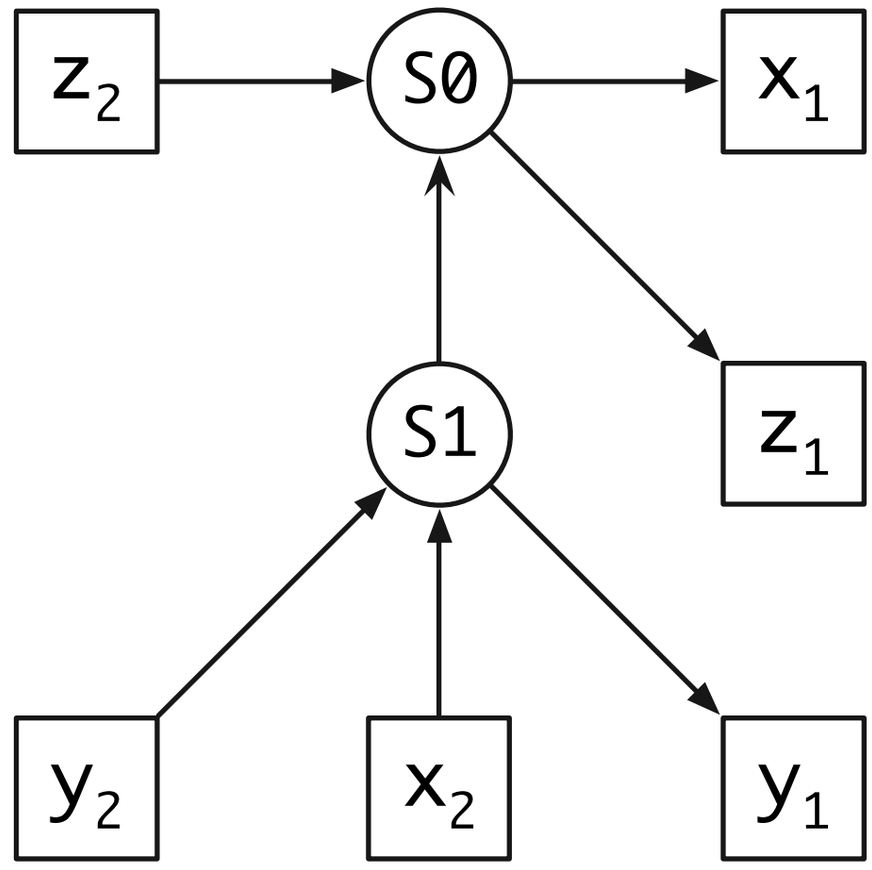
Lexical Scoping

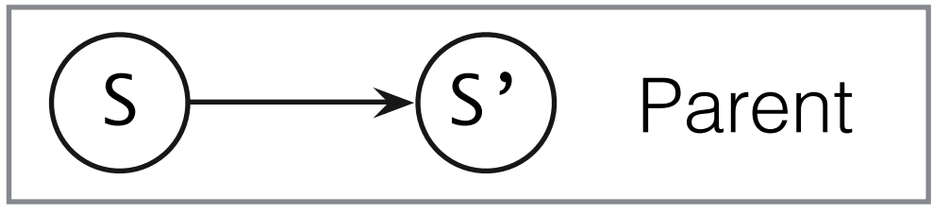
```
                                  S0
def x1 = z2 5

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



Lexical Scoping

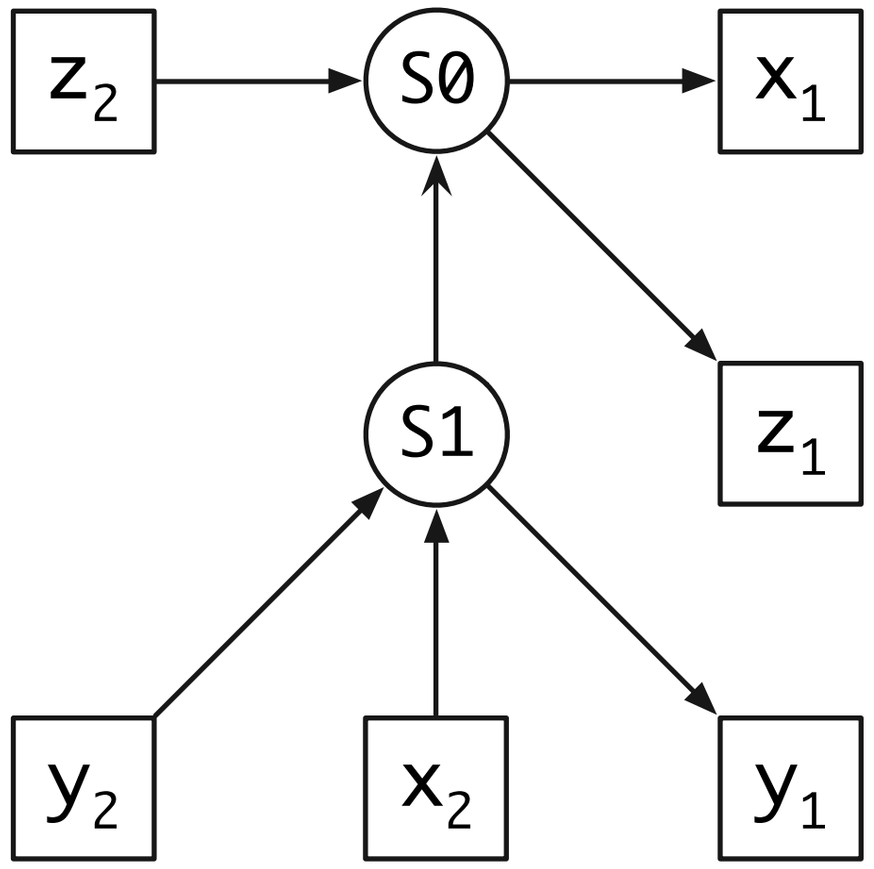
```
                                  S0
def x1 = z2 5

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

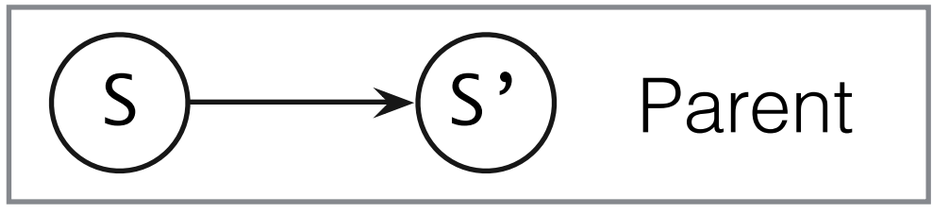




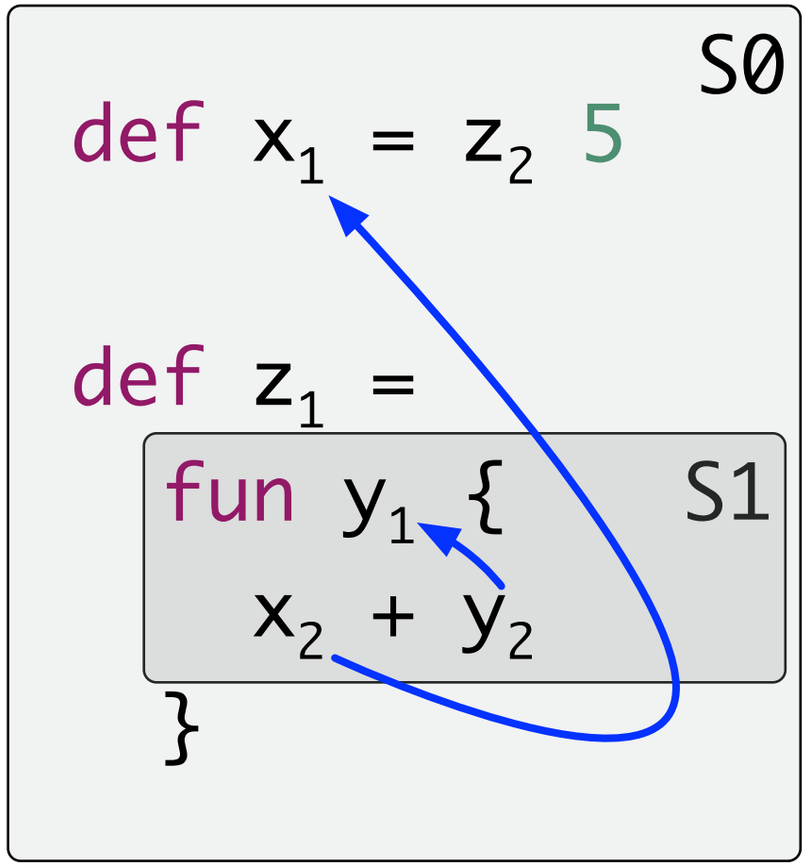
Lexical Scoping

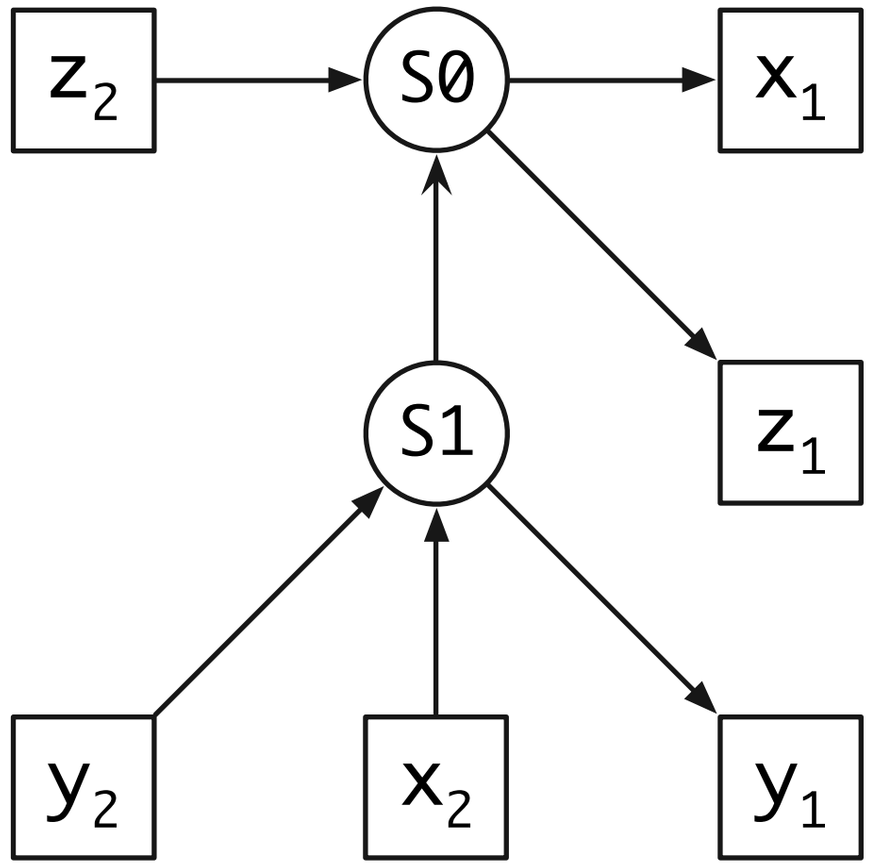
```
                                  S0
def x1 = z2 5

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

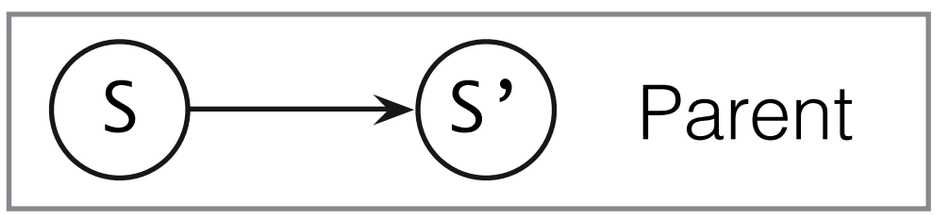




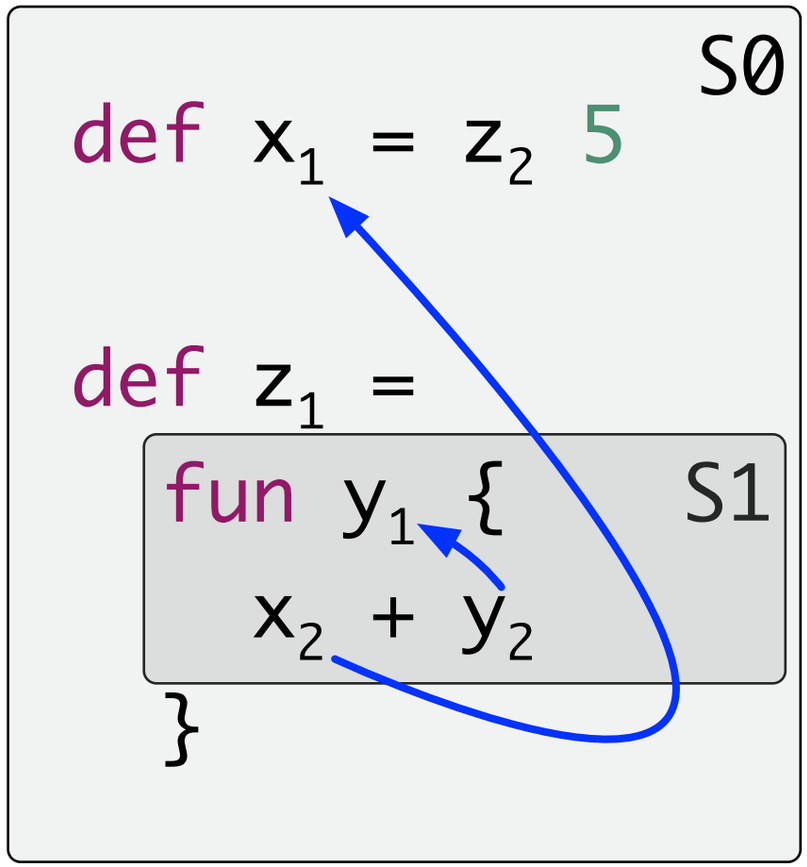
Lexical Scoping

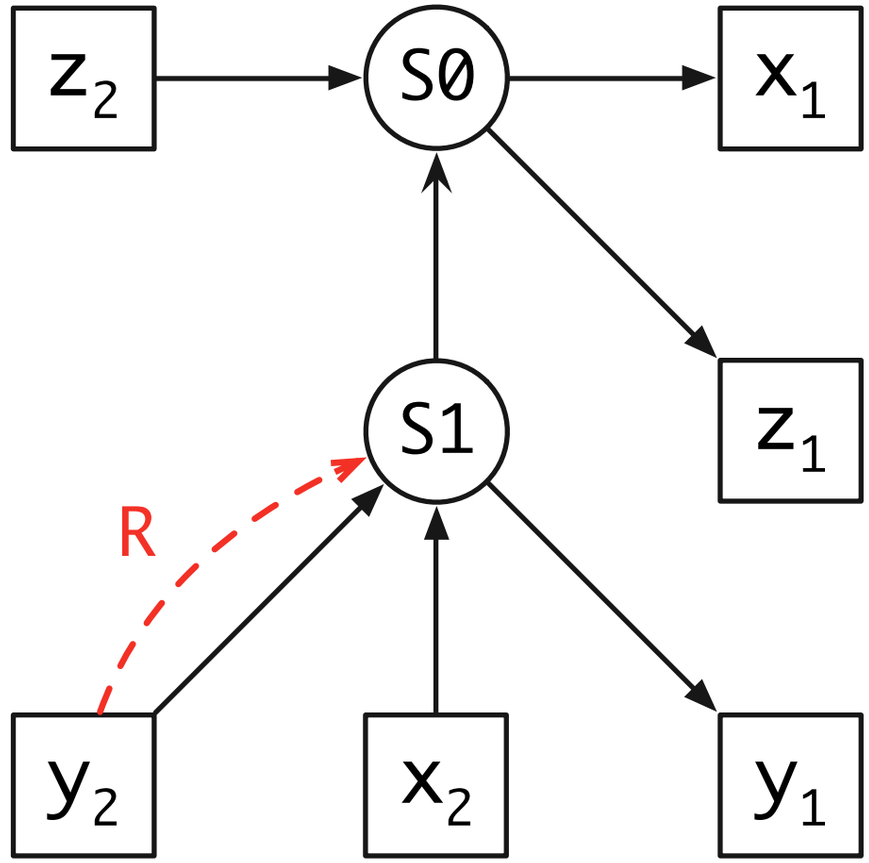


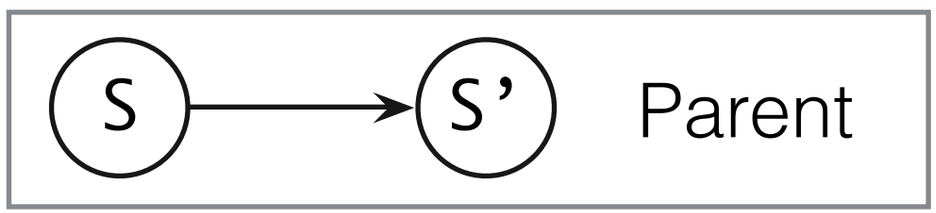




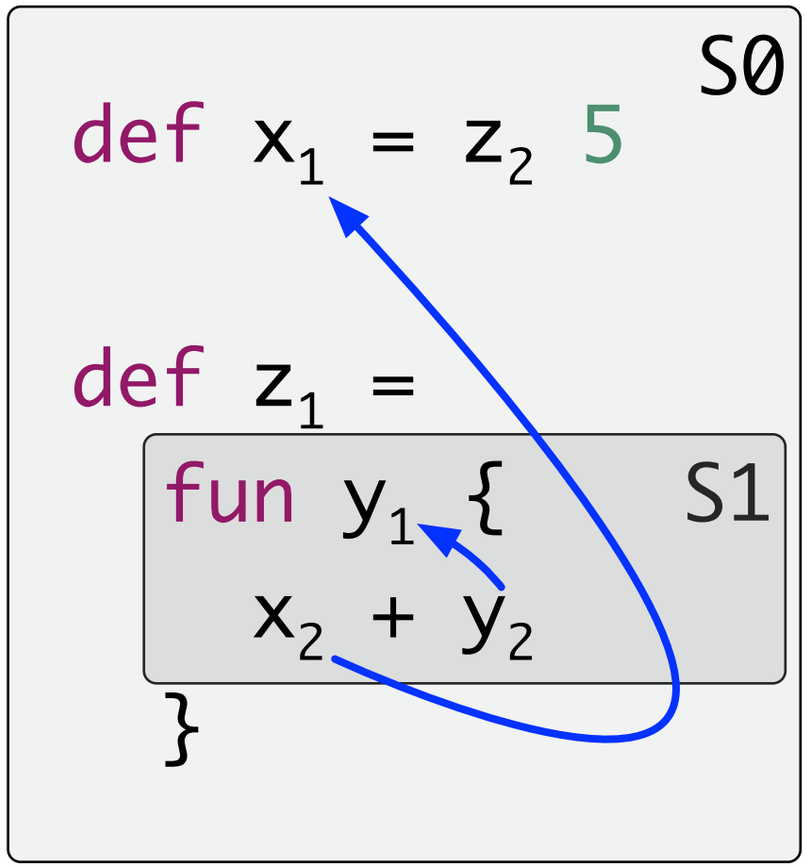
Lexical Scoping

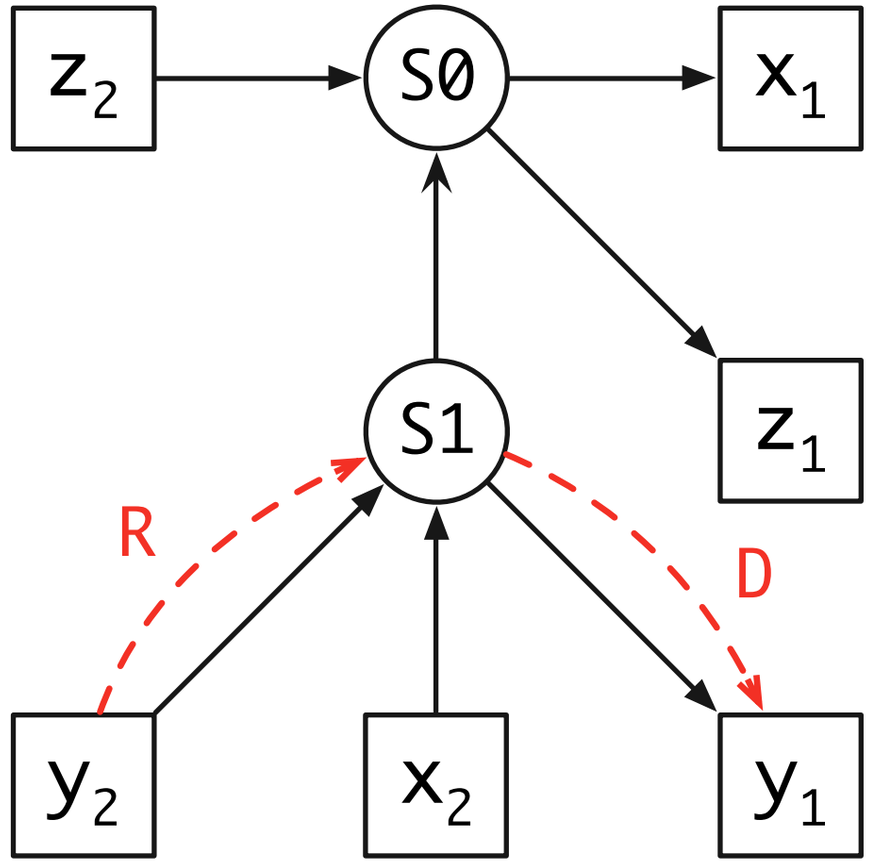


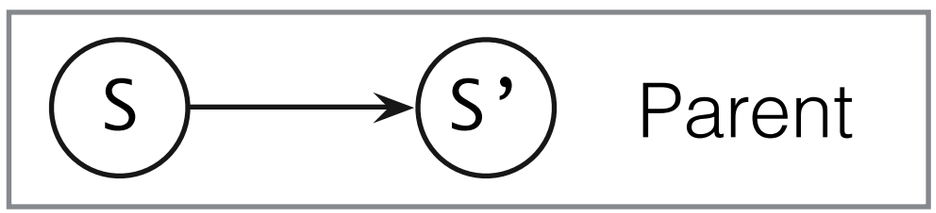




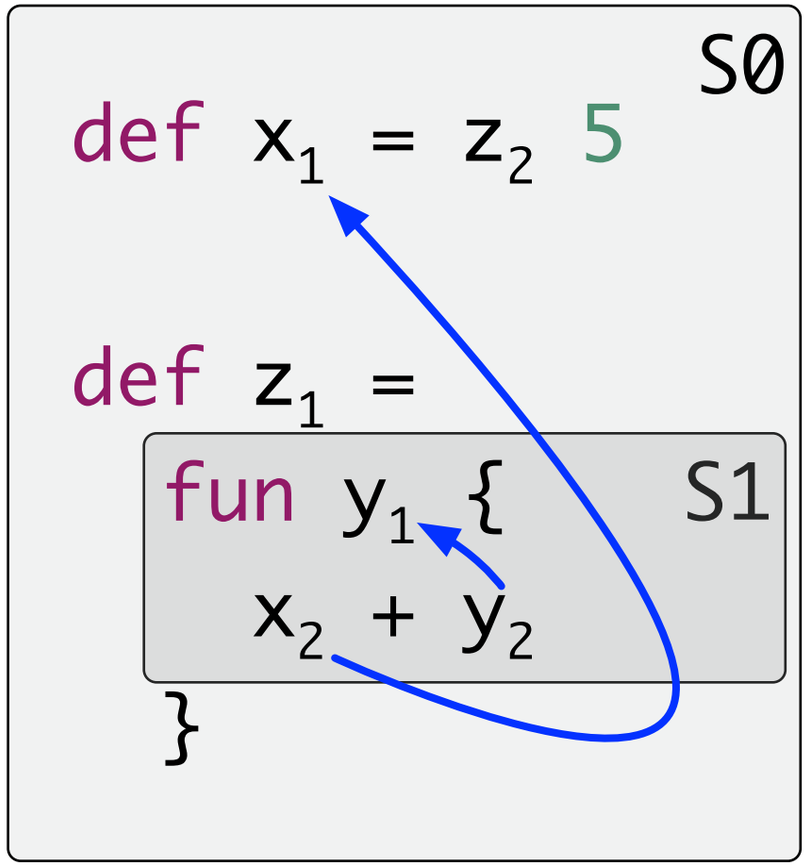
Lexical Scoping

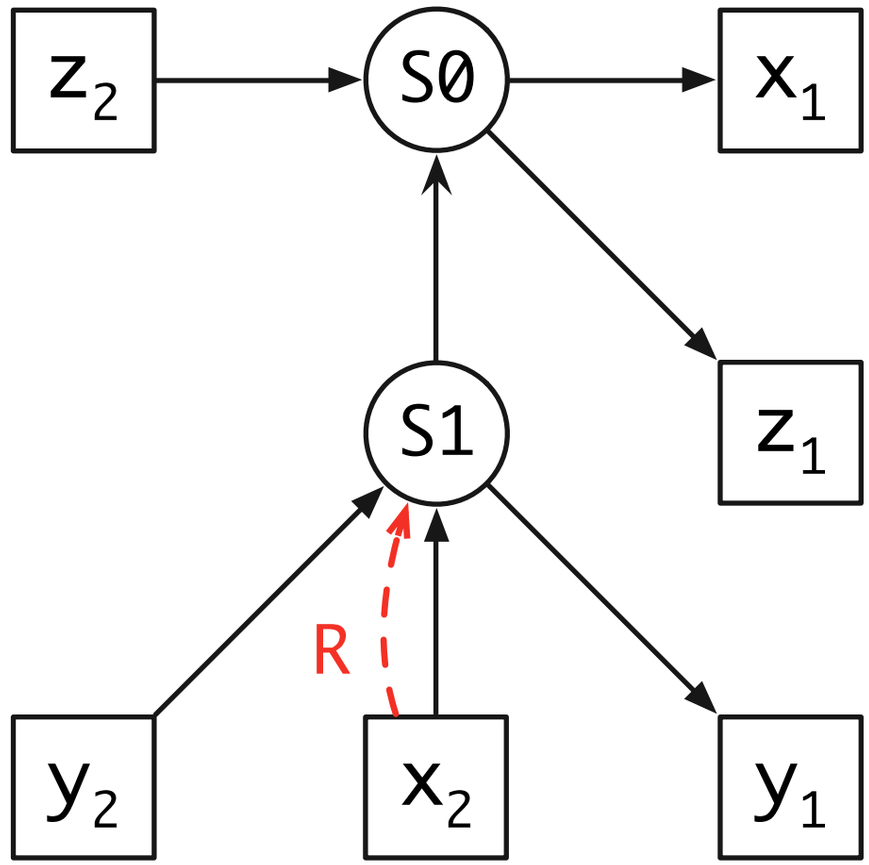


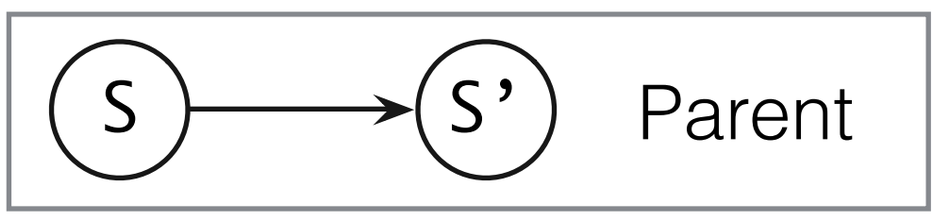




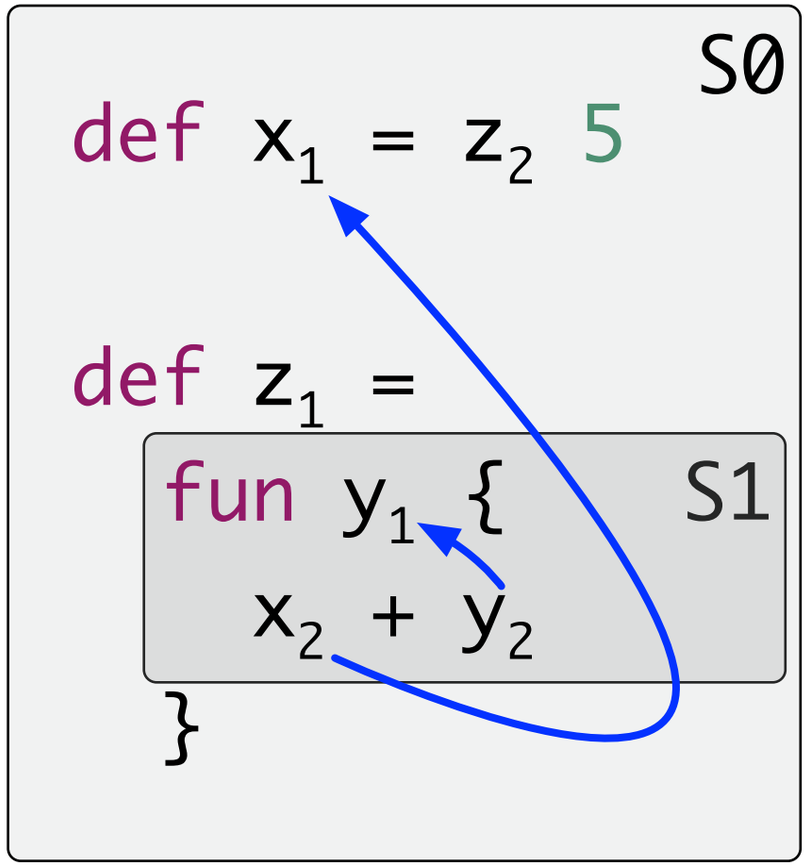
Lexical Scoping

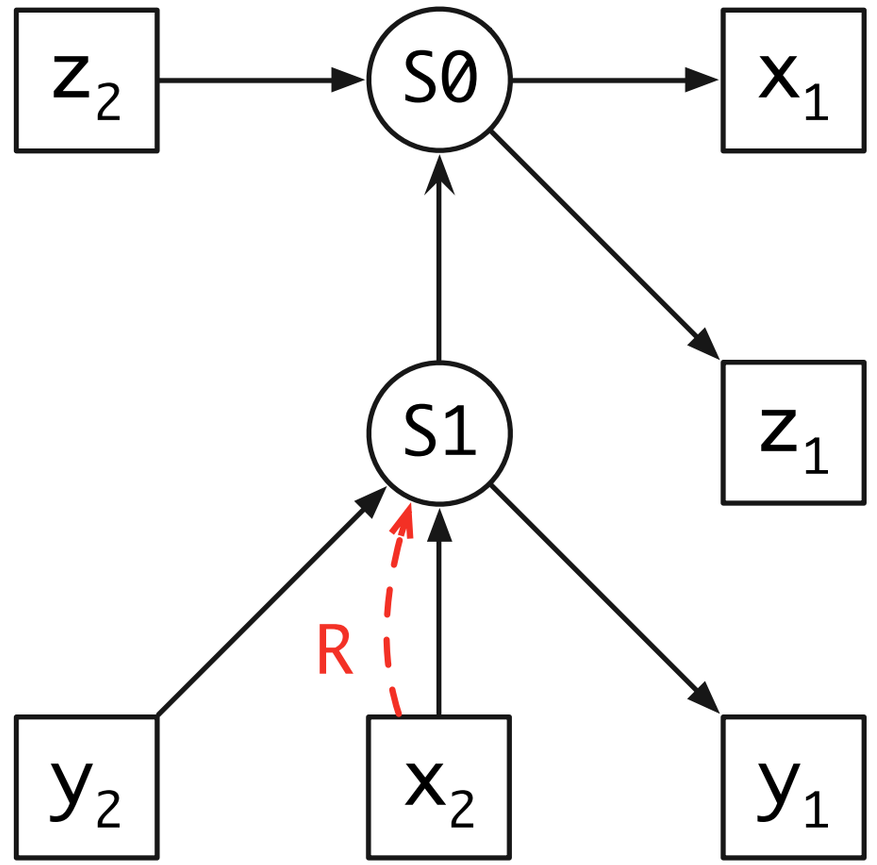


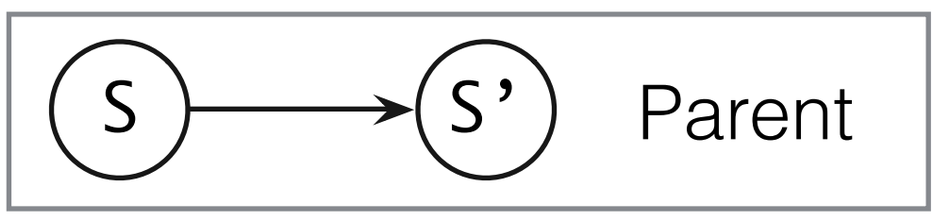


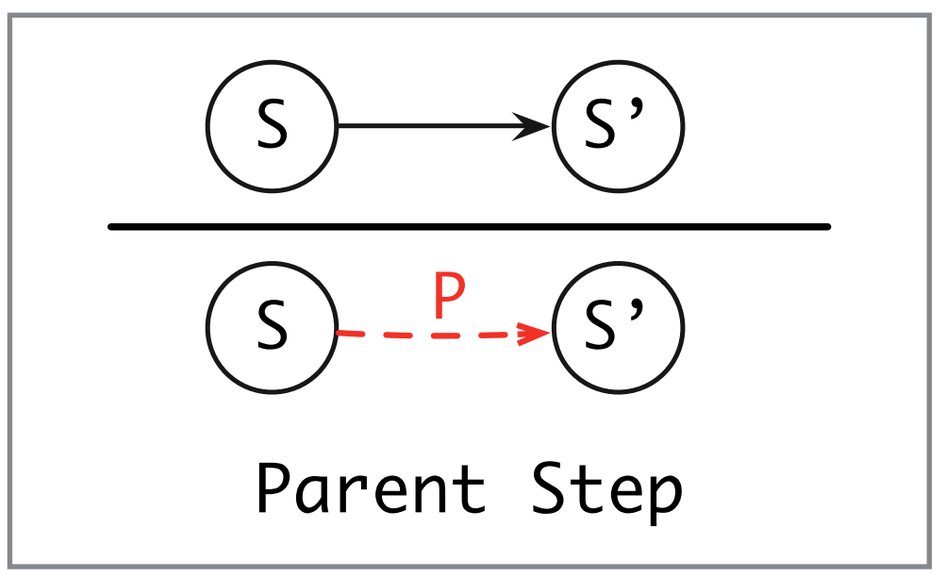


Lexical Scoping

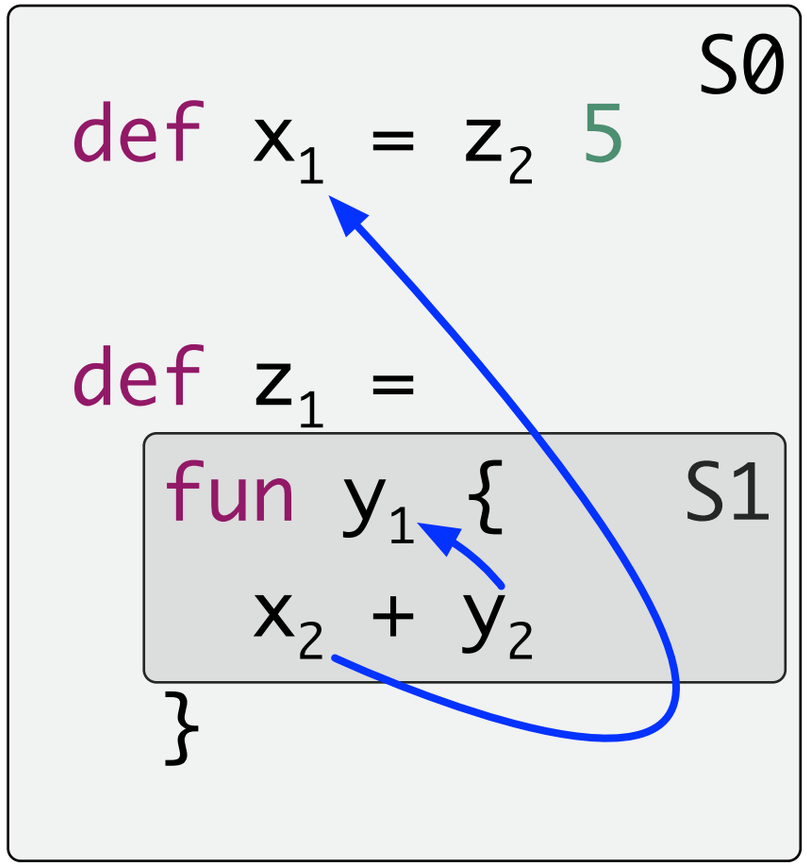


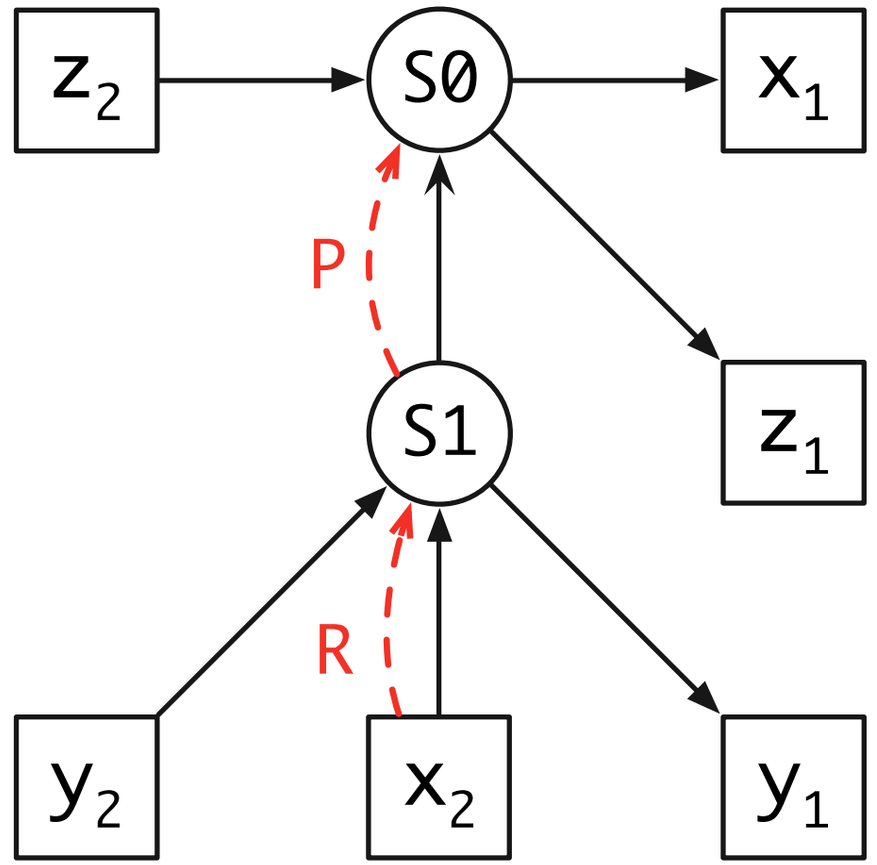


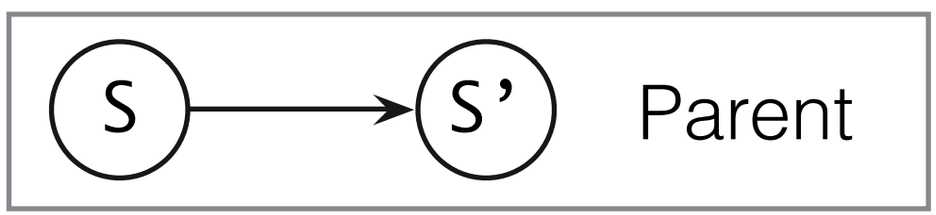


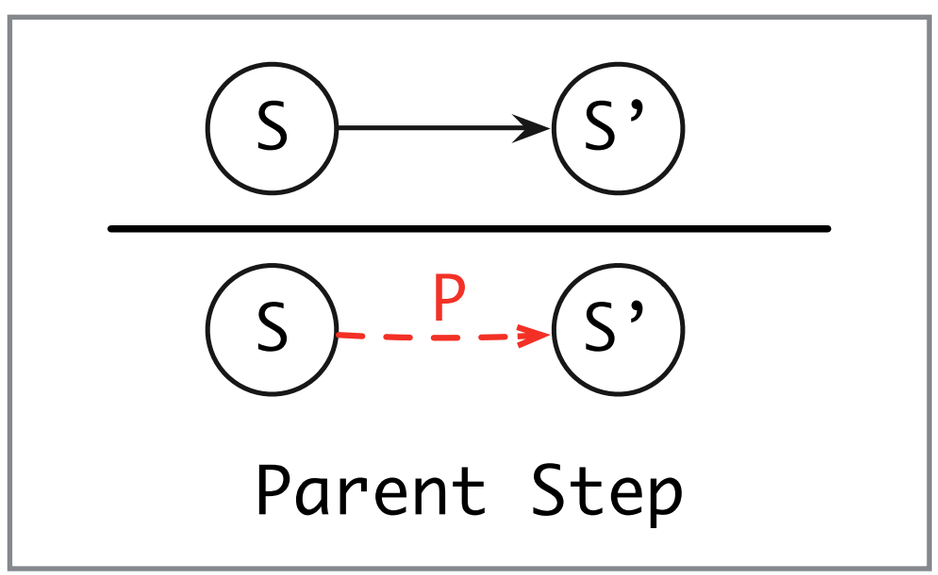


Lexical Scoping

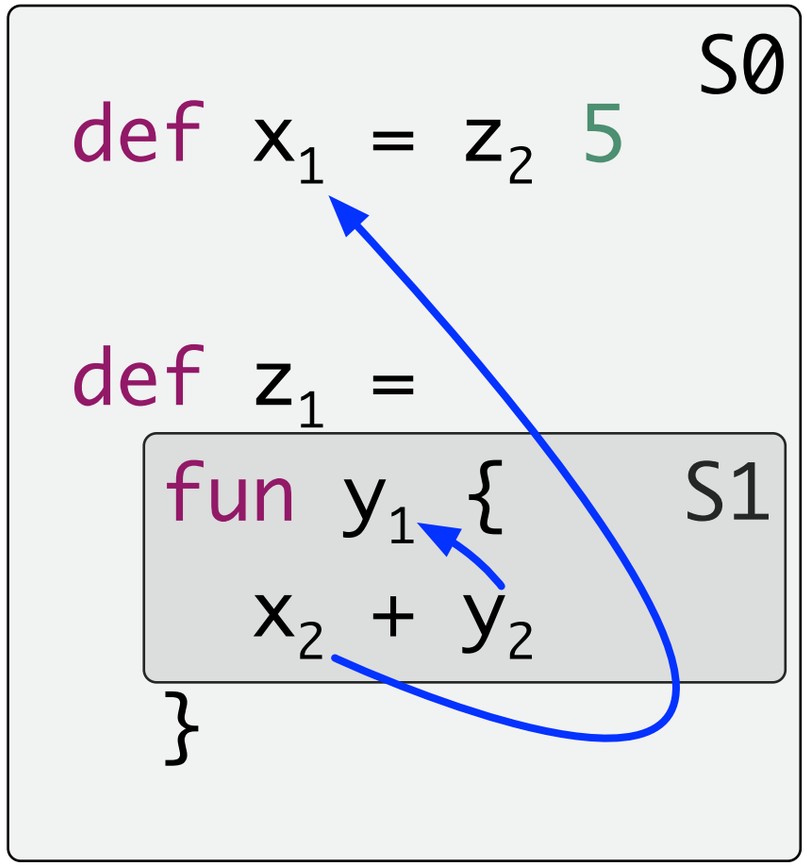


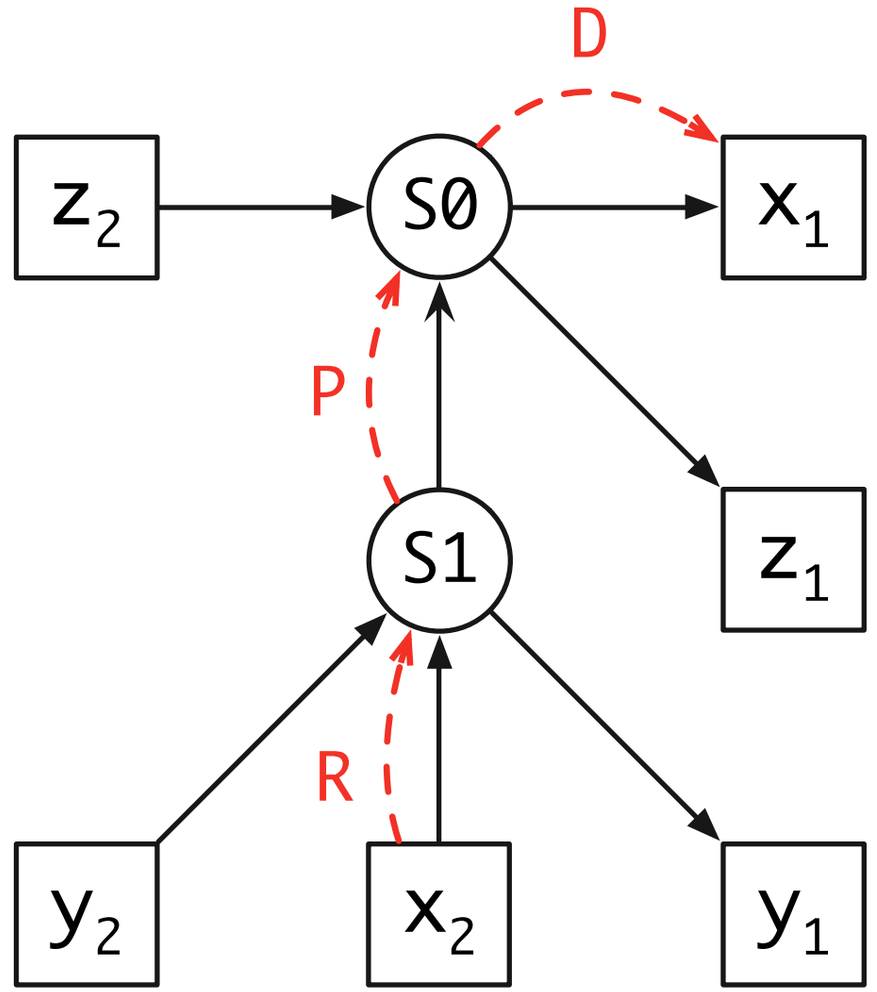


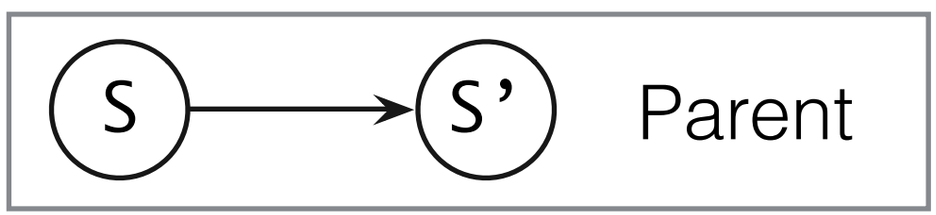


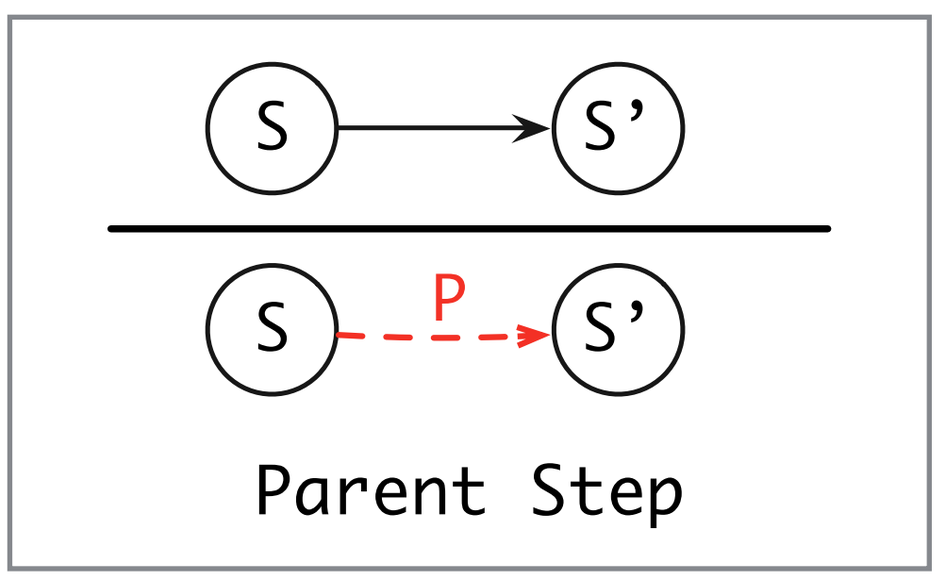


Lexical Scoping







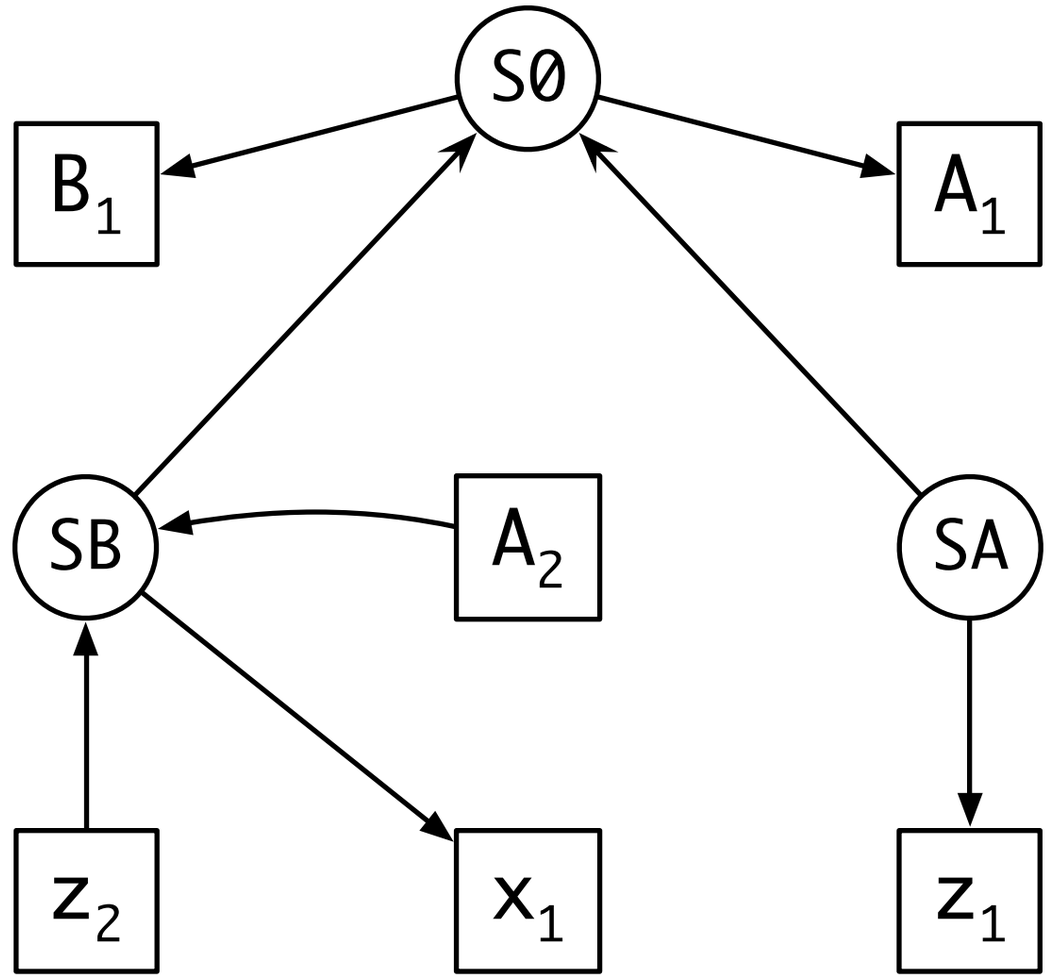


Imports

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

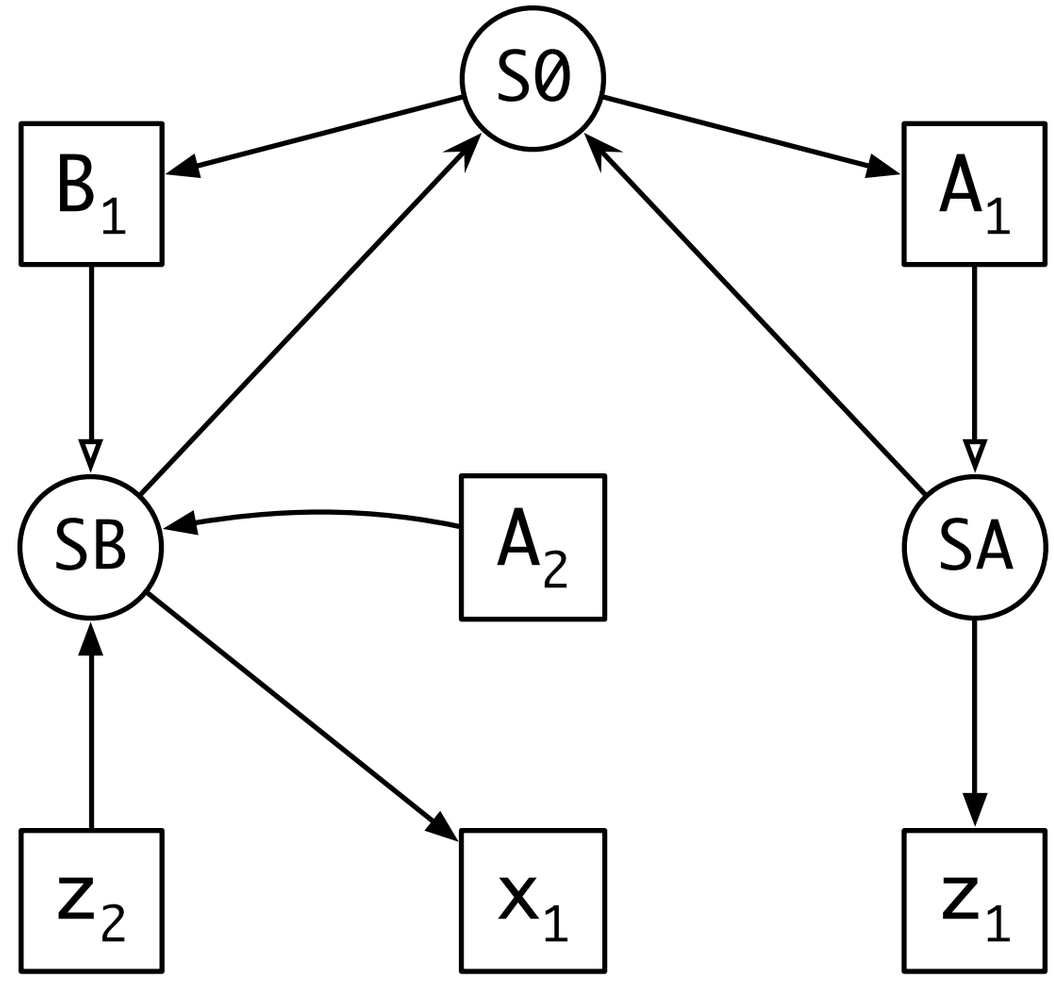
Imports

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



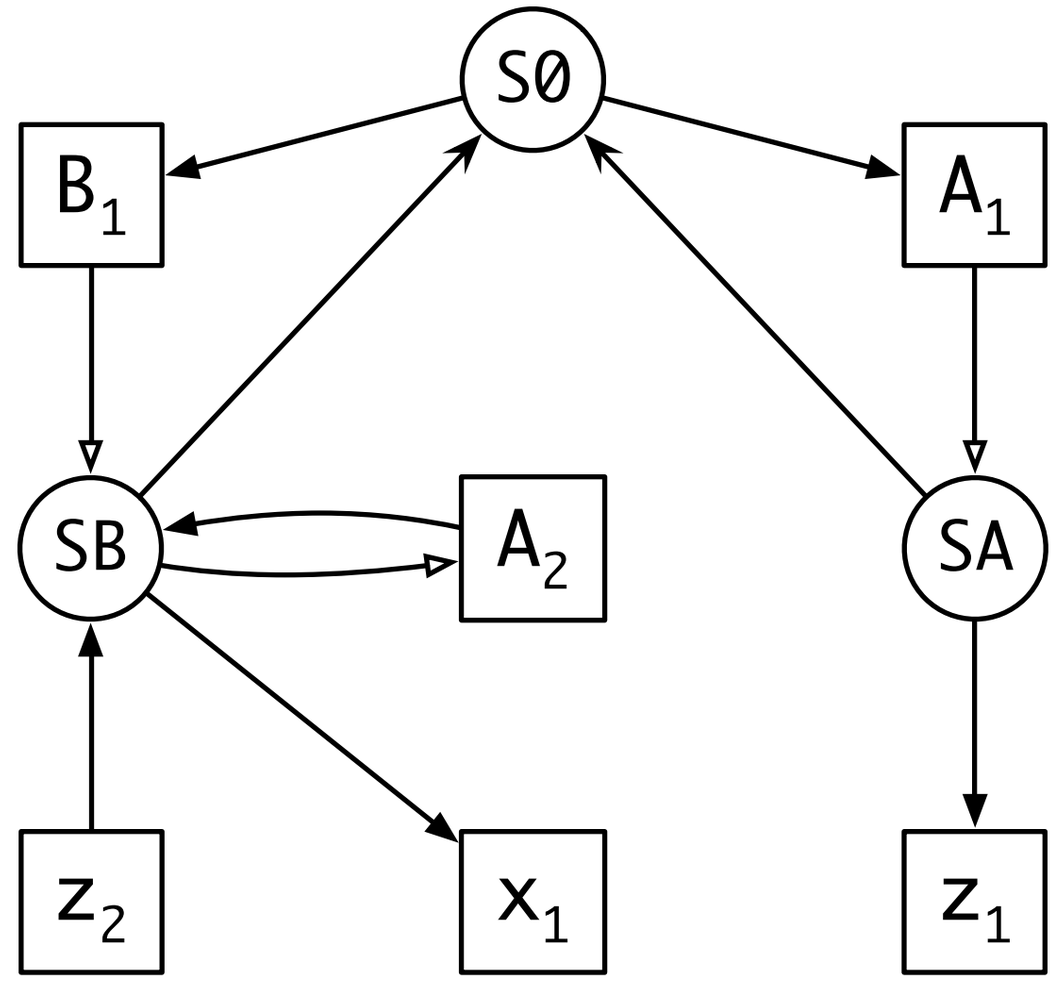
Imports

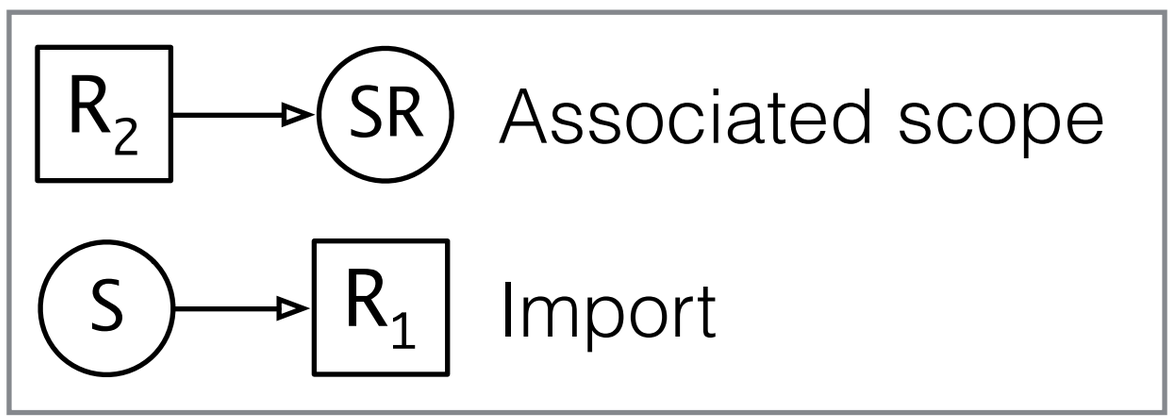
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```



Imports

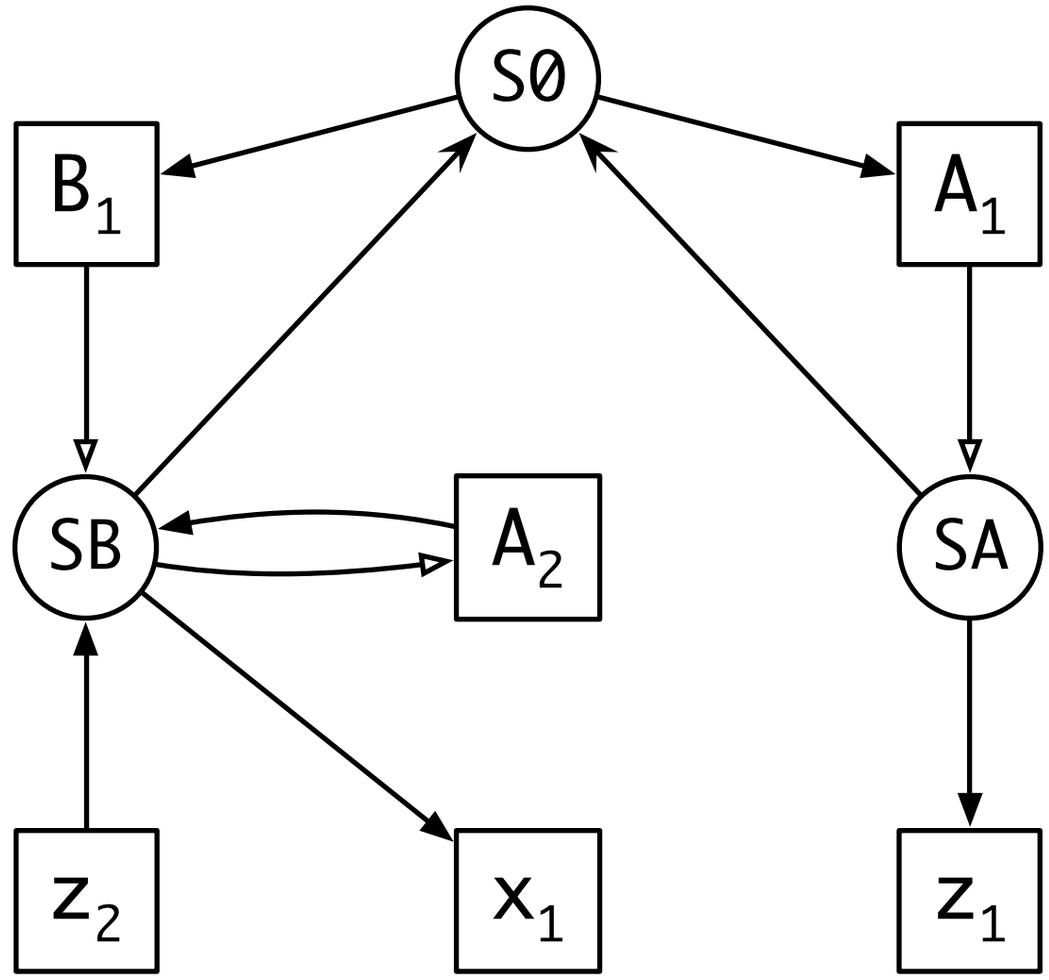
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

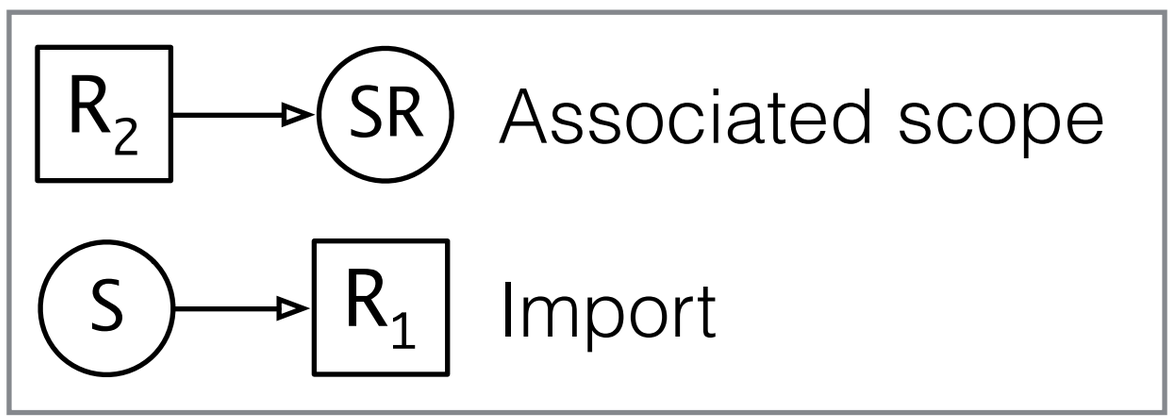




Imports

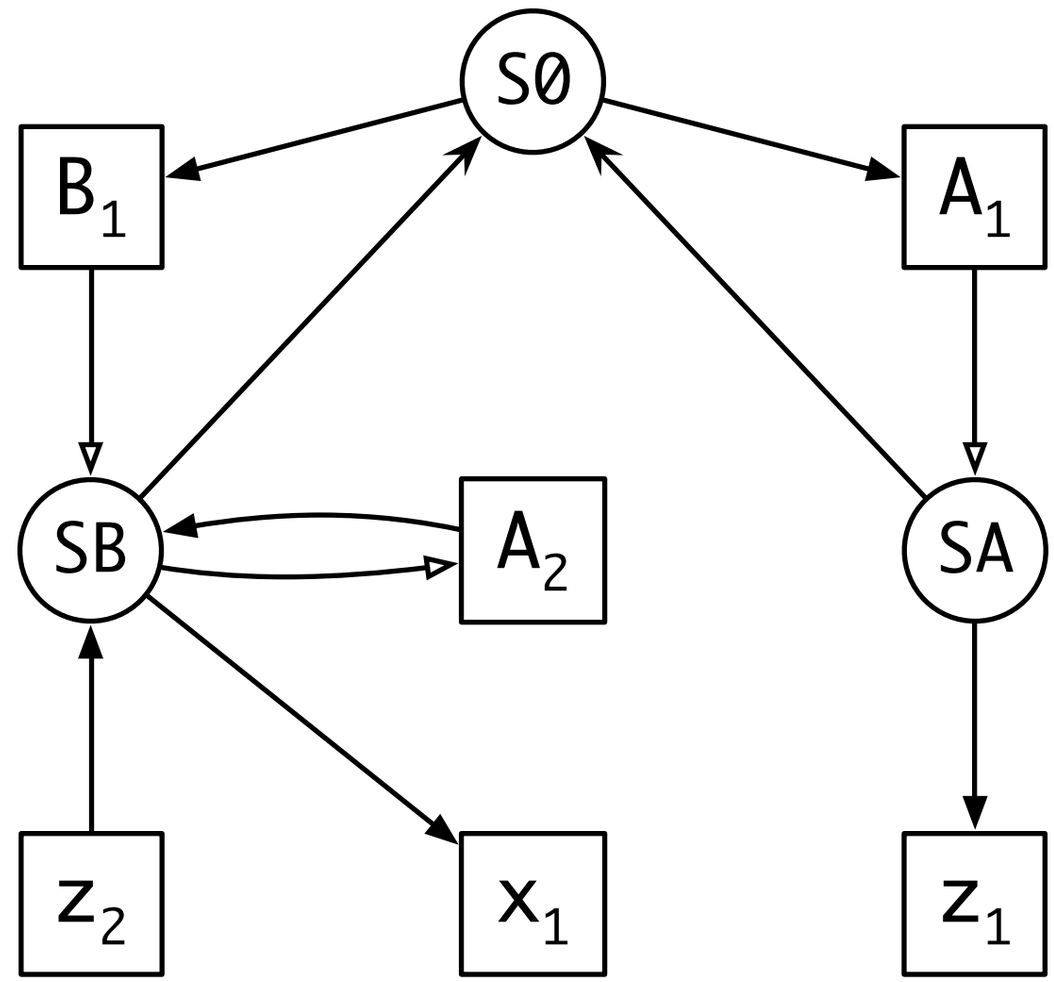
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

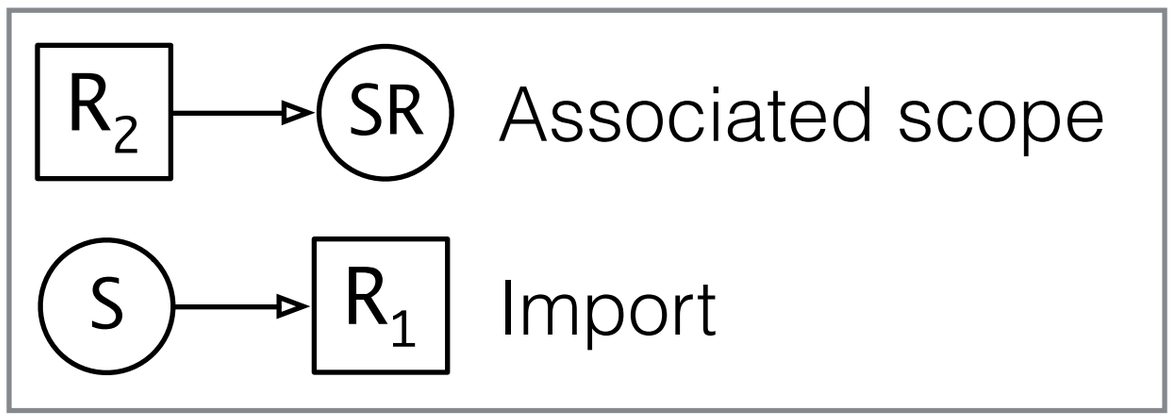




Imports

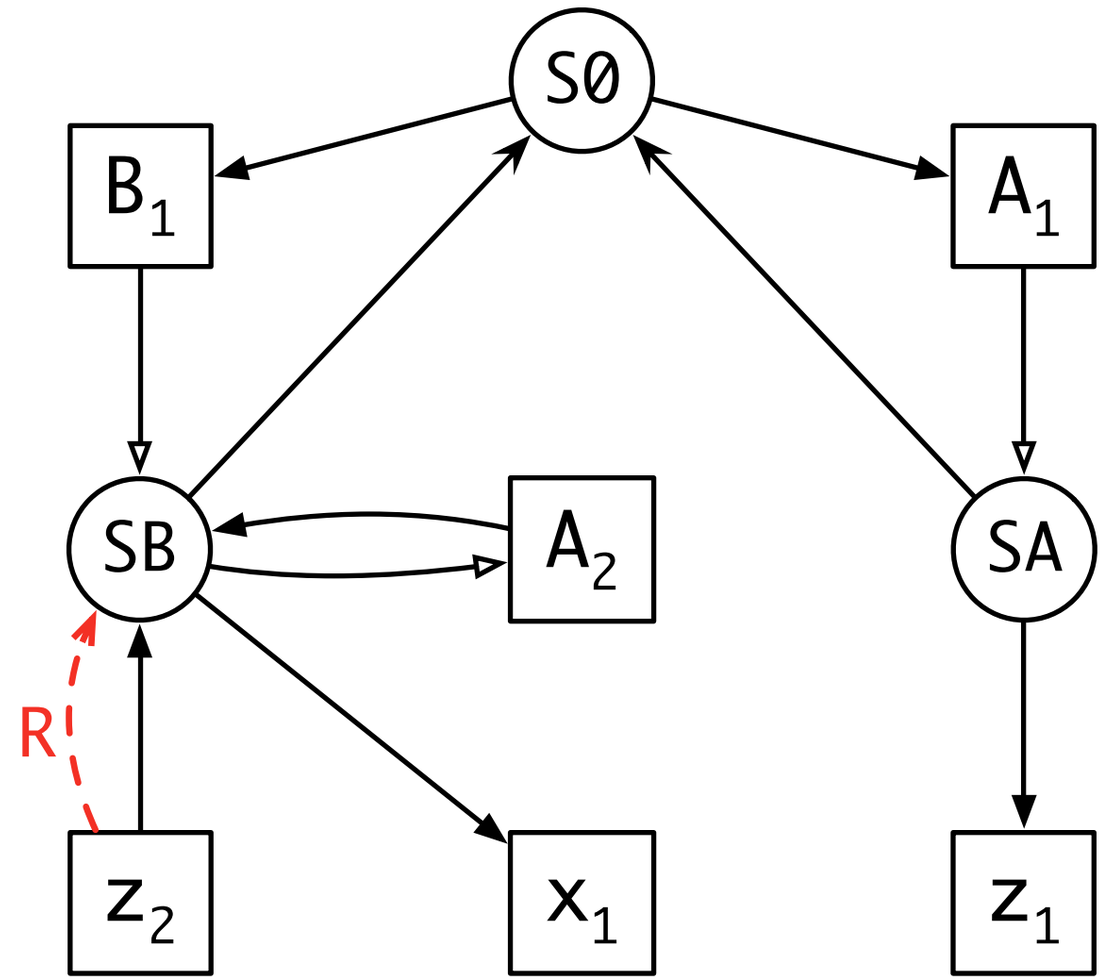
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

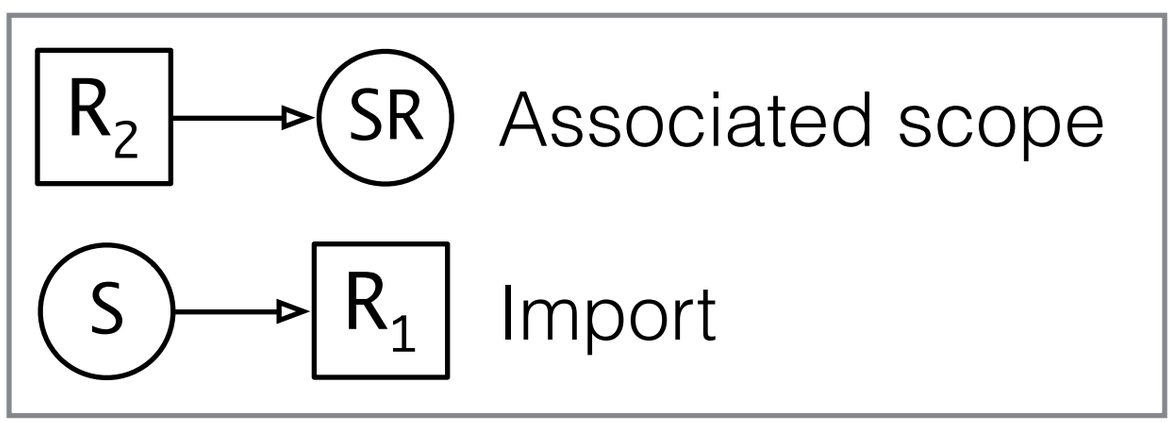




Imports

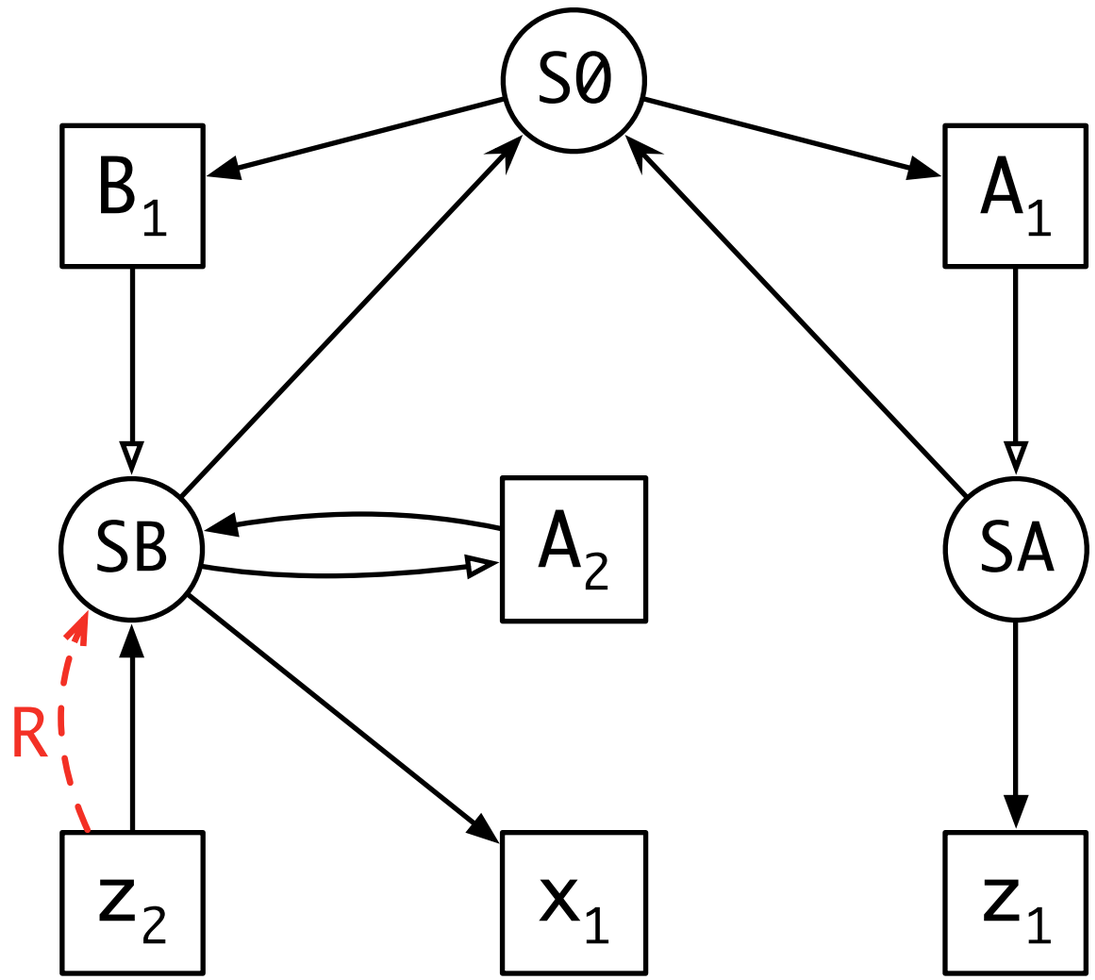
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

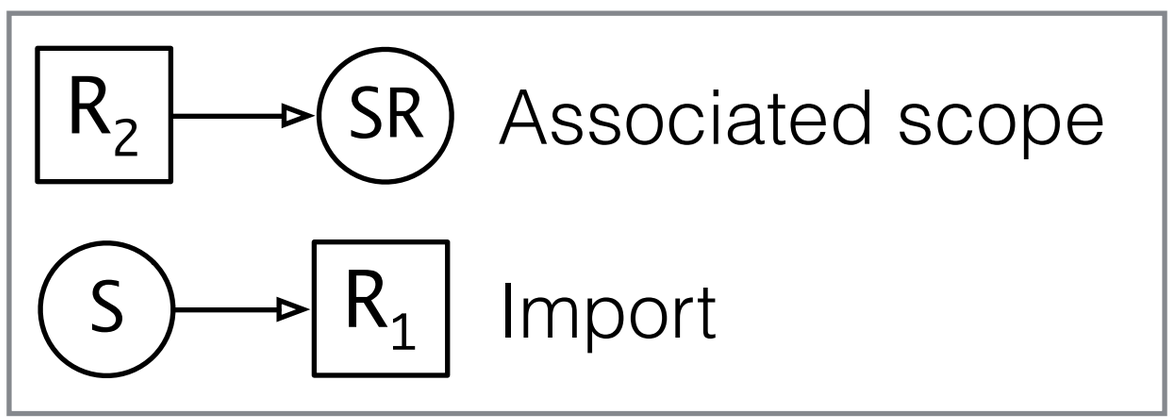


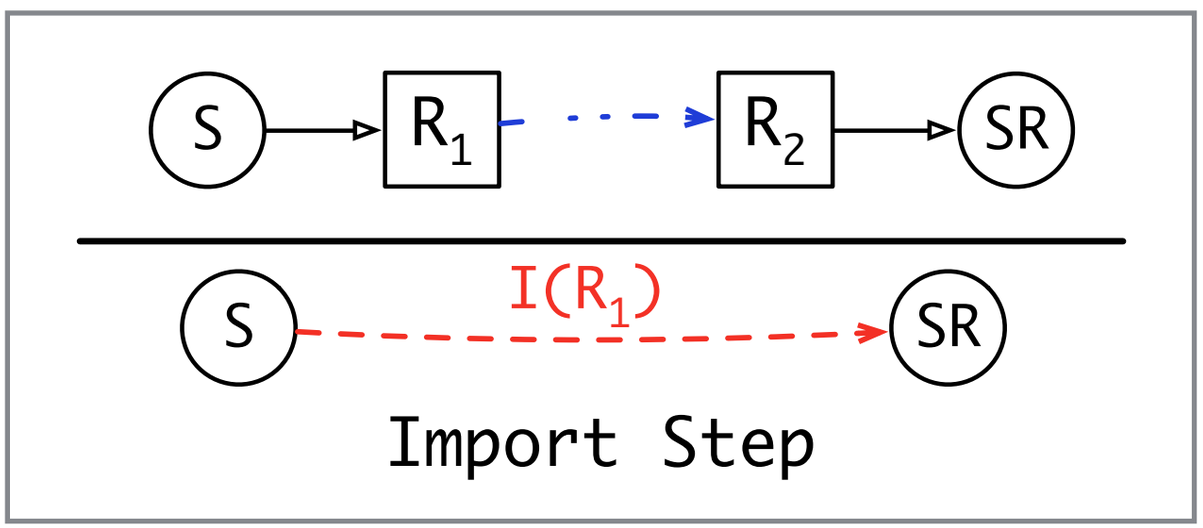


Imports

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

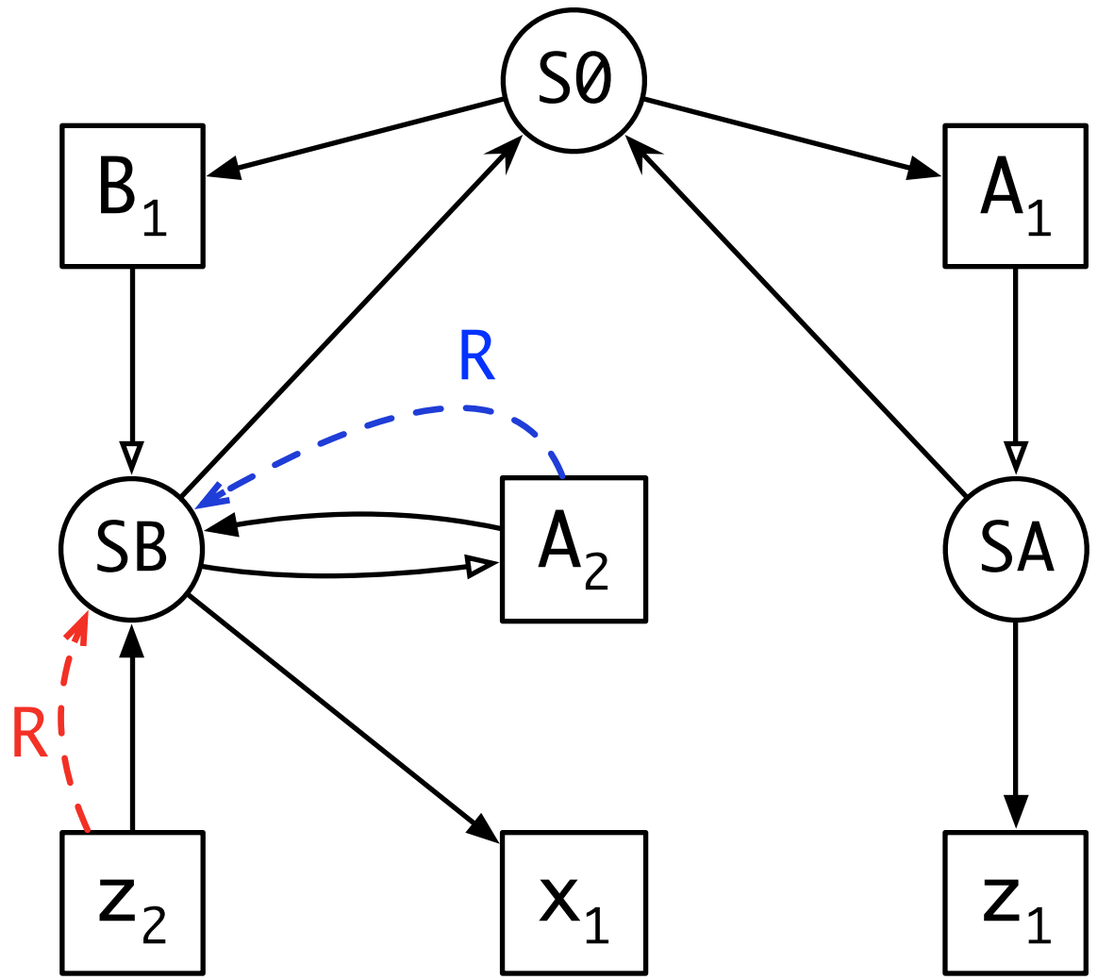


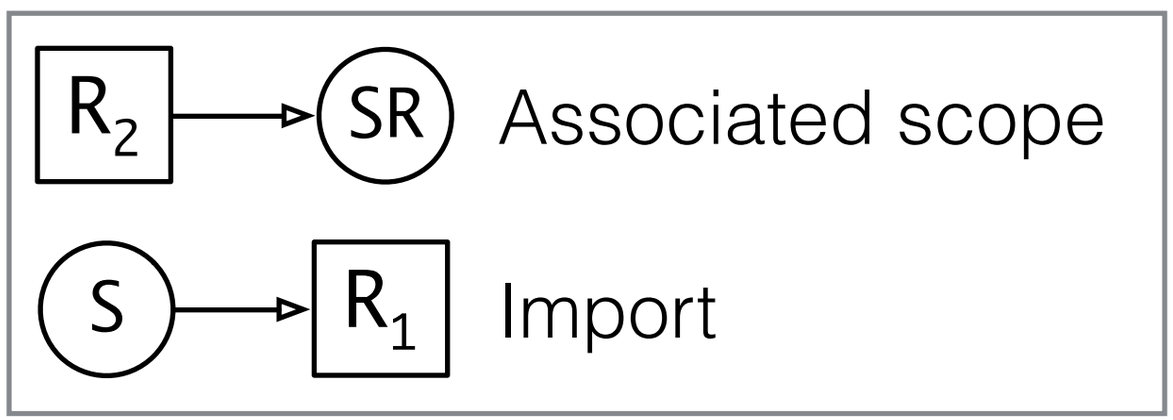


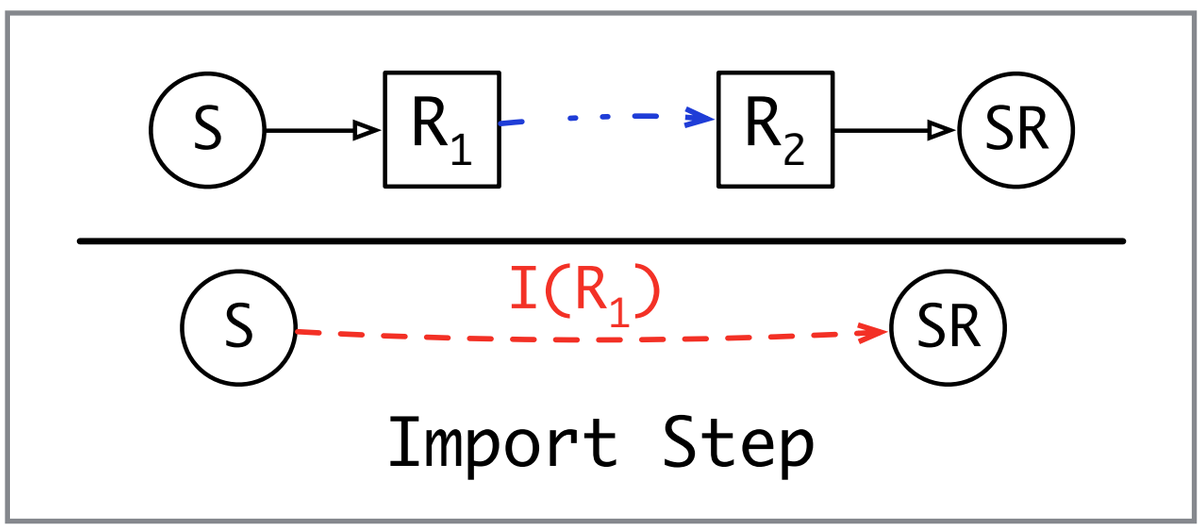


Imports

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

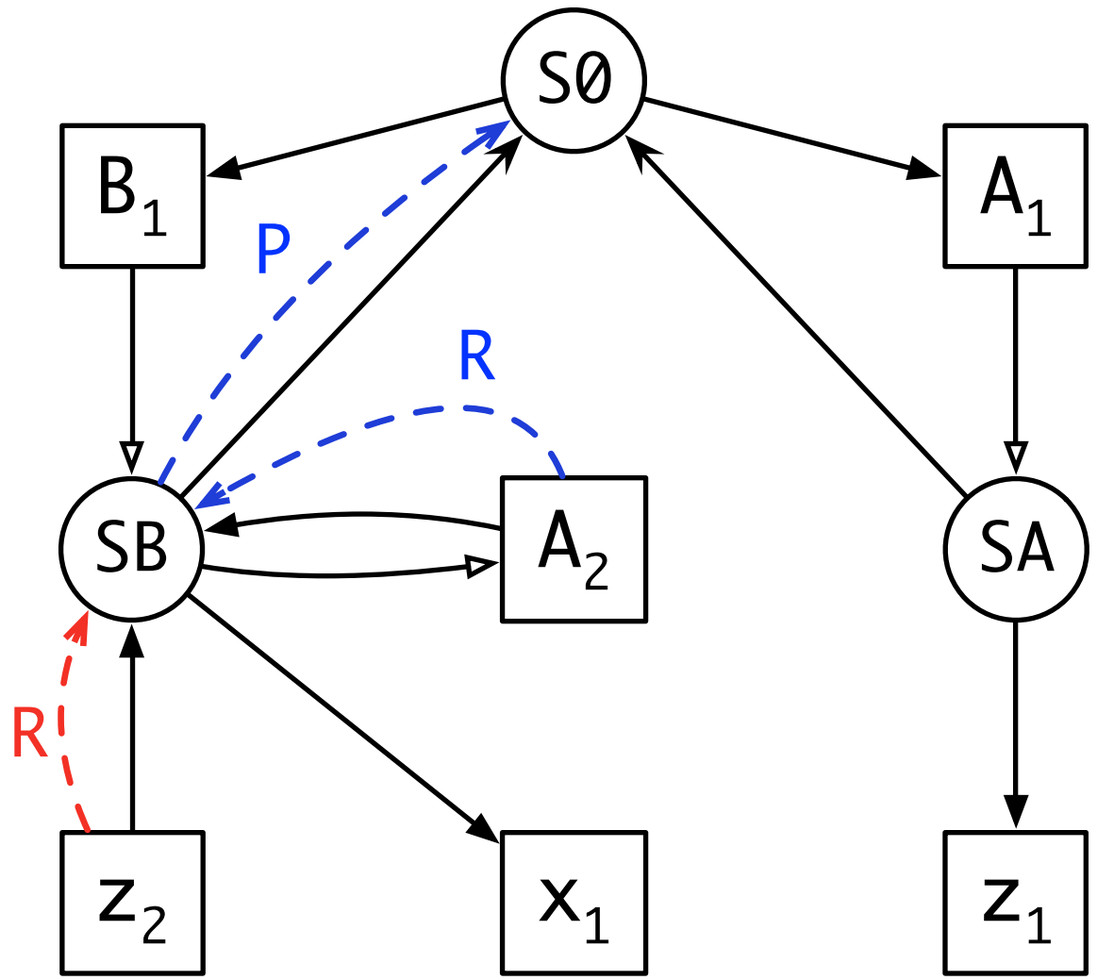


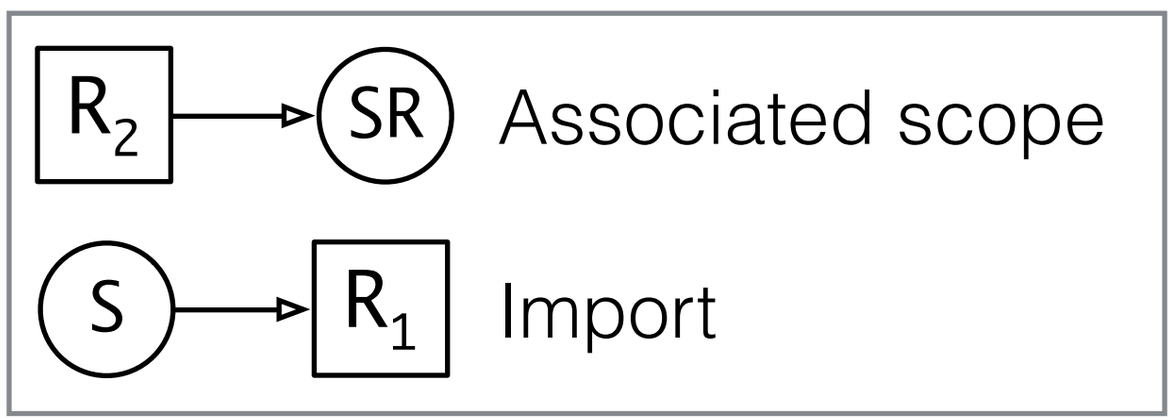


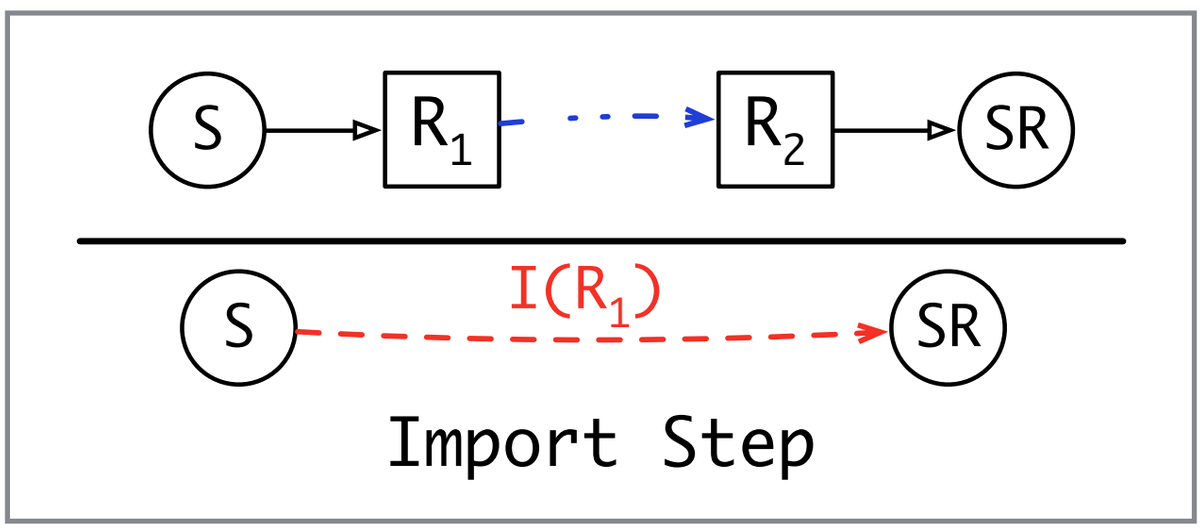


Imports

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

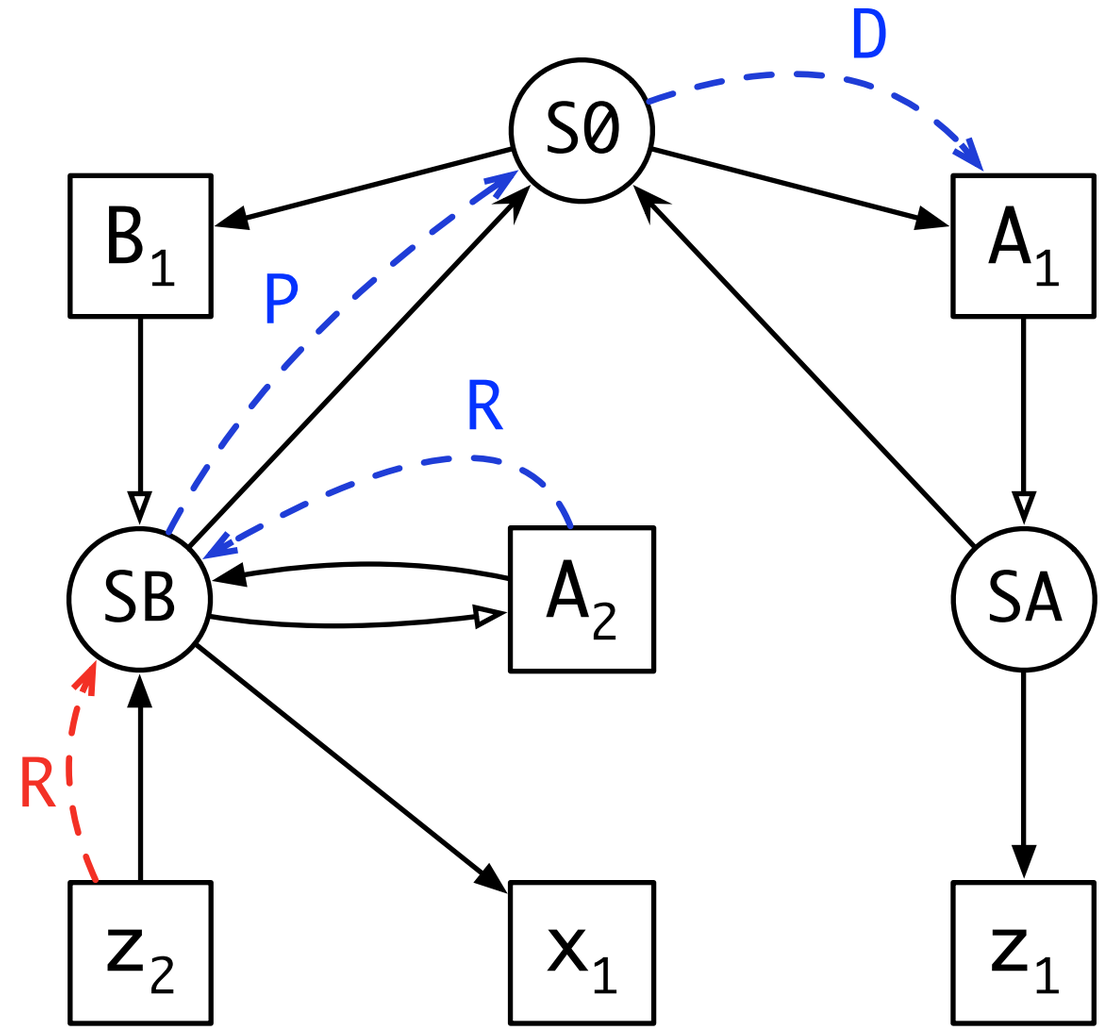


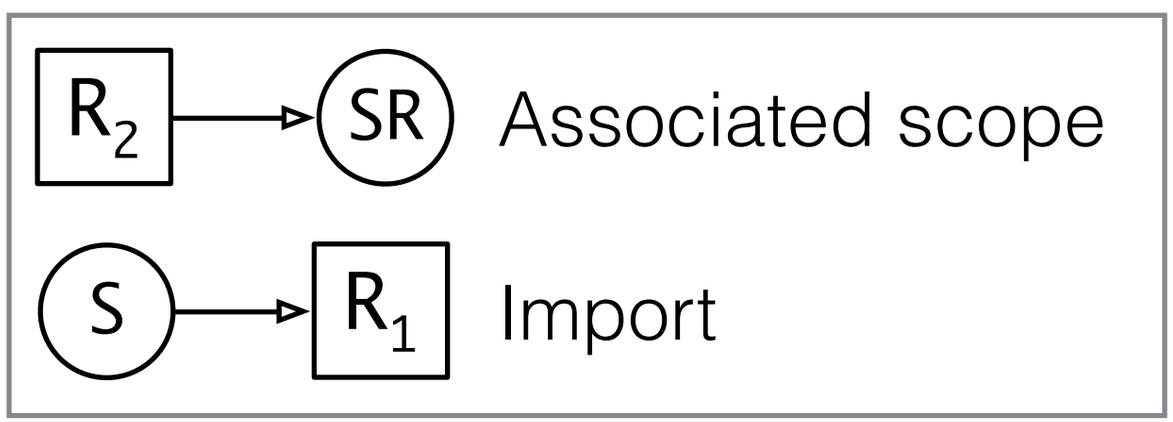


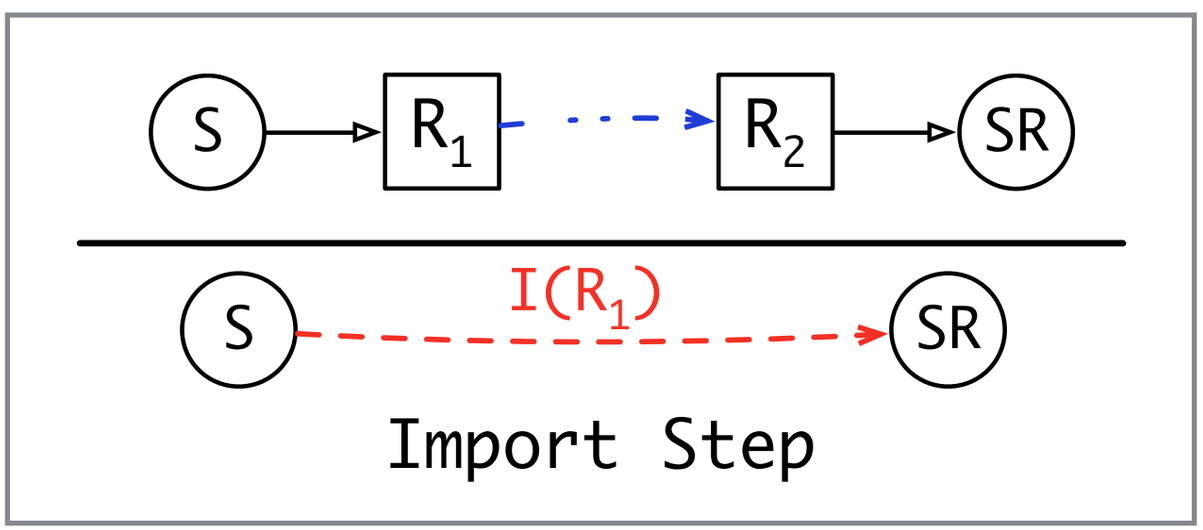


Imports

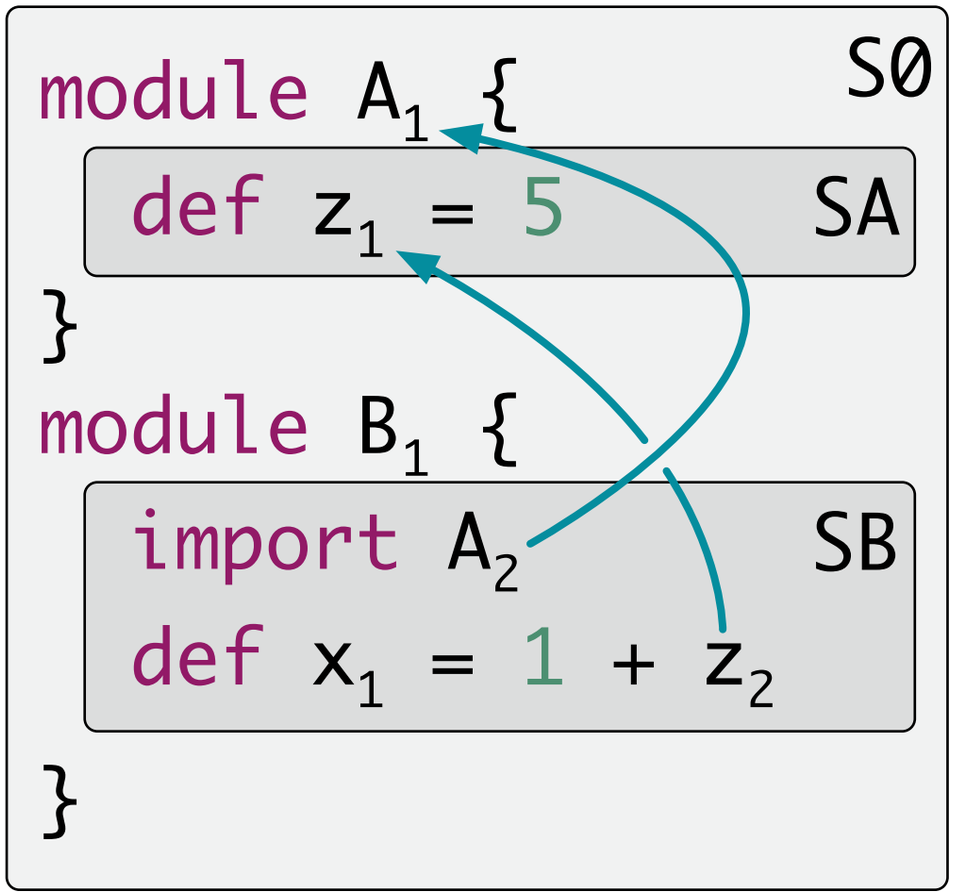
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

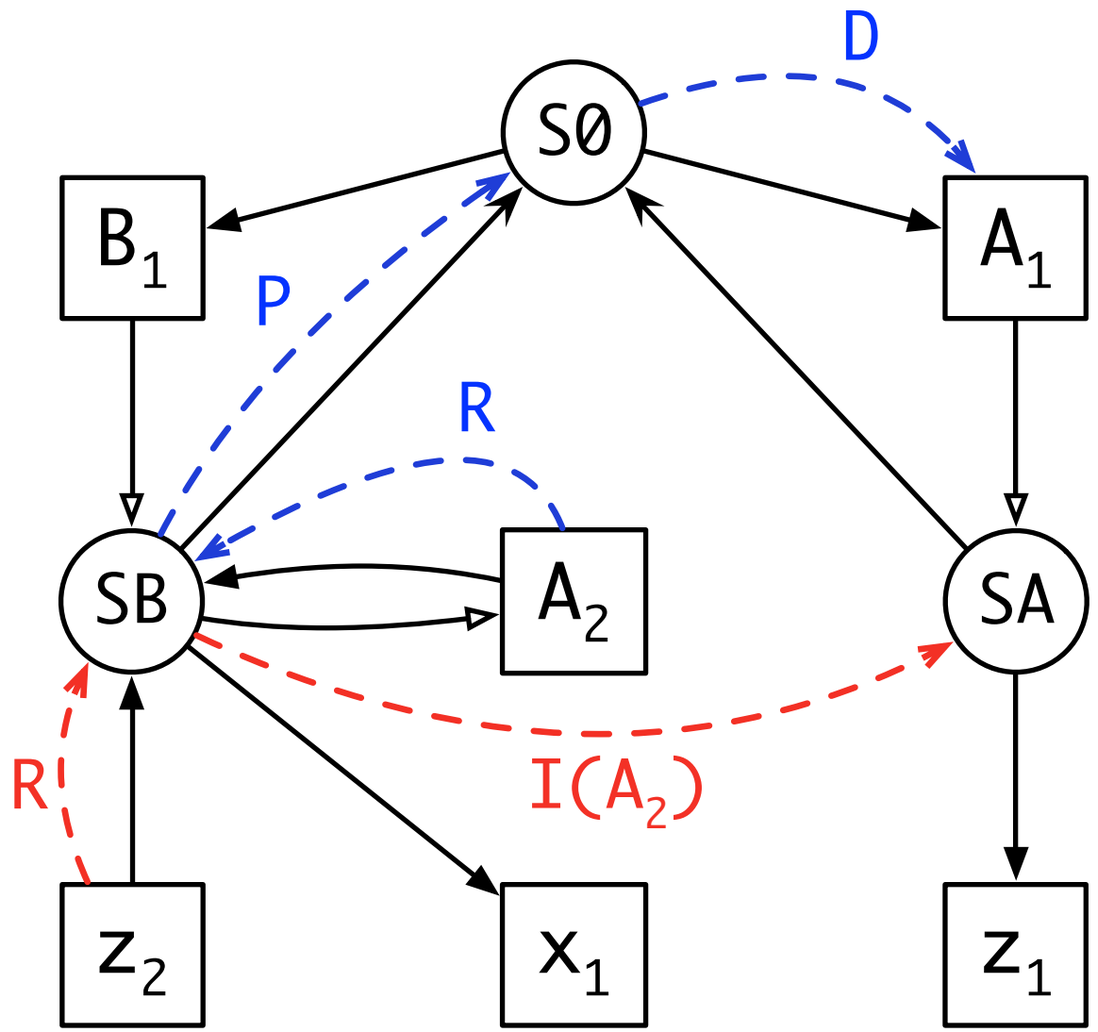


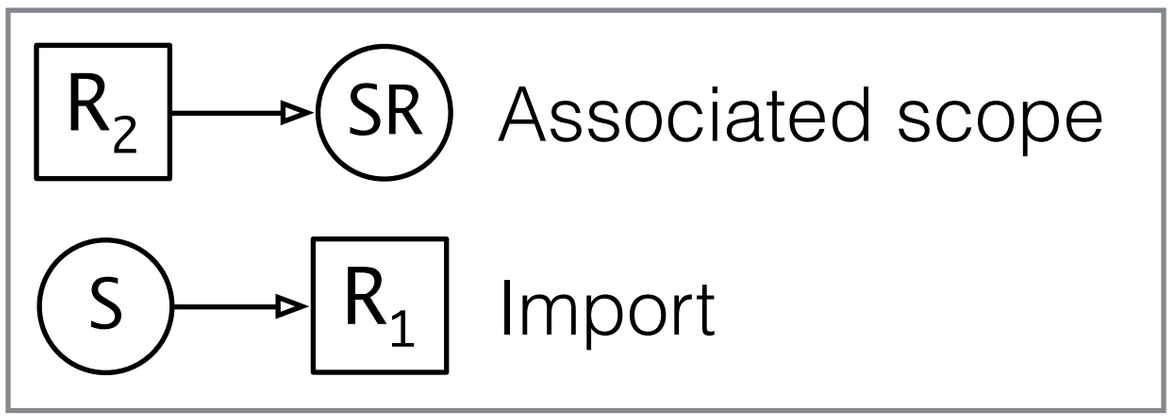


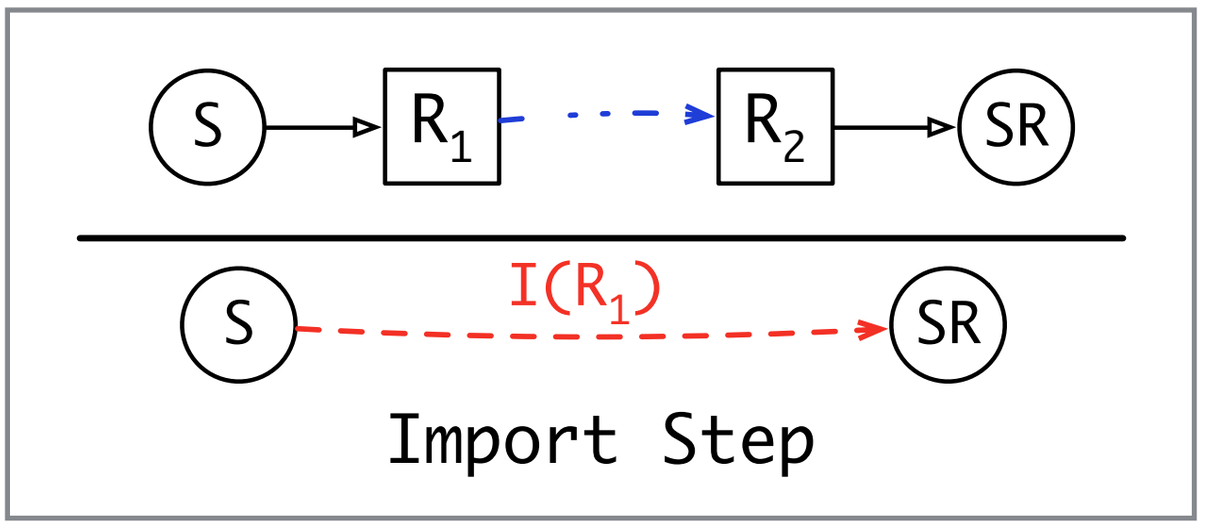


Imports


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

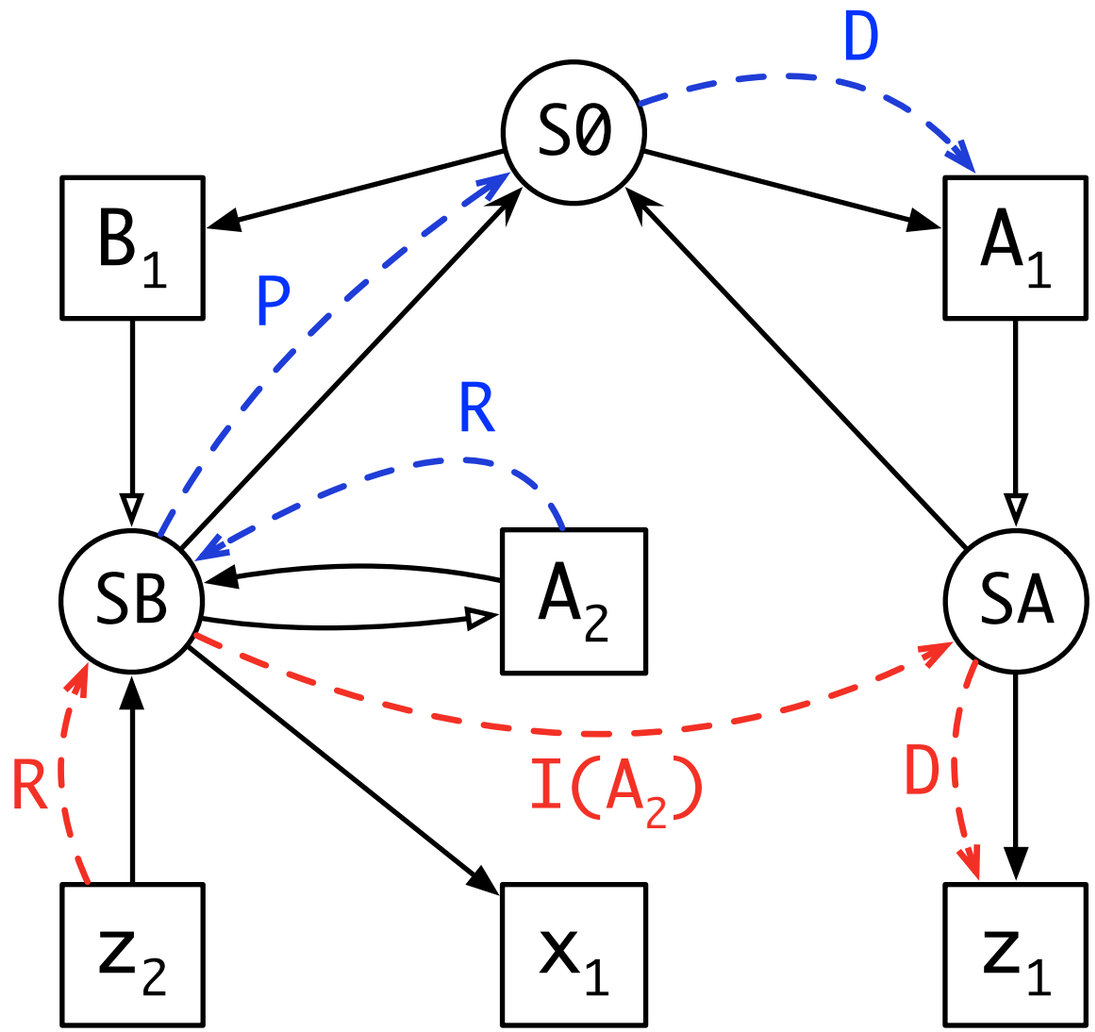

S0
D
B1
A1
P
R
SB
A2
SA
R
I(A2)
z2
x1
z1

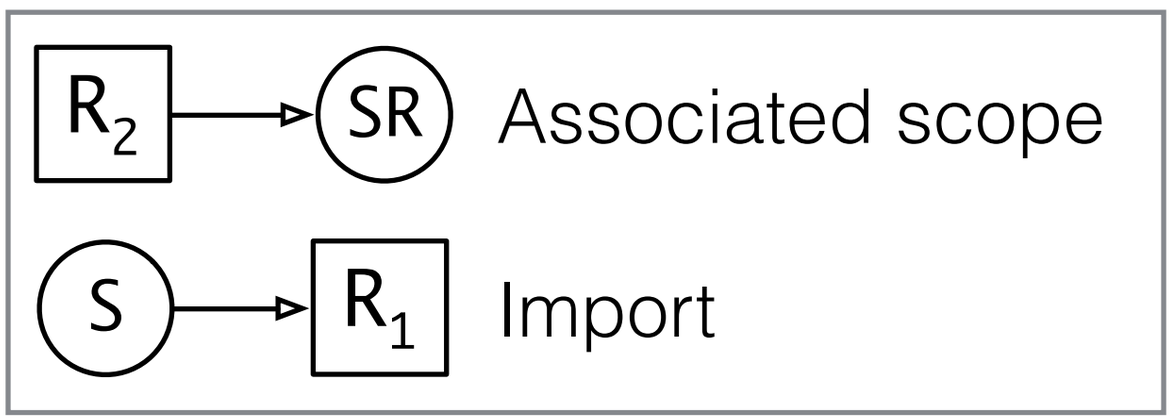

R2
SR
Associated scope
S
R1
Import

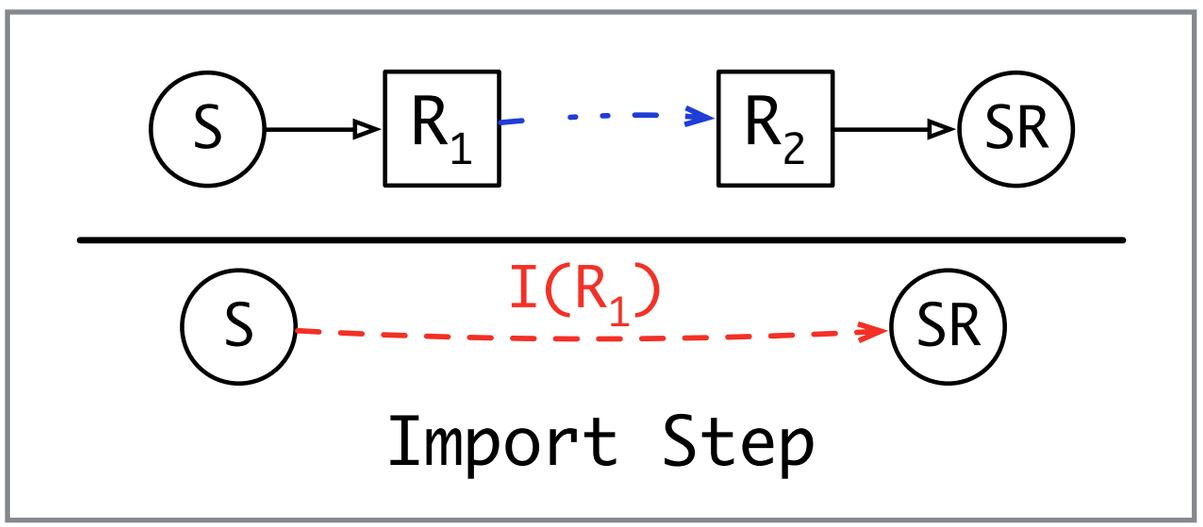

S
R1
R2
SR
I(R1)
S
SR
Import Step

Imports

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





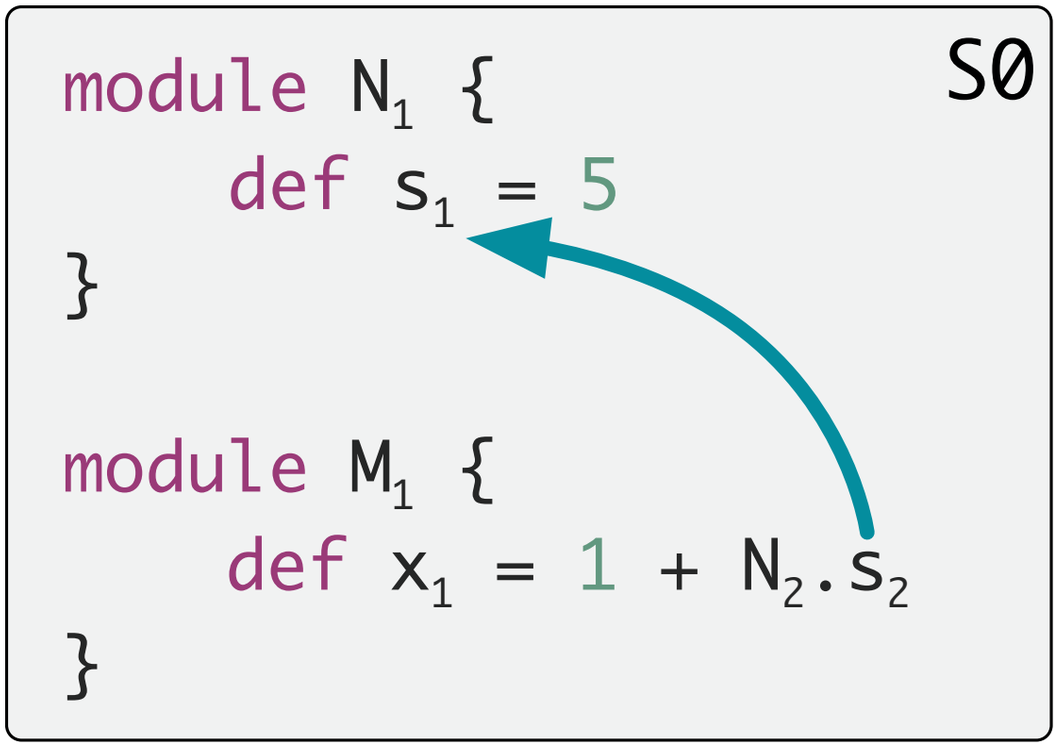


Qualified Names

S0

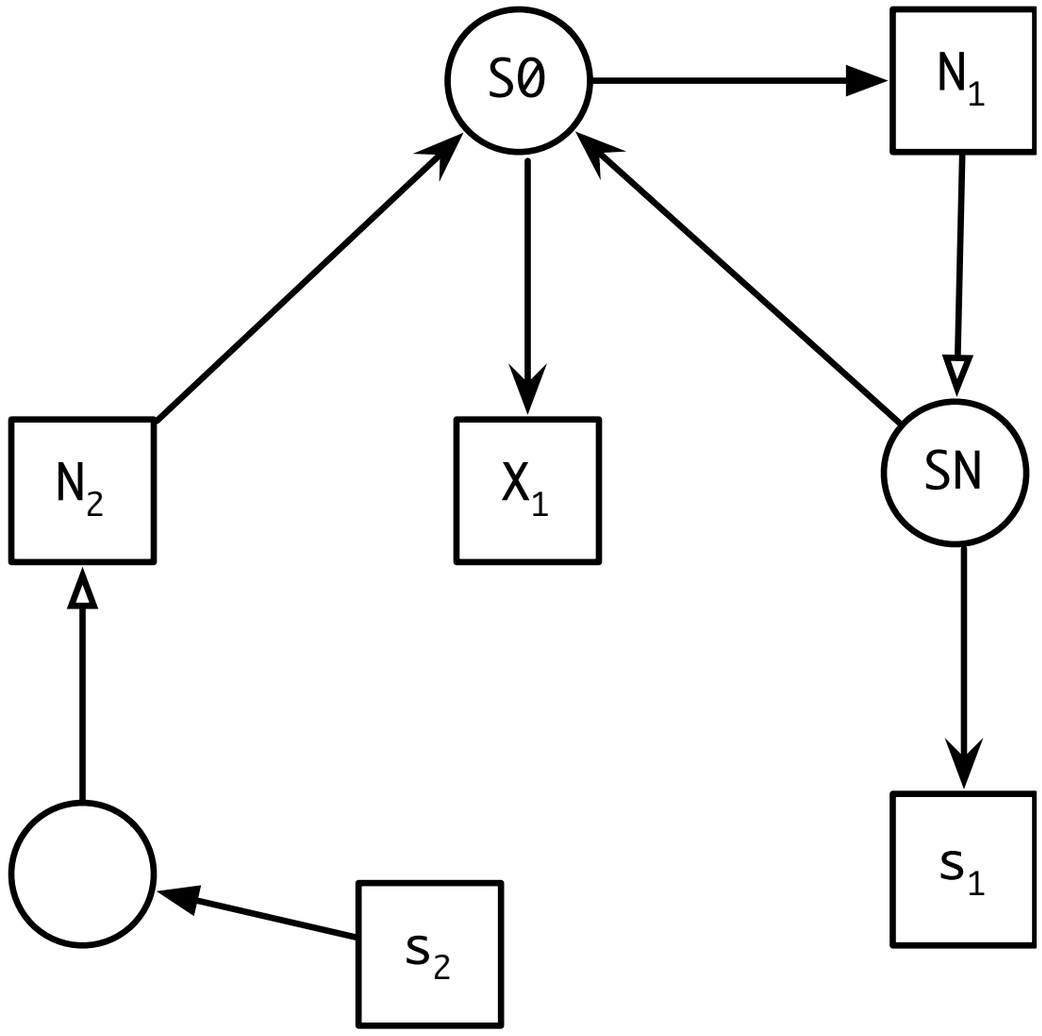
```
module N₁ {
    def s₁ = 5
}

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

Qualified Names



Qualified Names

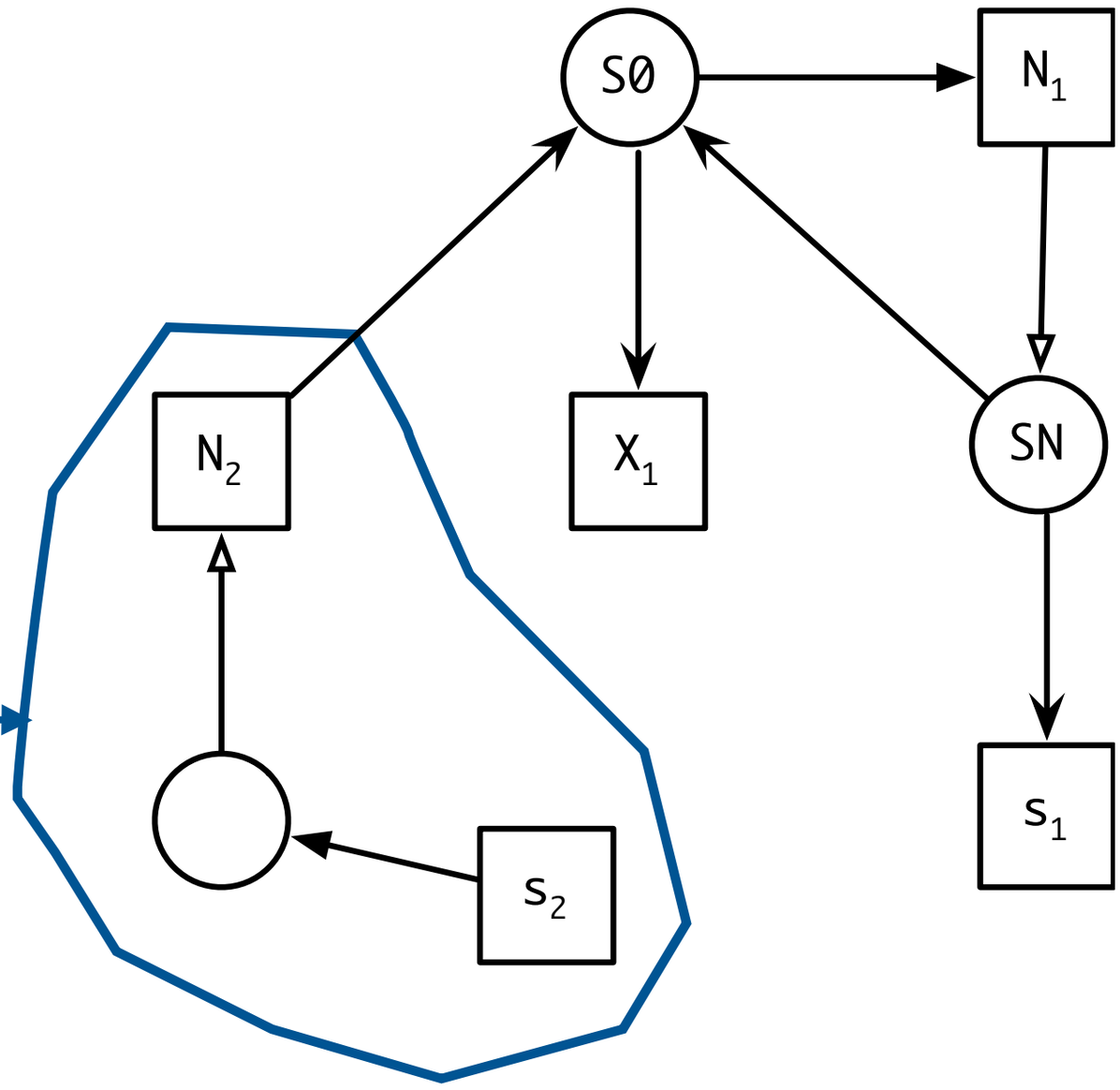
```
module N1 {                                    S0
     def s1 = 5
}

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



Qualified Names

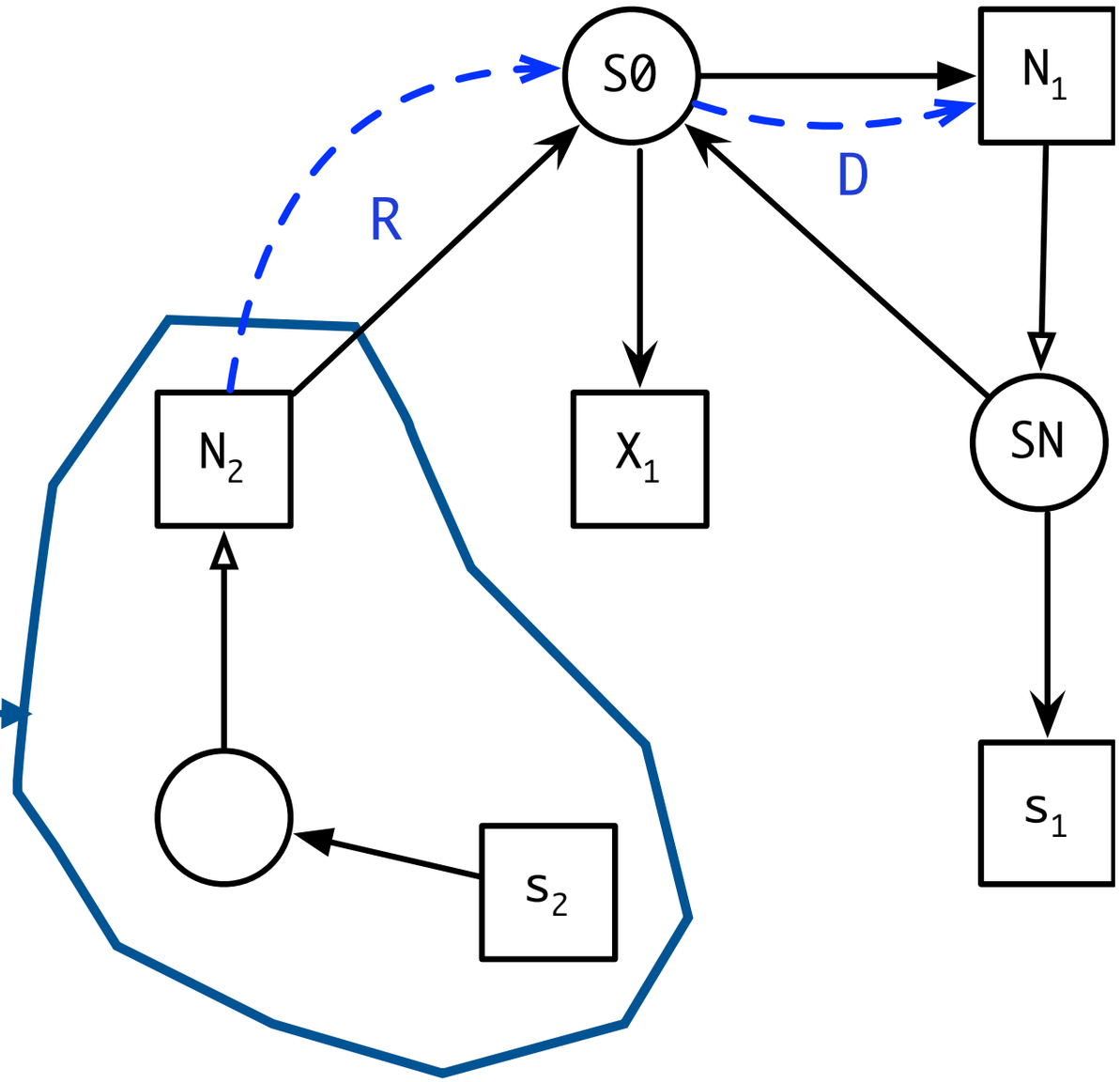
S0

```
module N1 {
     def s1 = 5
}

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



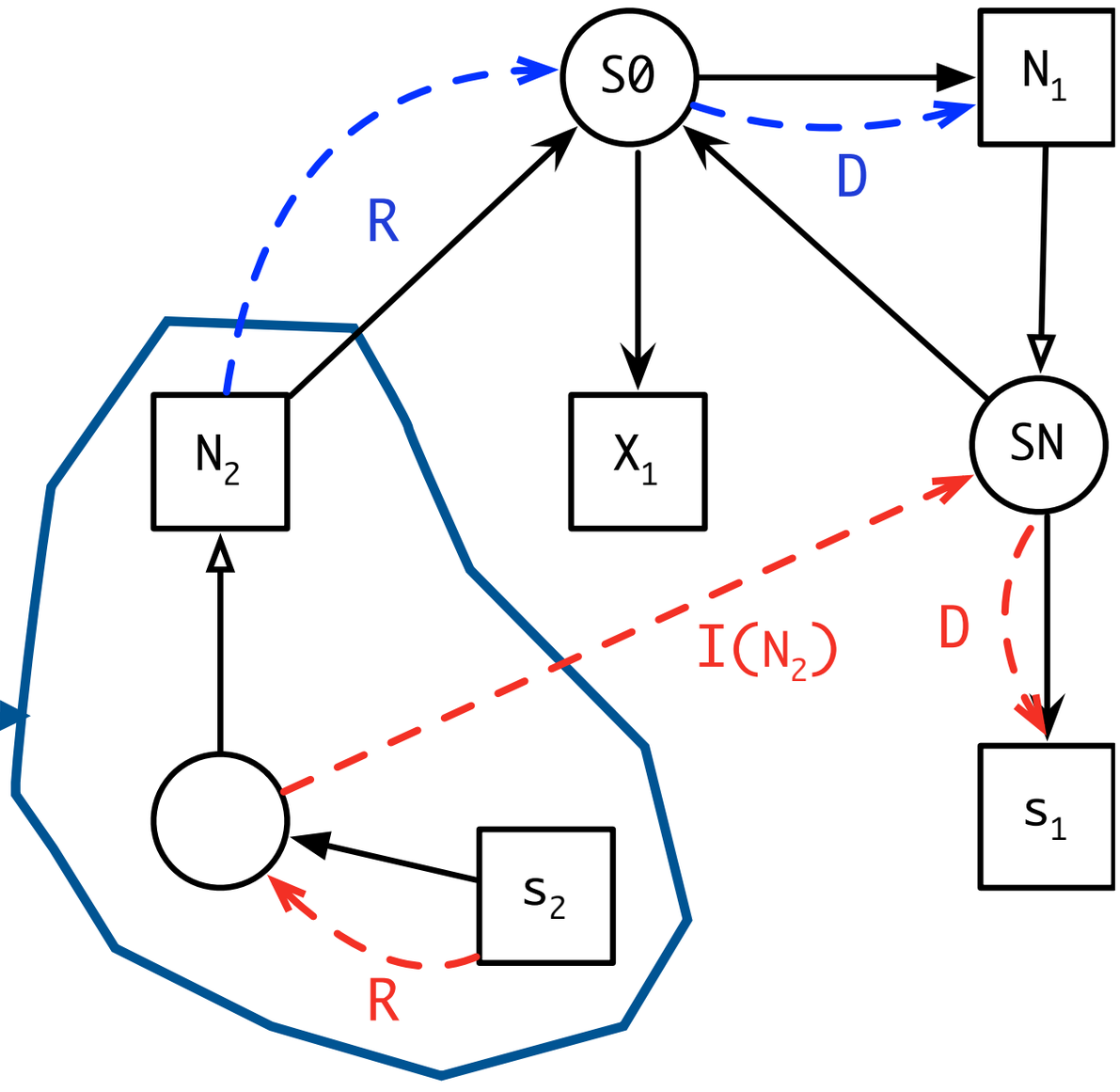
Qualified Names

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

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



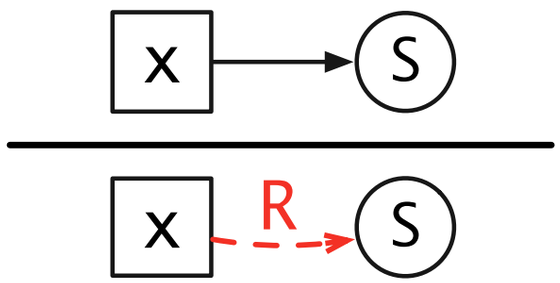
Qualified Names

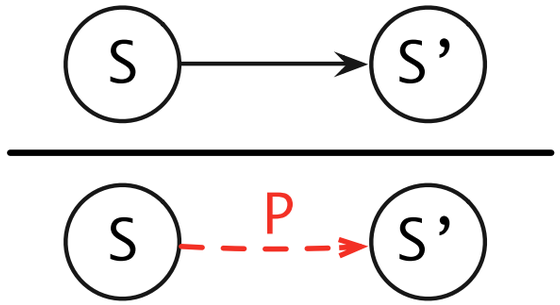
```
module N1 {                                    S0
     def s1 = 5
}

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

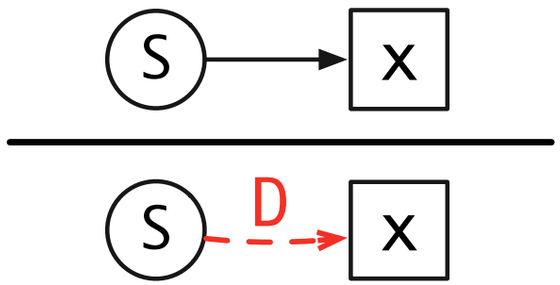


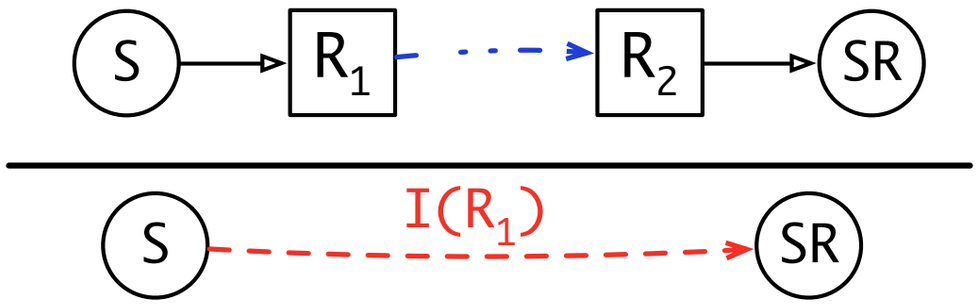
A Calculus for Name Resolution

Reachability of declarations from references
through scope graph edges









How about ambiguities?
References with multiple paths

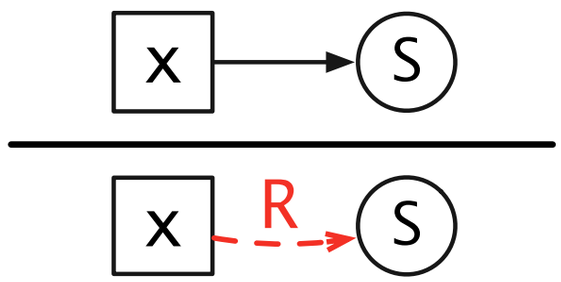
Visibility

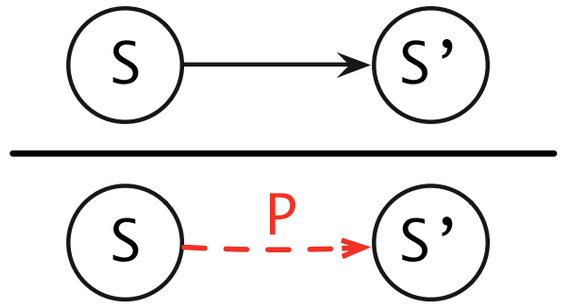
(Disambiguation)

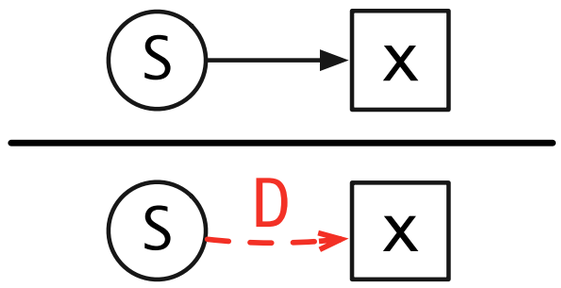
(Slides by Pierre Néron)

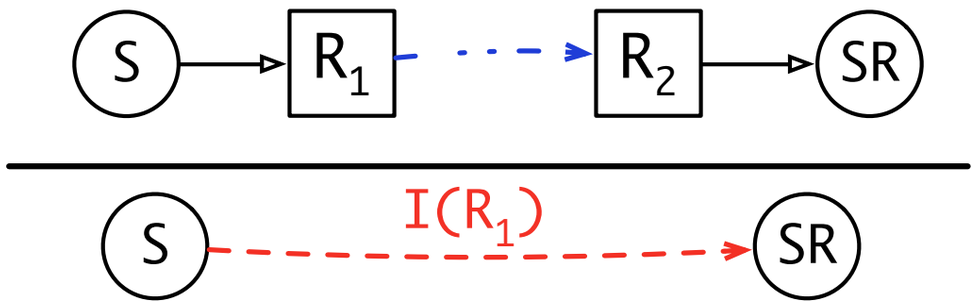
A Calculus for Name Resolution

Reachability









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

Visibility

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

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

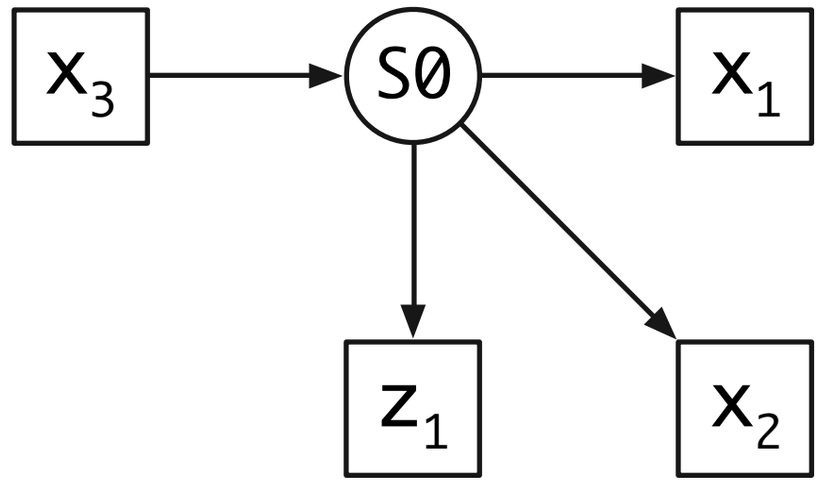
Ambiguous Resolutions

S0

```
def x₁ = 5

def x₂ = 3

def z₁ = x₃ + 1
```

Ambiguous Resolutions

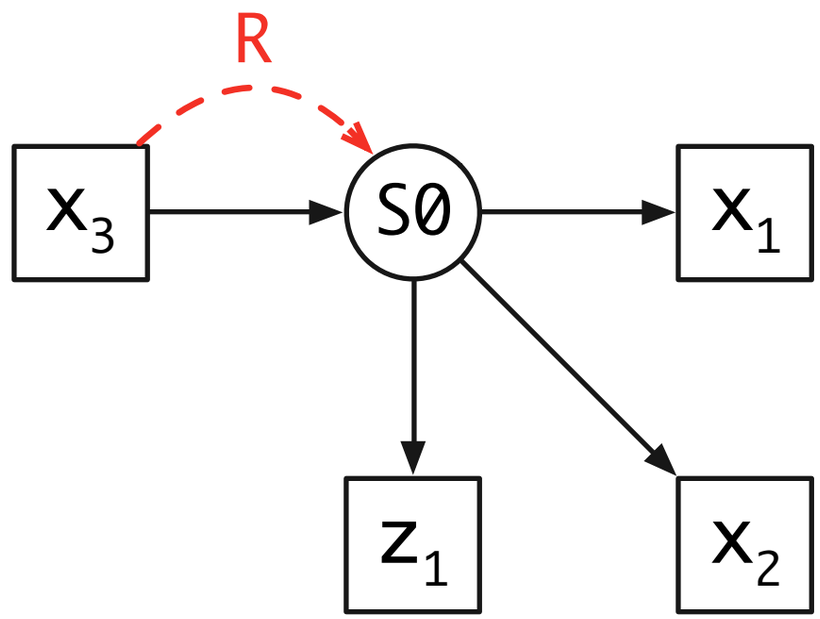
```
S0
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

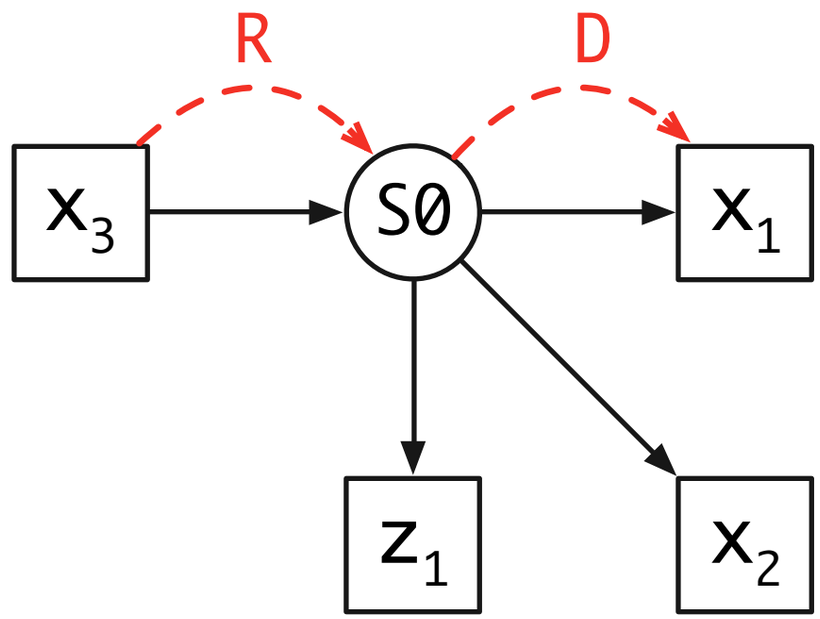
S0

```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



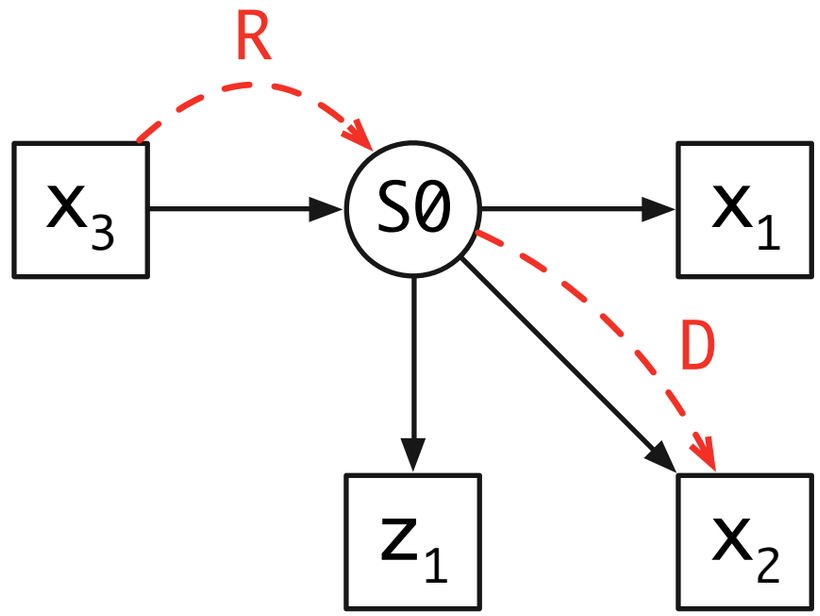
Ambiguous Resolutions

```
S0
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



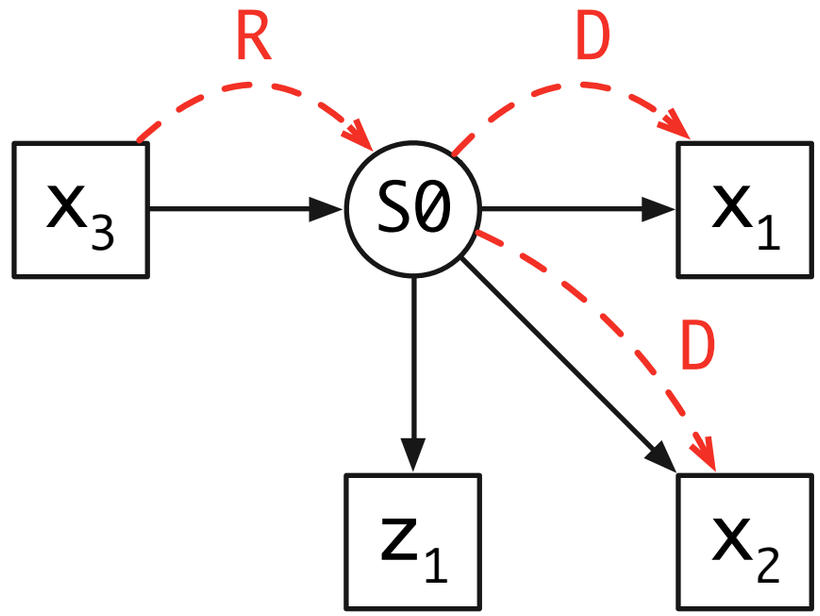
Ambiguous Resolutions

```
def x1 = 5                    S0

def x2 = 3

def z1 = x3 + 1
```



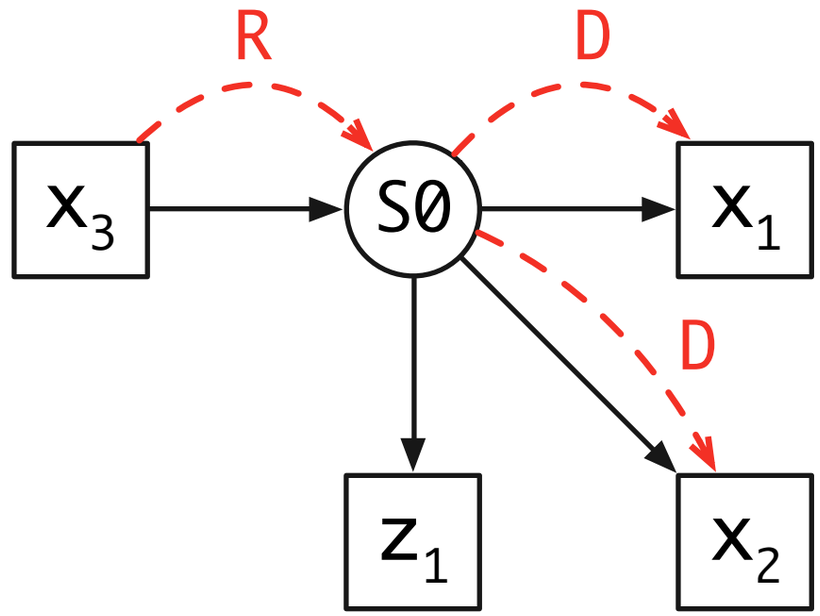
Ambiguous Resolutions

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



Ambiguous Resolutions

```
def x1 = 5                    S0

def x2 = 3

def z1 = x3 + 1
```




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

Shadowing

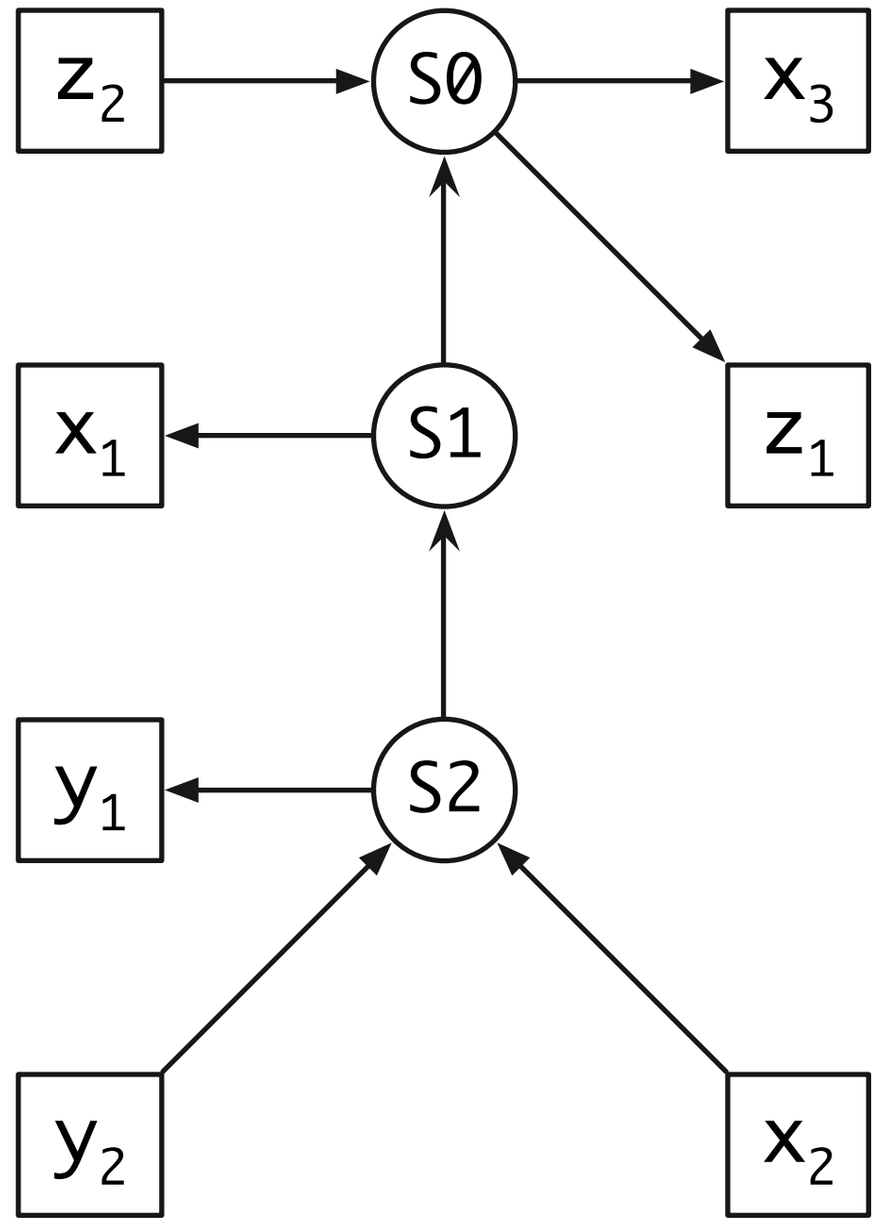
```
def x3 = z2 5 7

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

Shadowing

```
def x3 = z2 5 7                    S0

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

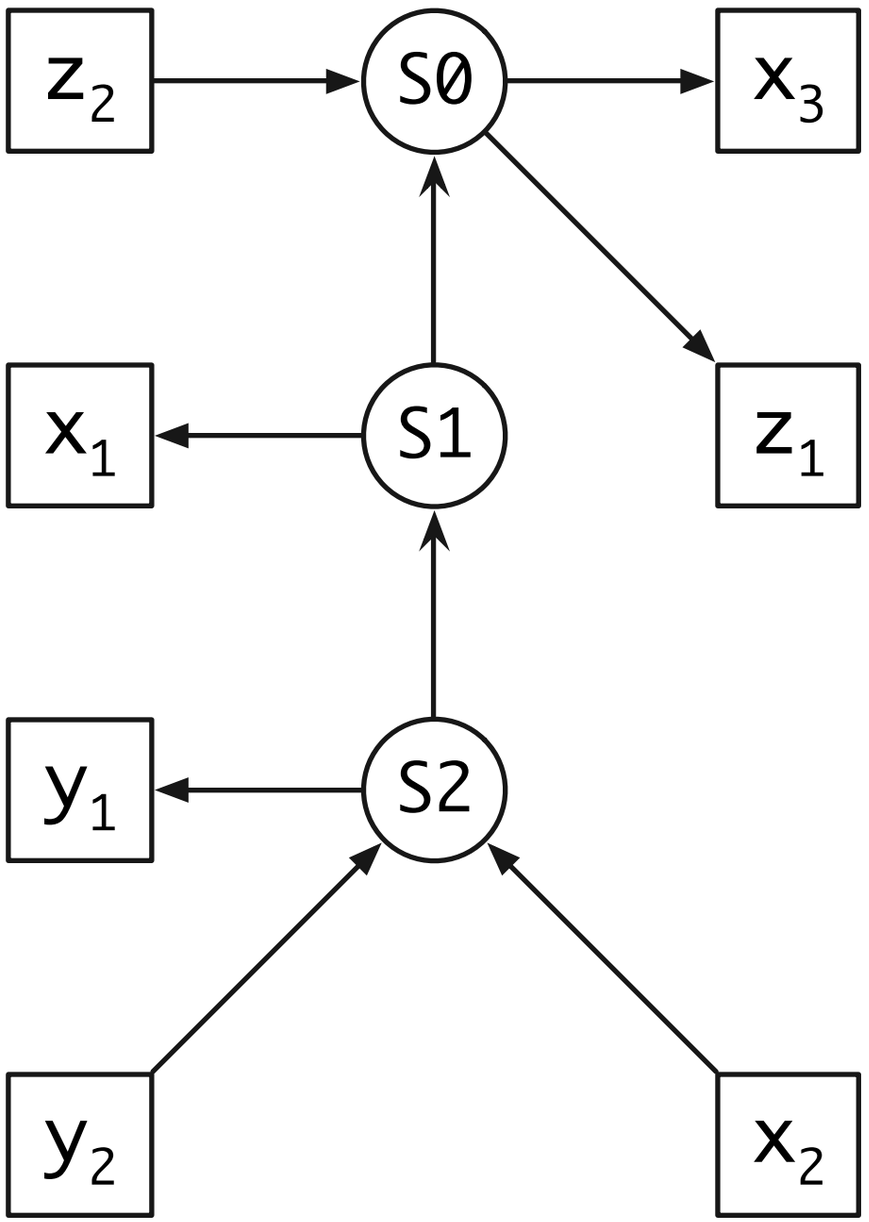
Shadowing

```
def x3 = z2 5 7                    S0

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



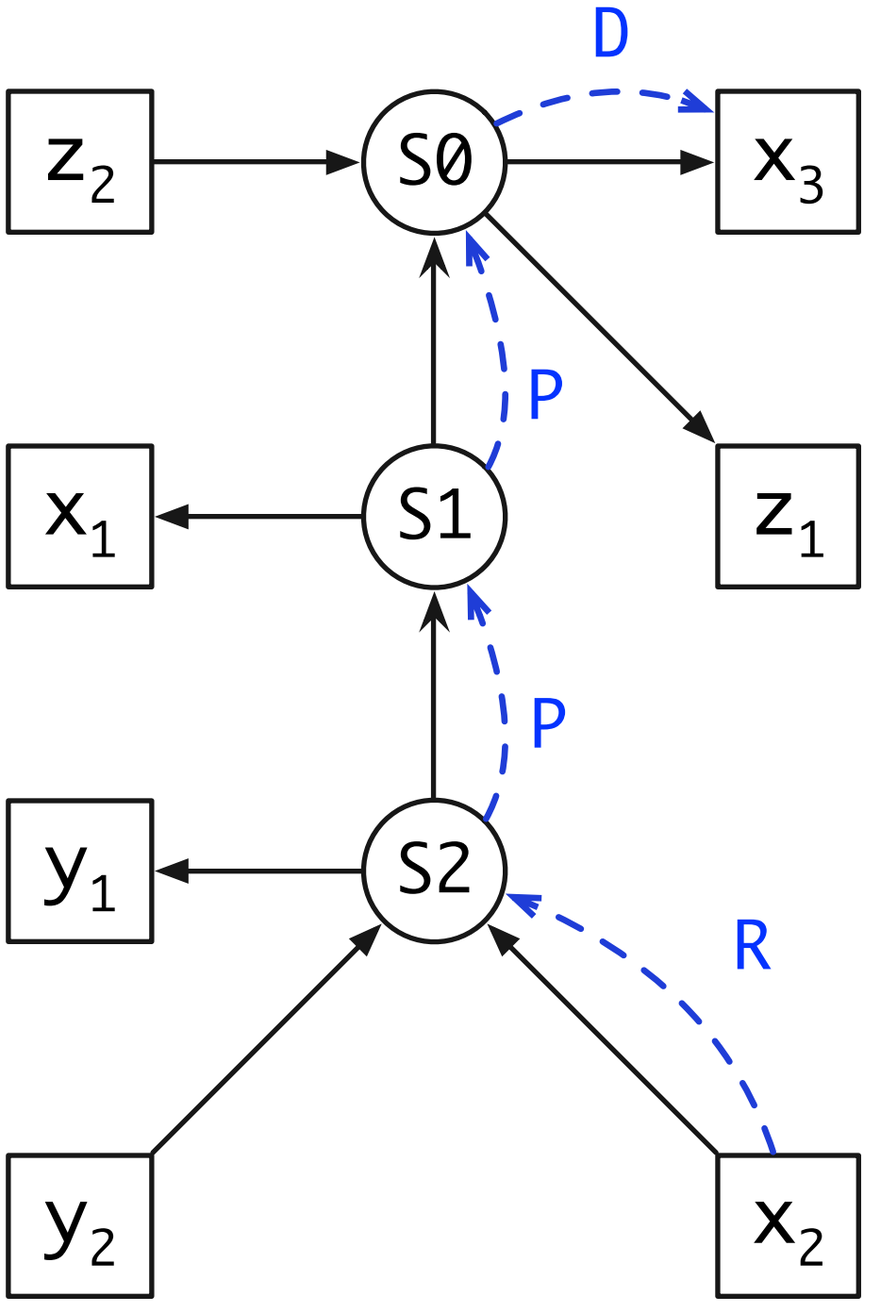
Shadowing

```
def x3 = z2 5 7                    S0

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



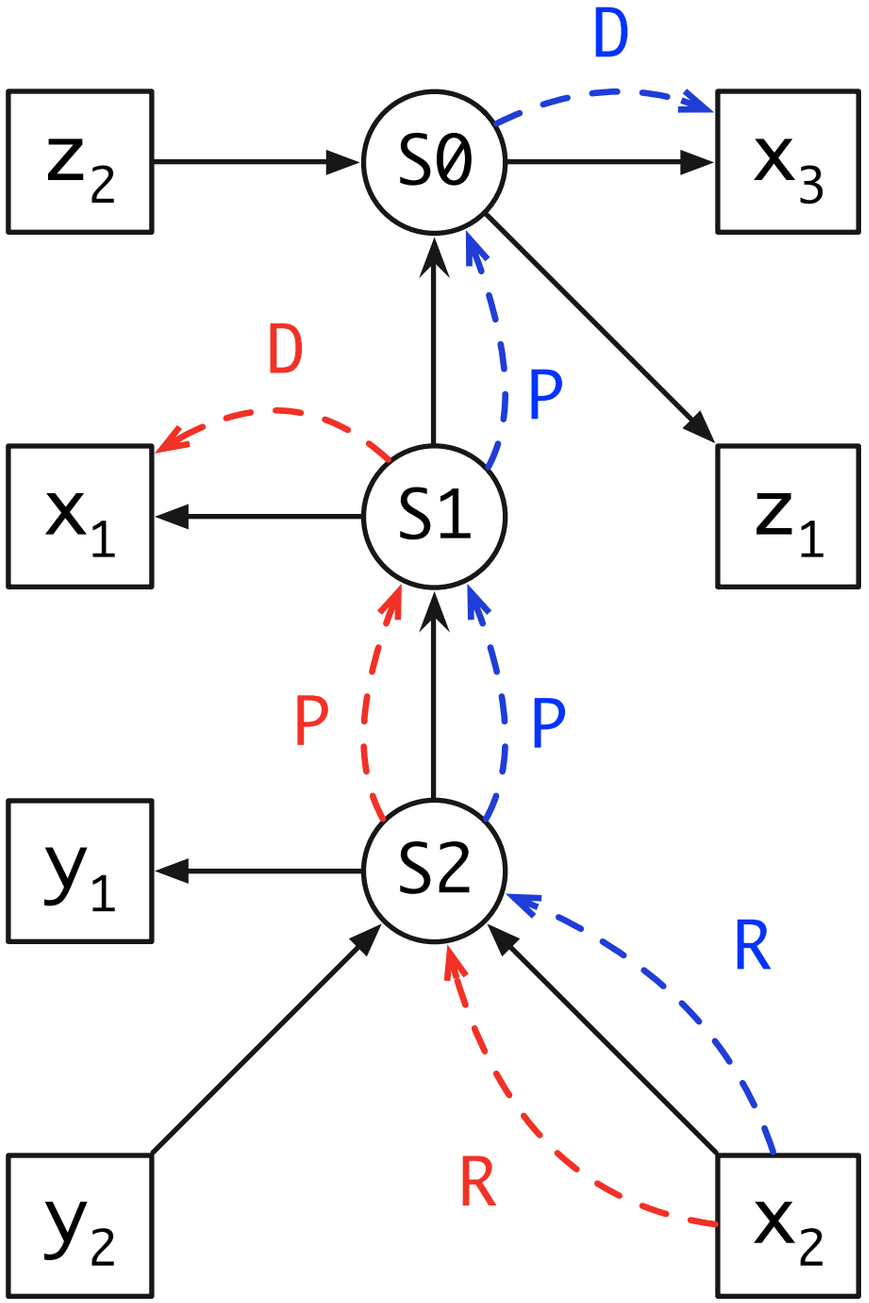
Shadowing

```
def x3 = z2 5 7                    S0

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



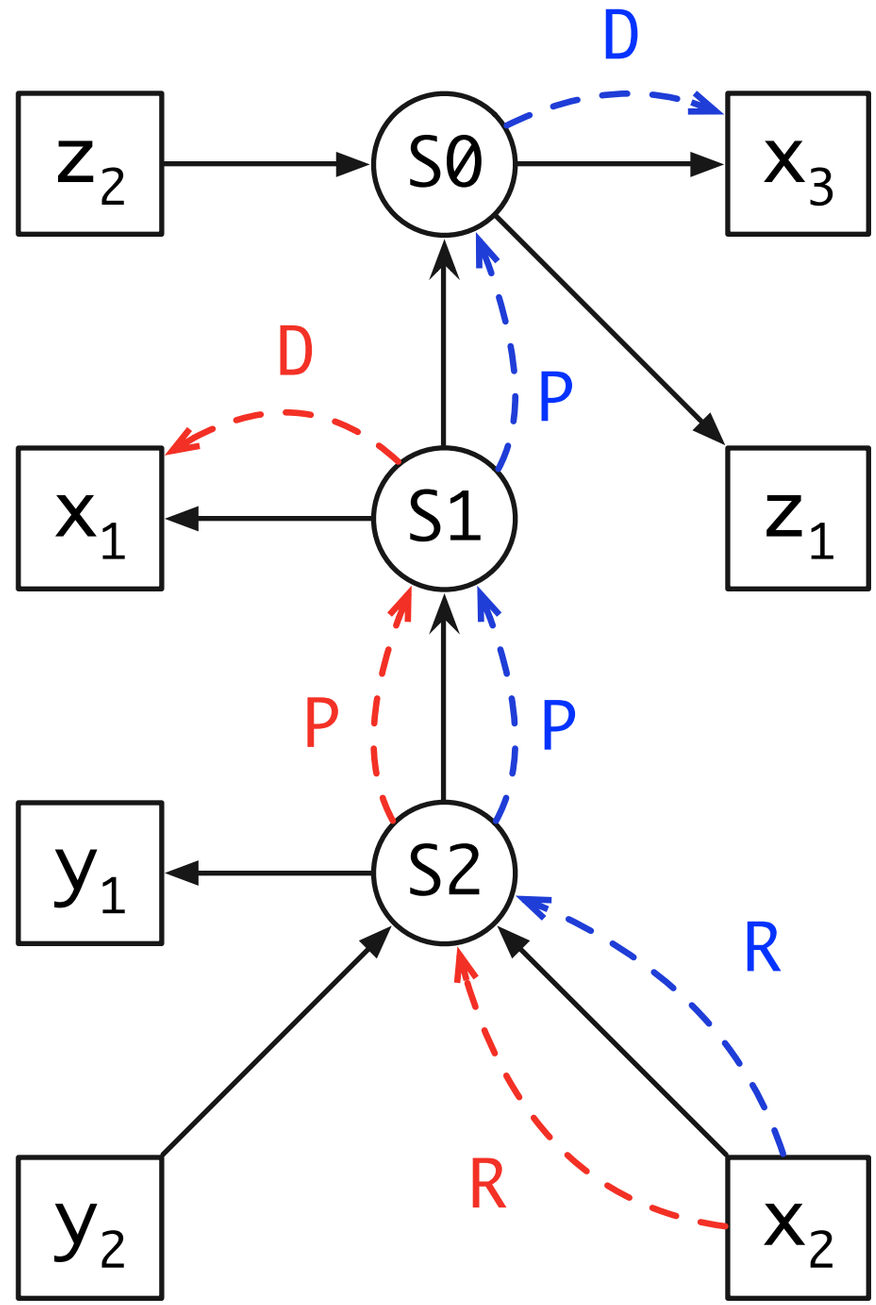
Shadowing

```
def x3 = z2 5 7                    S0

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



Shadowing

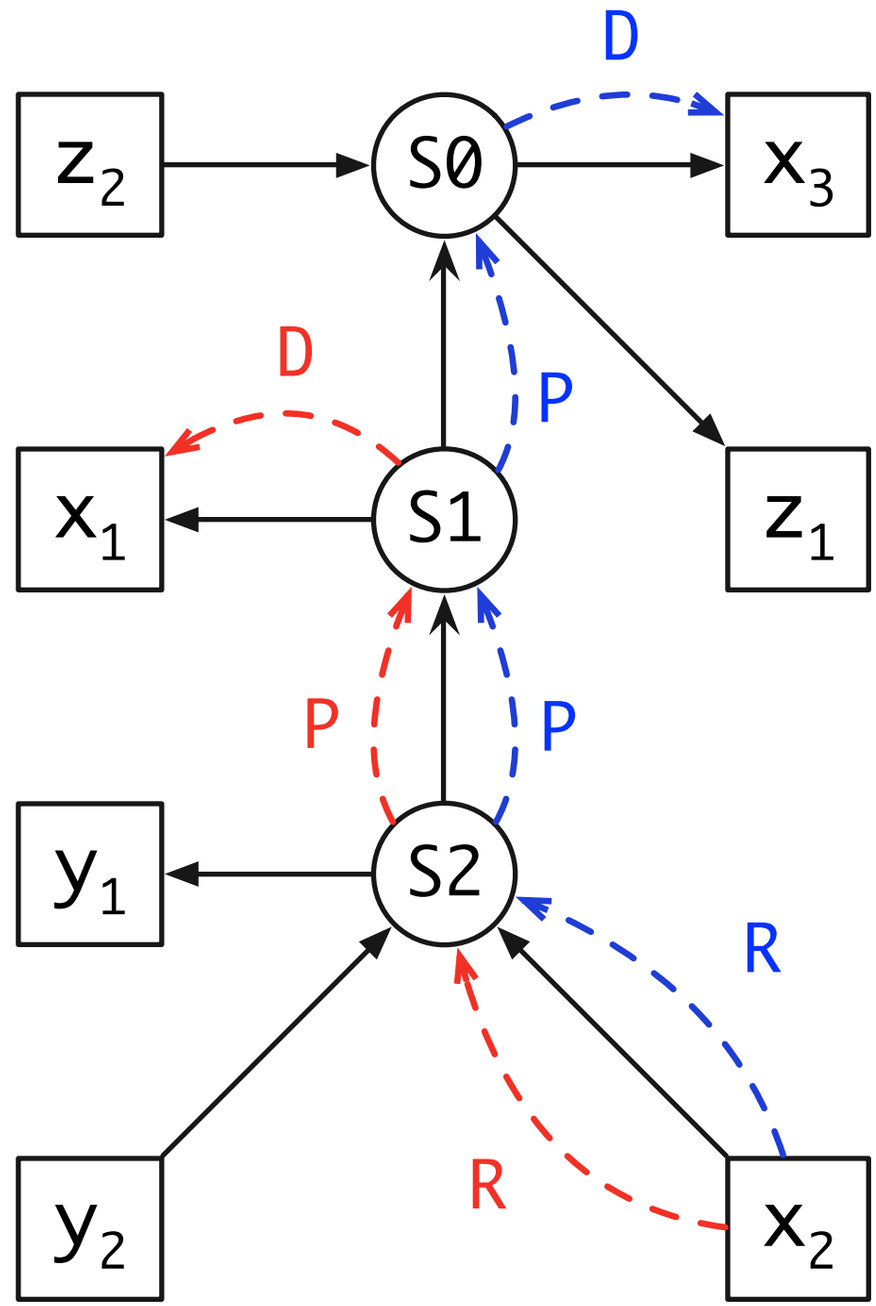
```
def x3 = z2 5 7                    S0

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



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

Shadowing

```
def x3 = z2 5 7                    S0

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

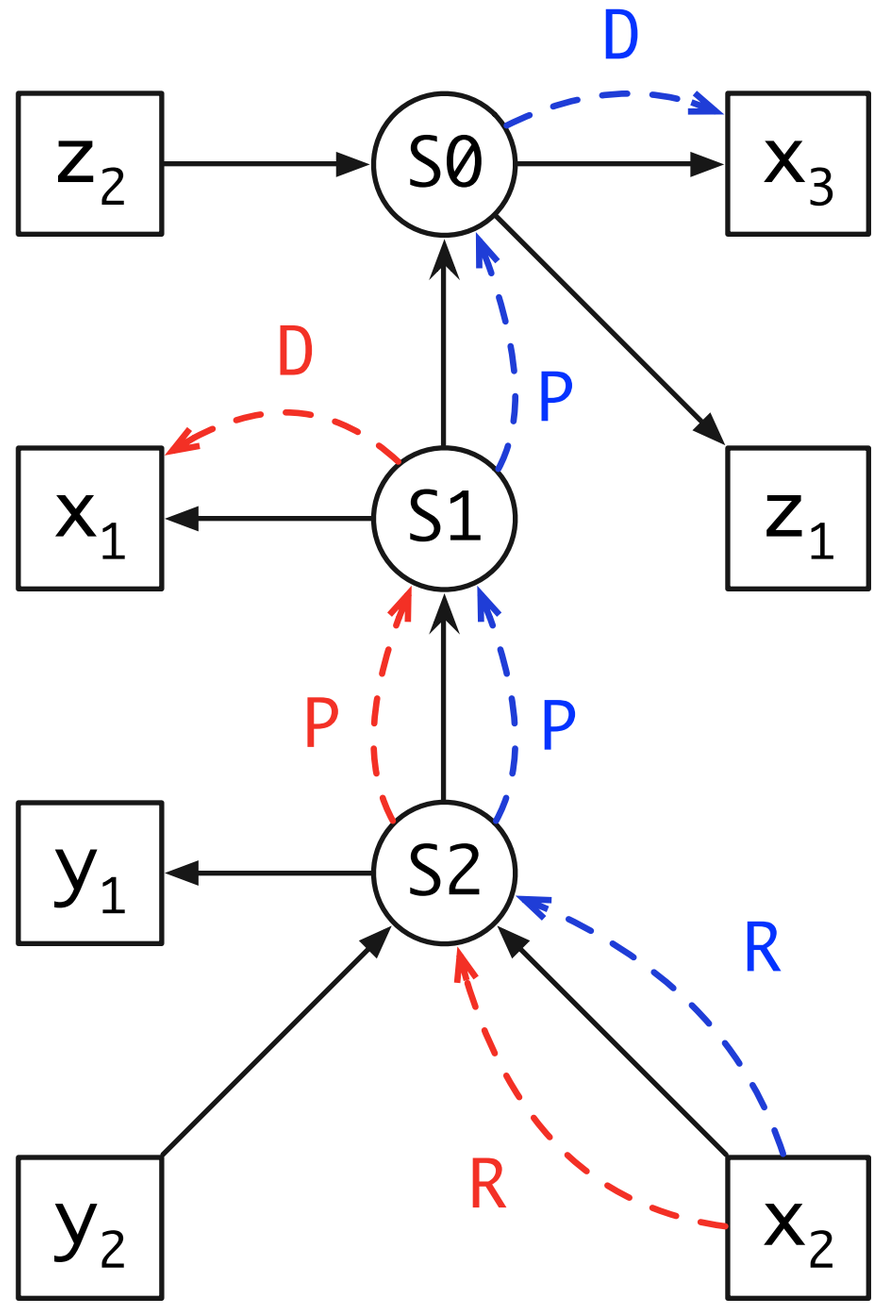


$$\frac{}{D < P.p}$$

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

Shadowing

```
def x3 = z2 5 7                    S0

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



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

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

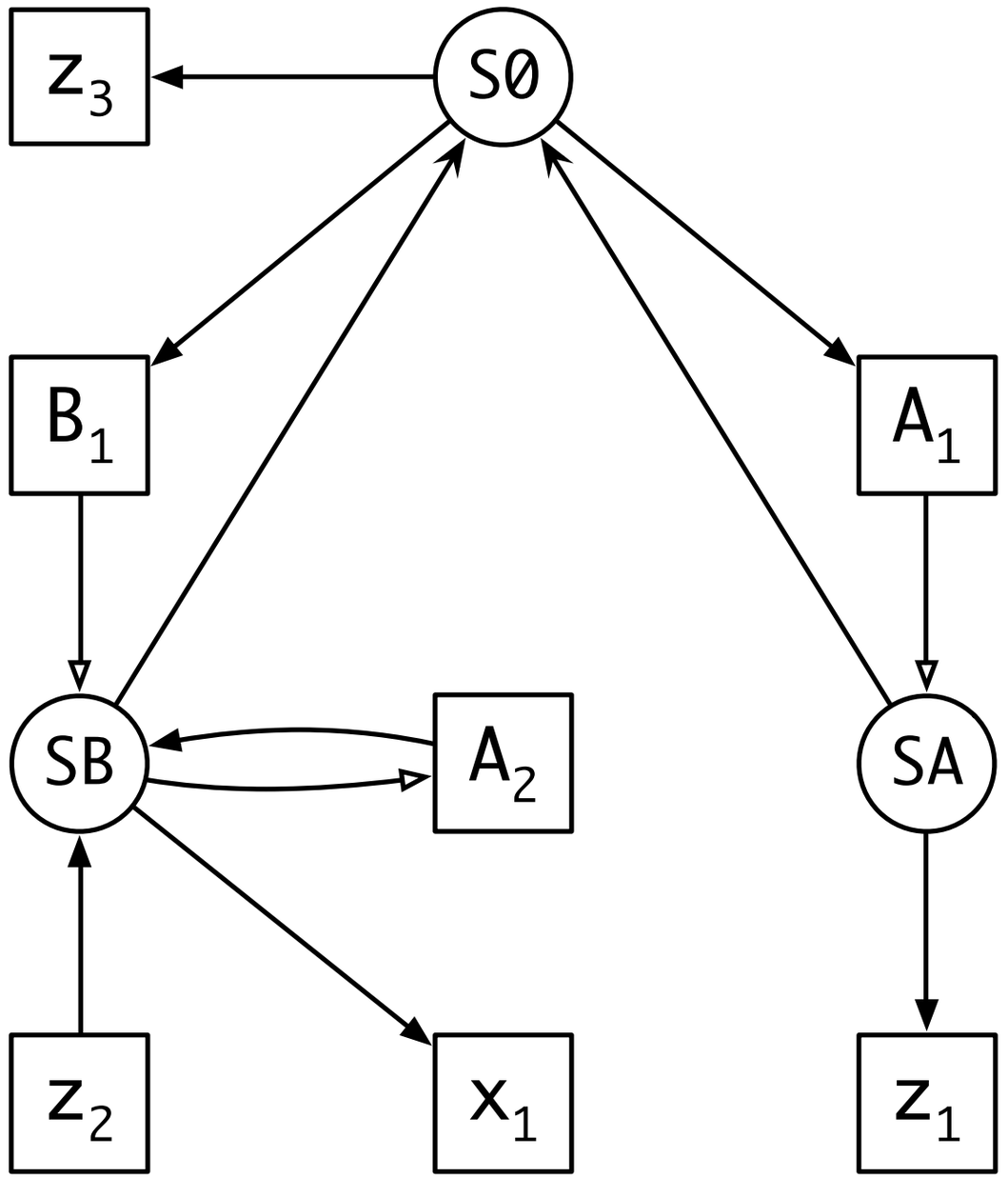
$$R.P.D < R.P.P.D$$

Imports shadow Parents

```
def z3 = 2                                  S0

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

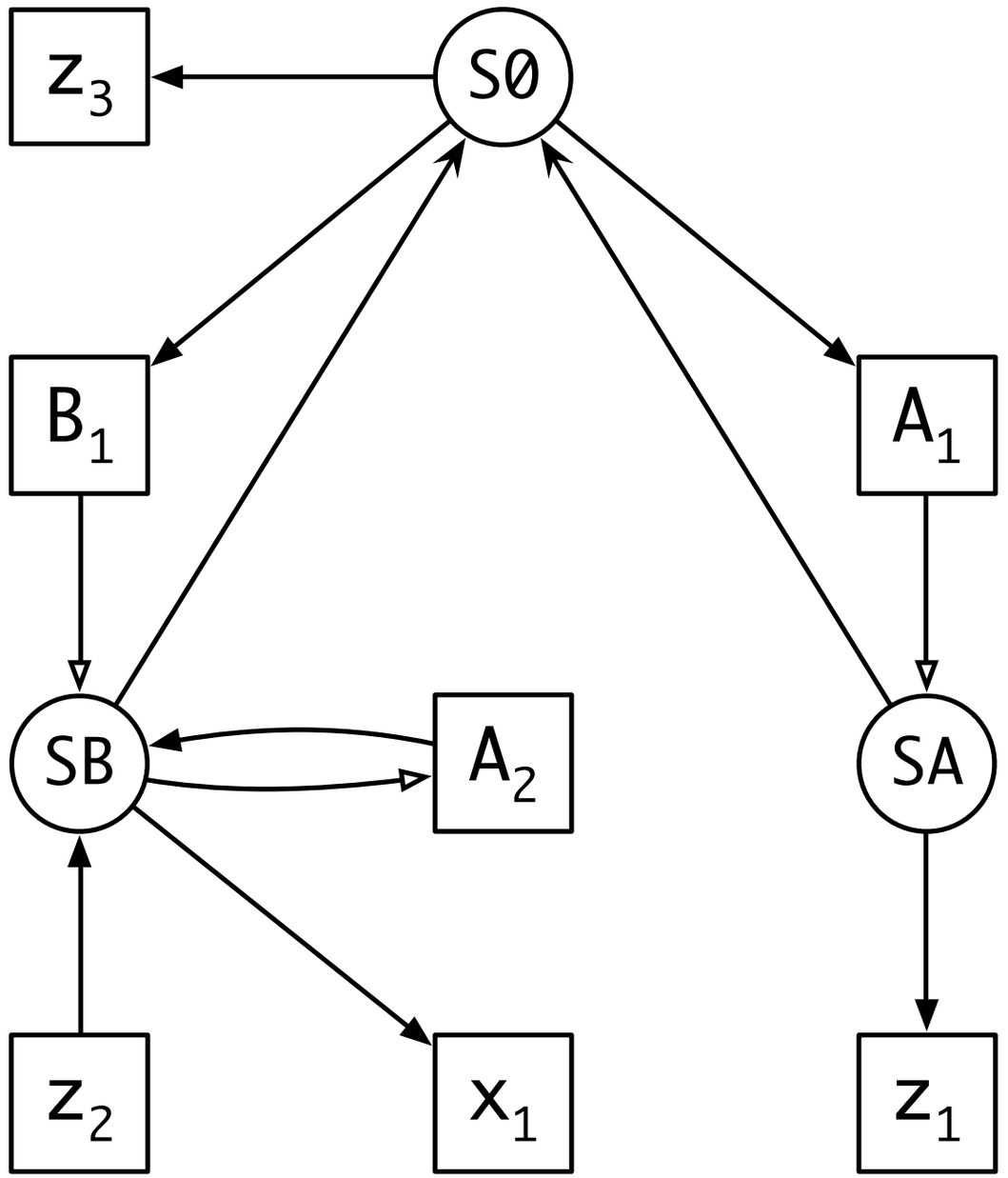
Imports shadow Parents

```
def z₃ = 2                                  S0

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



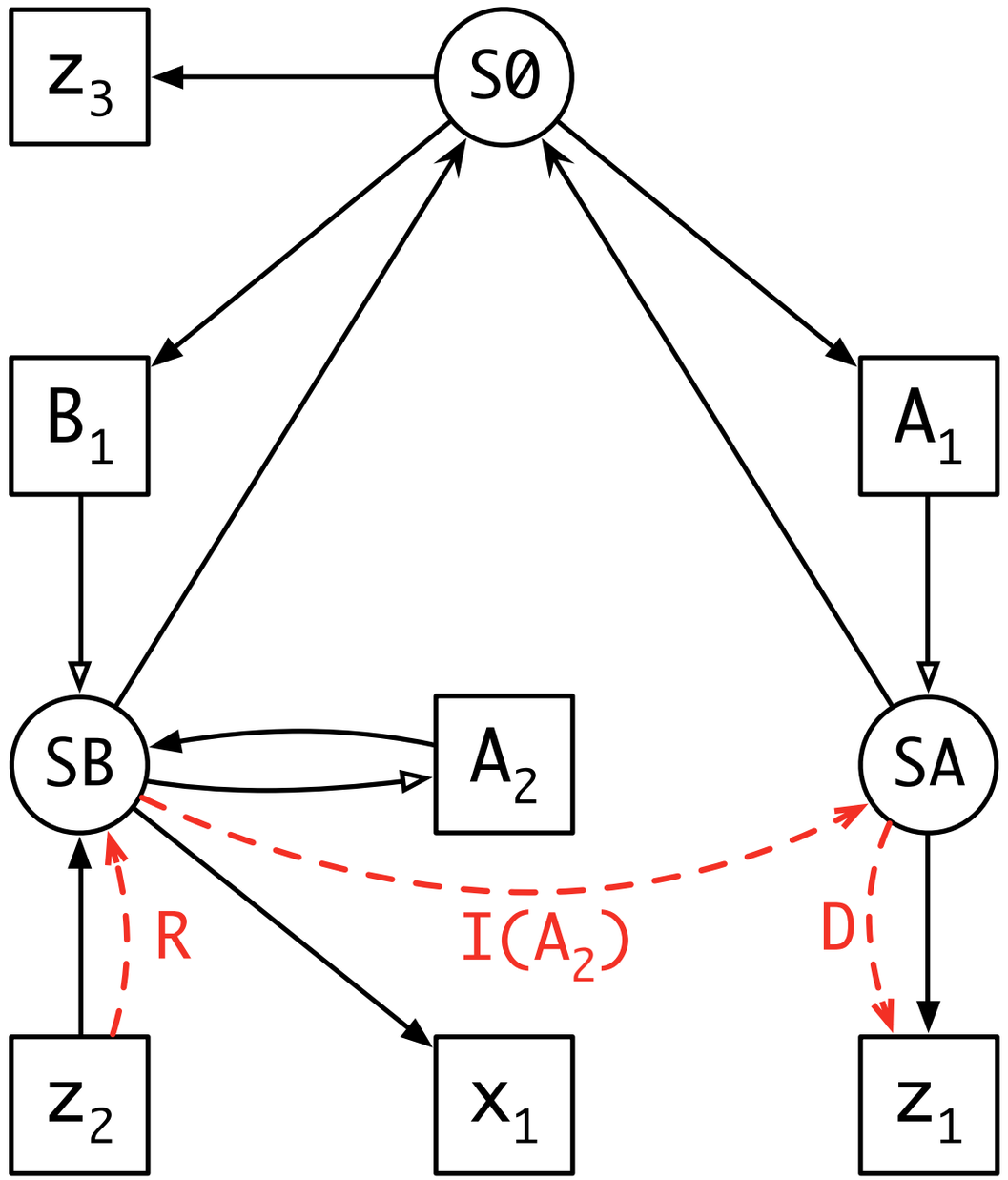
Imports shadow Parents

```
def z₃ = 2                                  S0

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



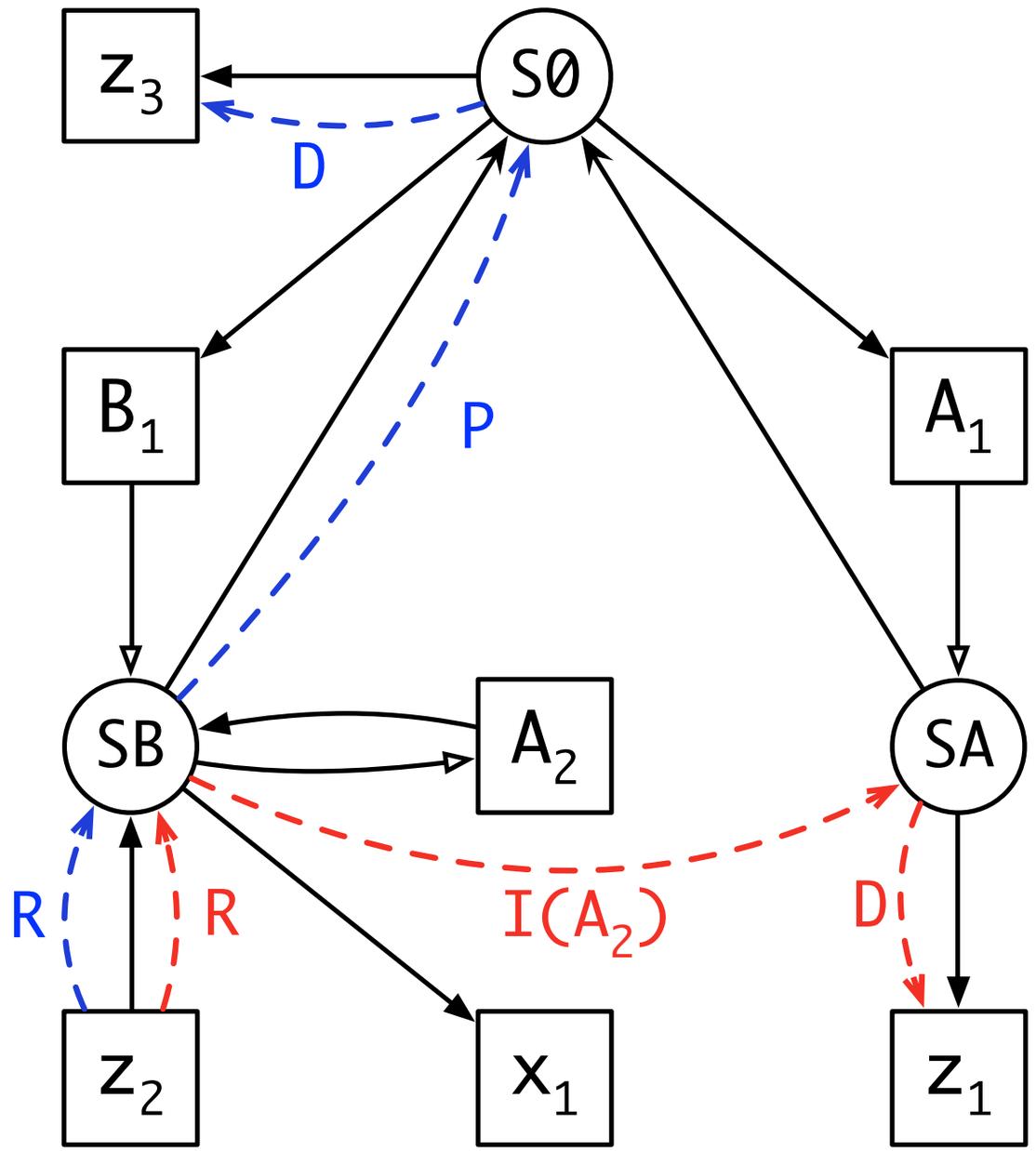
Imports shadow Parents

```
def z₃ = 2                                  S0

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



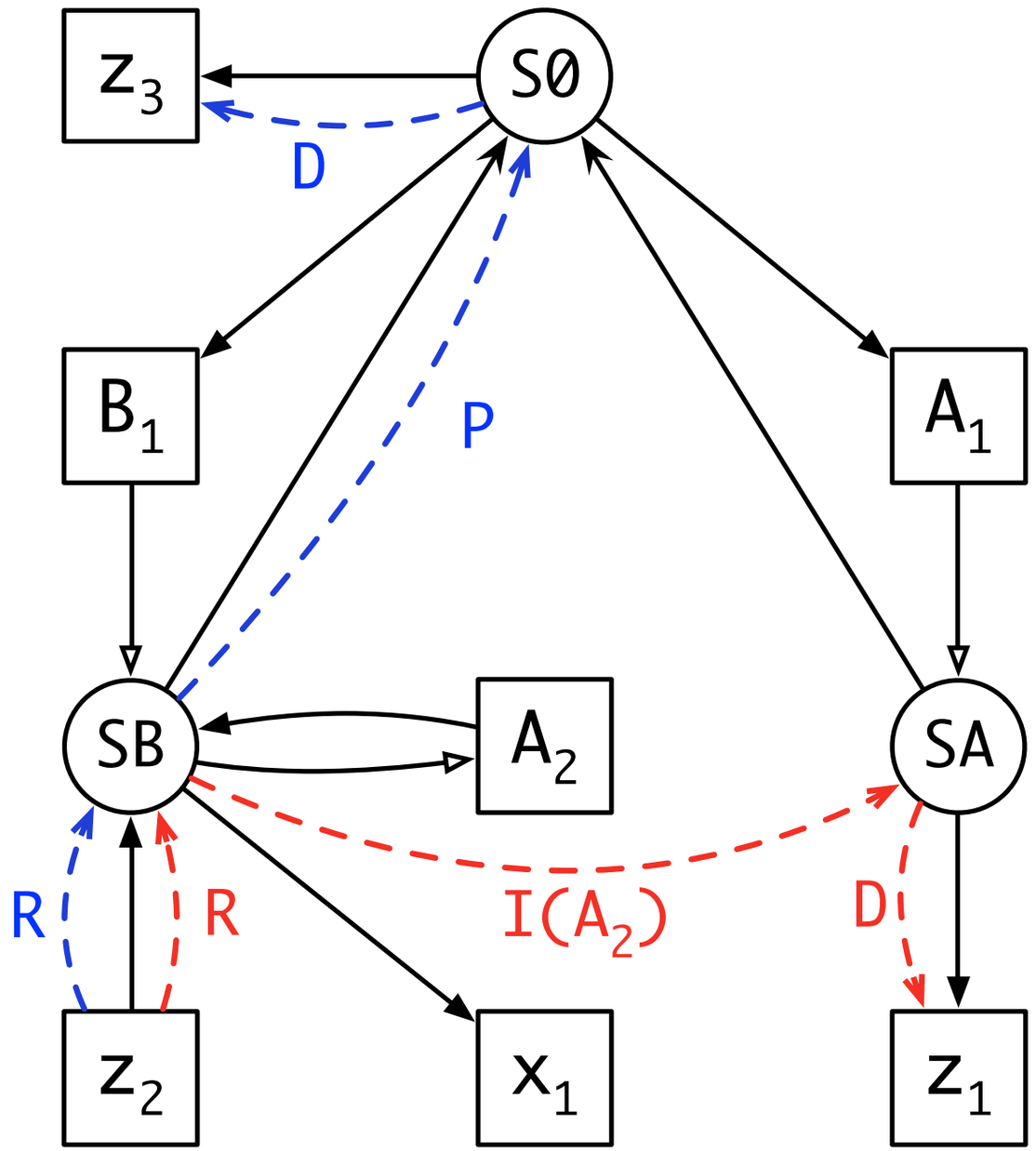
Imports shadow Parents

```
def z₃ = 2                                  S0

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



Imports shadow Parents

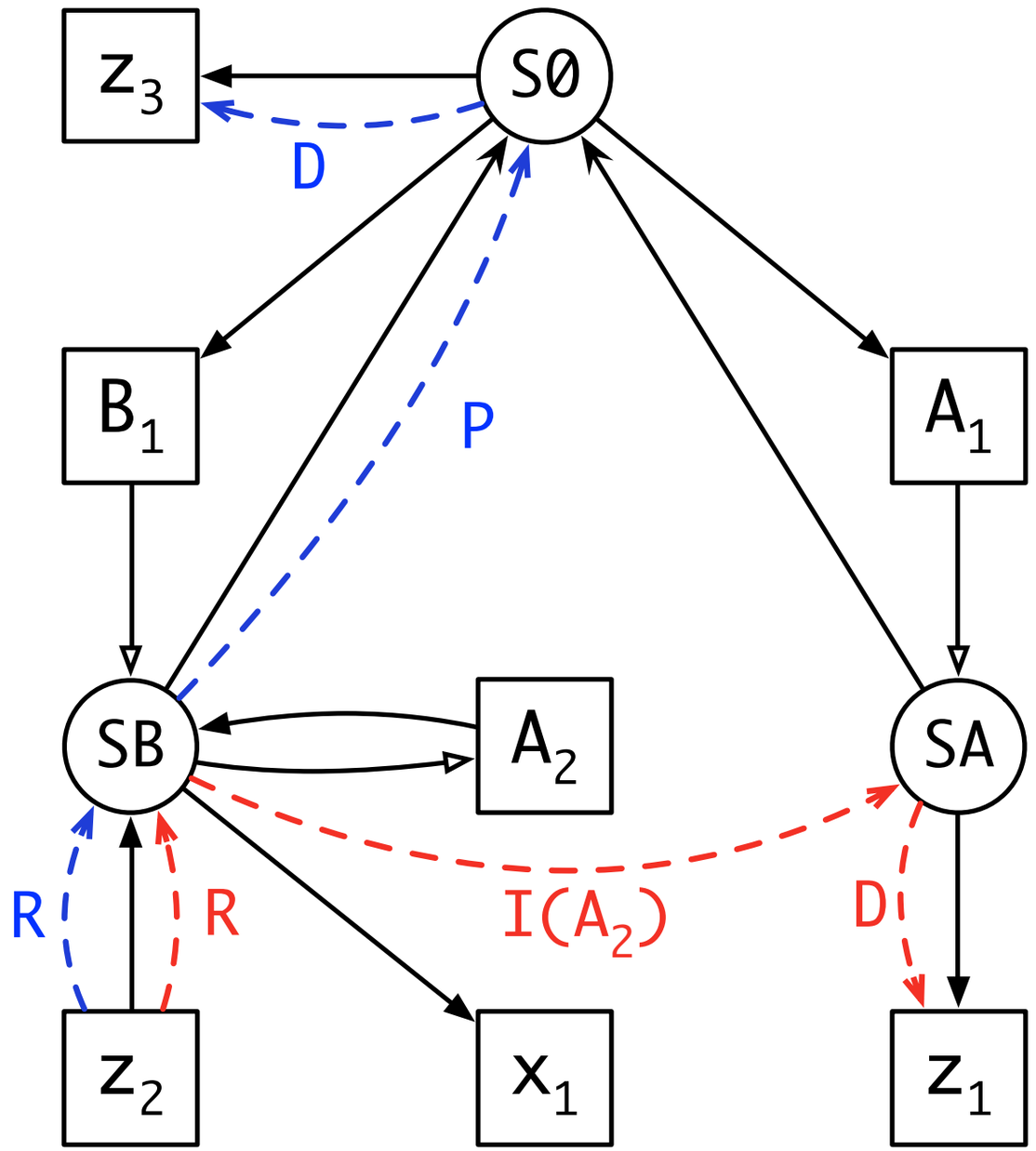
```
def z3 = 2                                  S0

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



I(_).p' < P.p

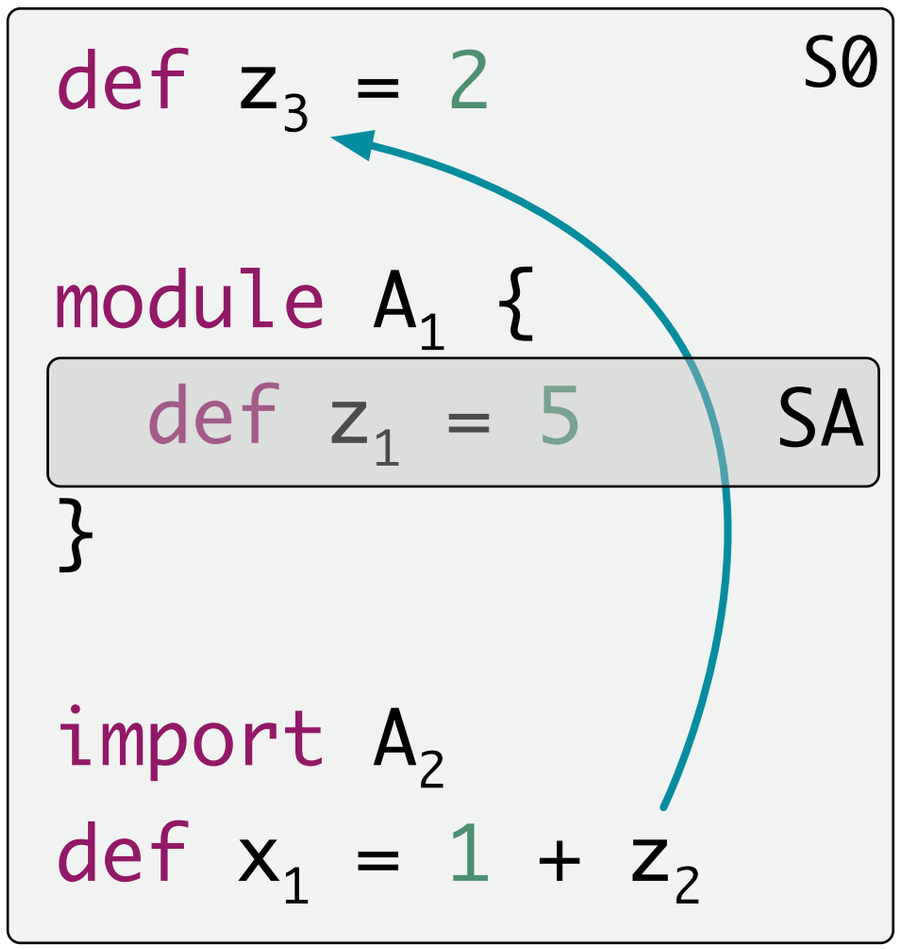
Imports shadow Parents

```
def z3 = 2                                  S0

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



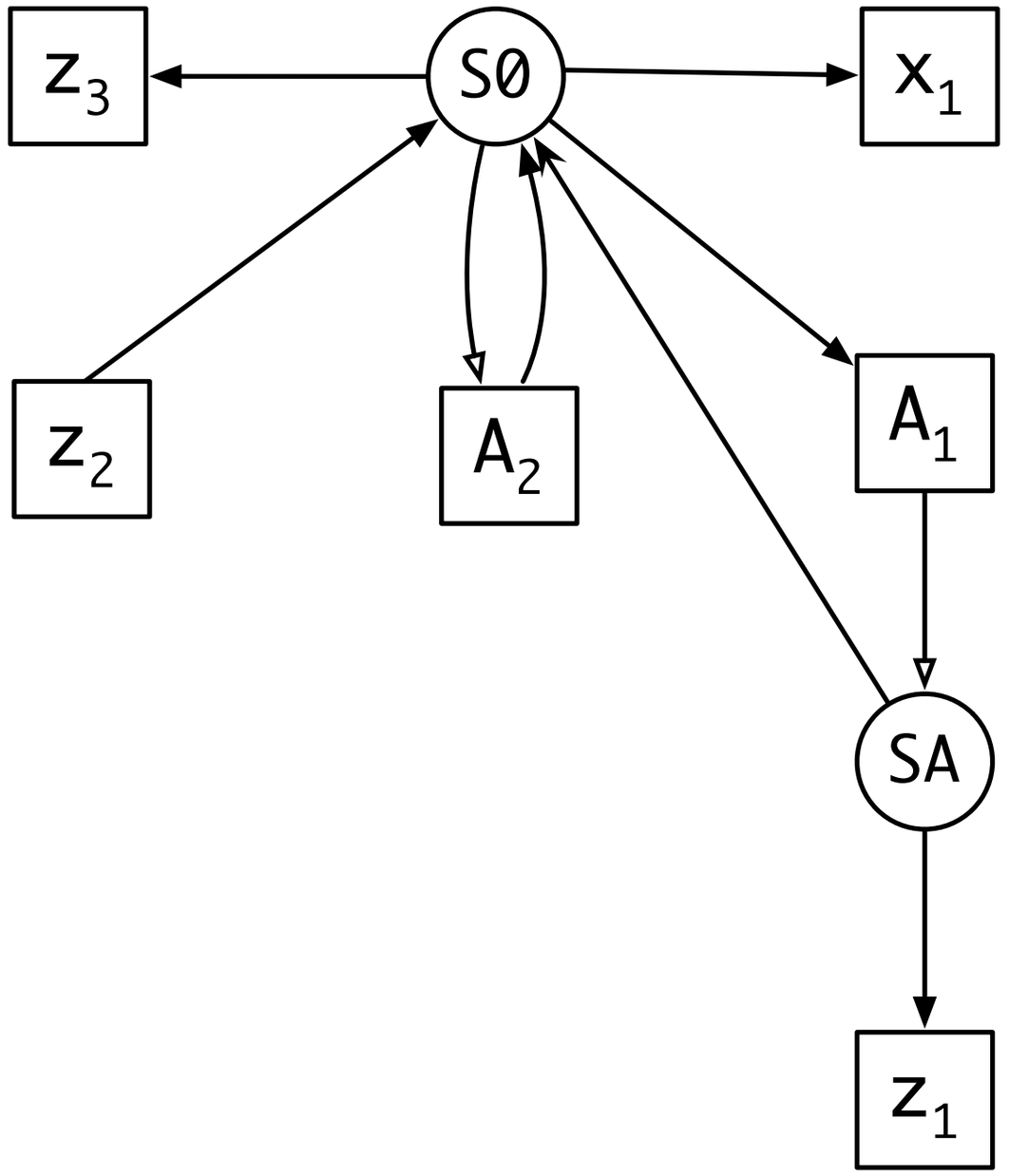
$$\frac{}{\texttt{I(\_).p'} < \texttt{P.p}} \quad \Rightarrow \quad \texttt{R.I(A2).D} < \texttt{R.P.D}$$

Imports vs. Includes



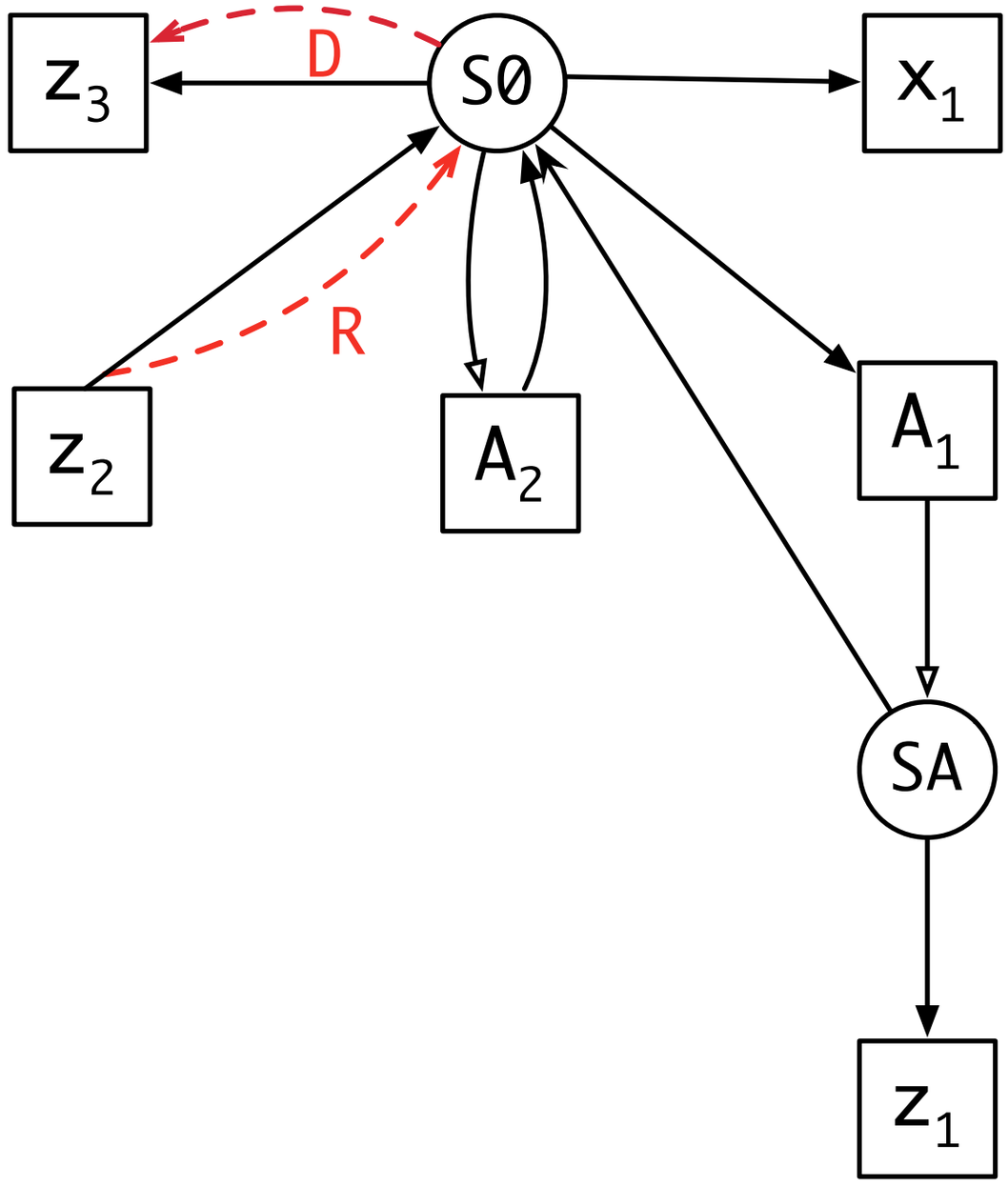
Imports vs. Includes

```
                                    S0
def z3 = 2
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



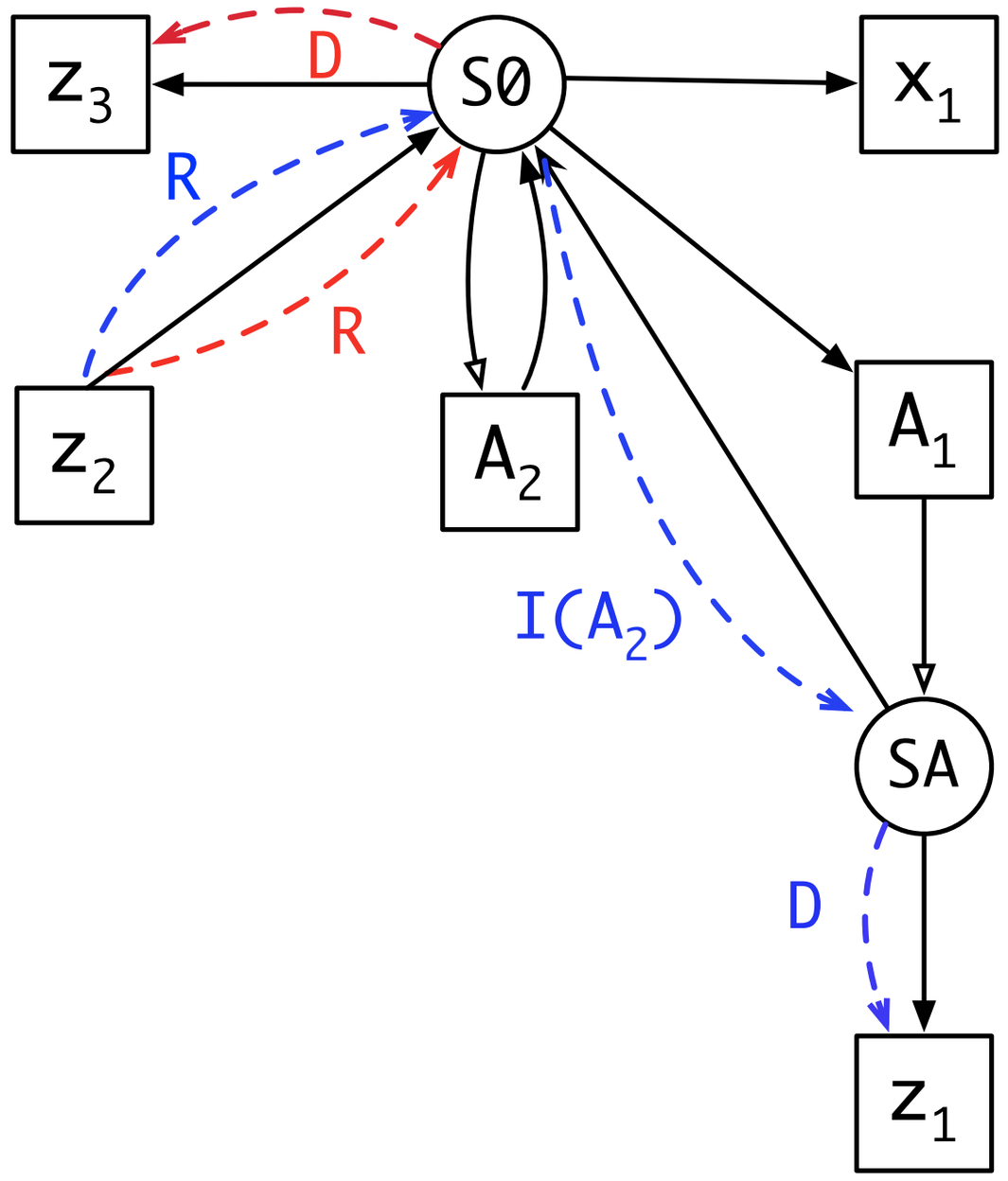
Imports vs. Includes

```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



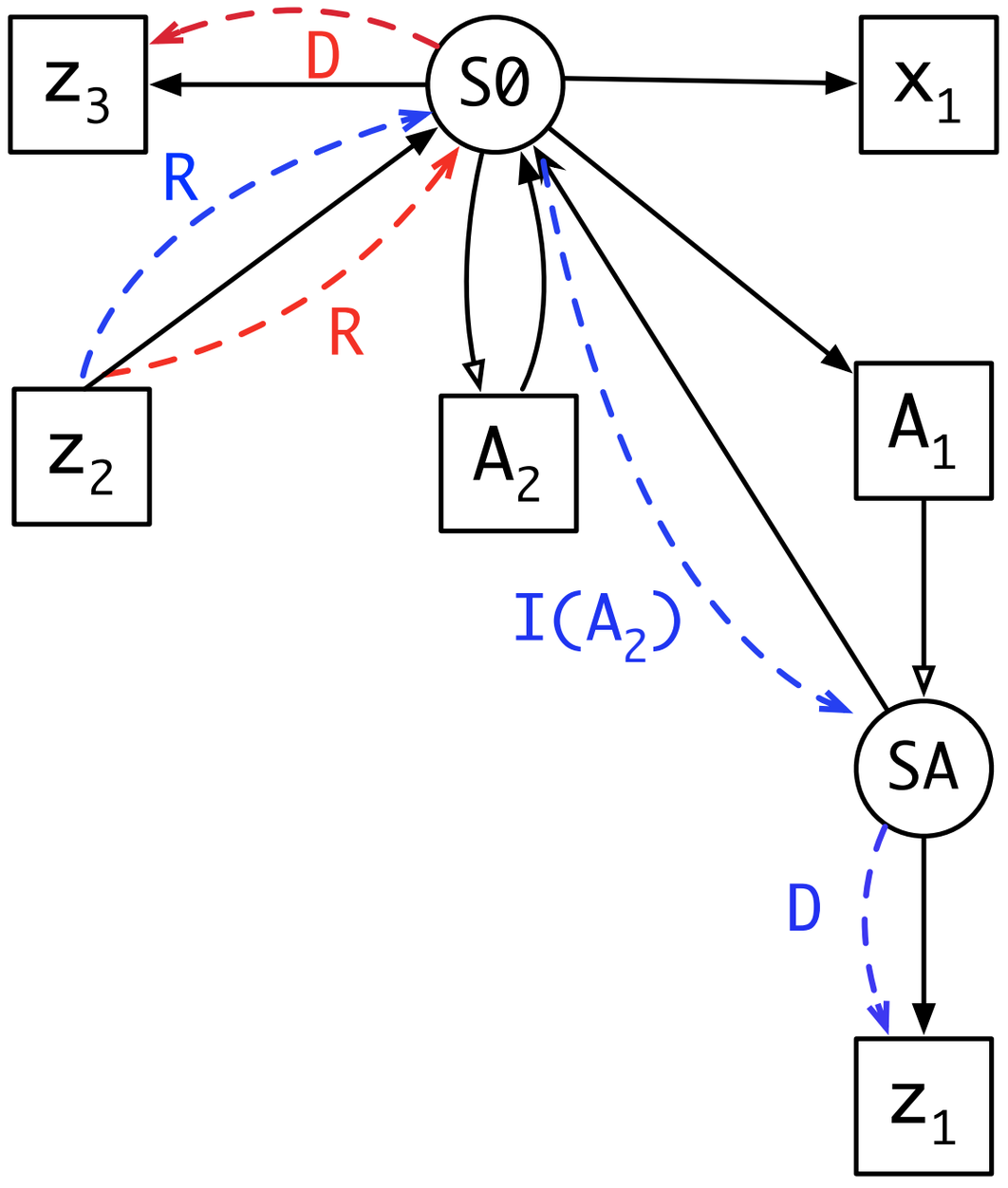
Imports vs. Includes

```
                                  S0
def z3 = 2
   ┌──────────────────────────────┐
module A1 {                      │
  def z1 = 5                  SA │
}  └──────────────────────────────┘


import A2
def x1 = 1 + z2
```



Imports vs. Includes

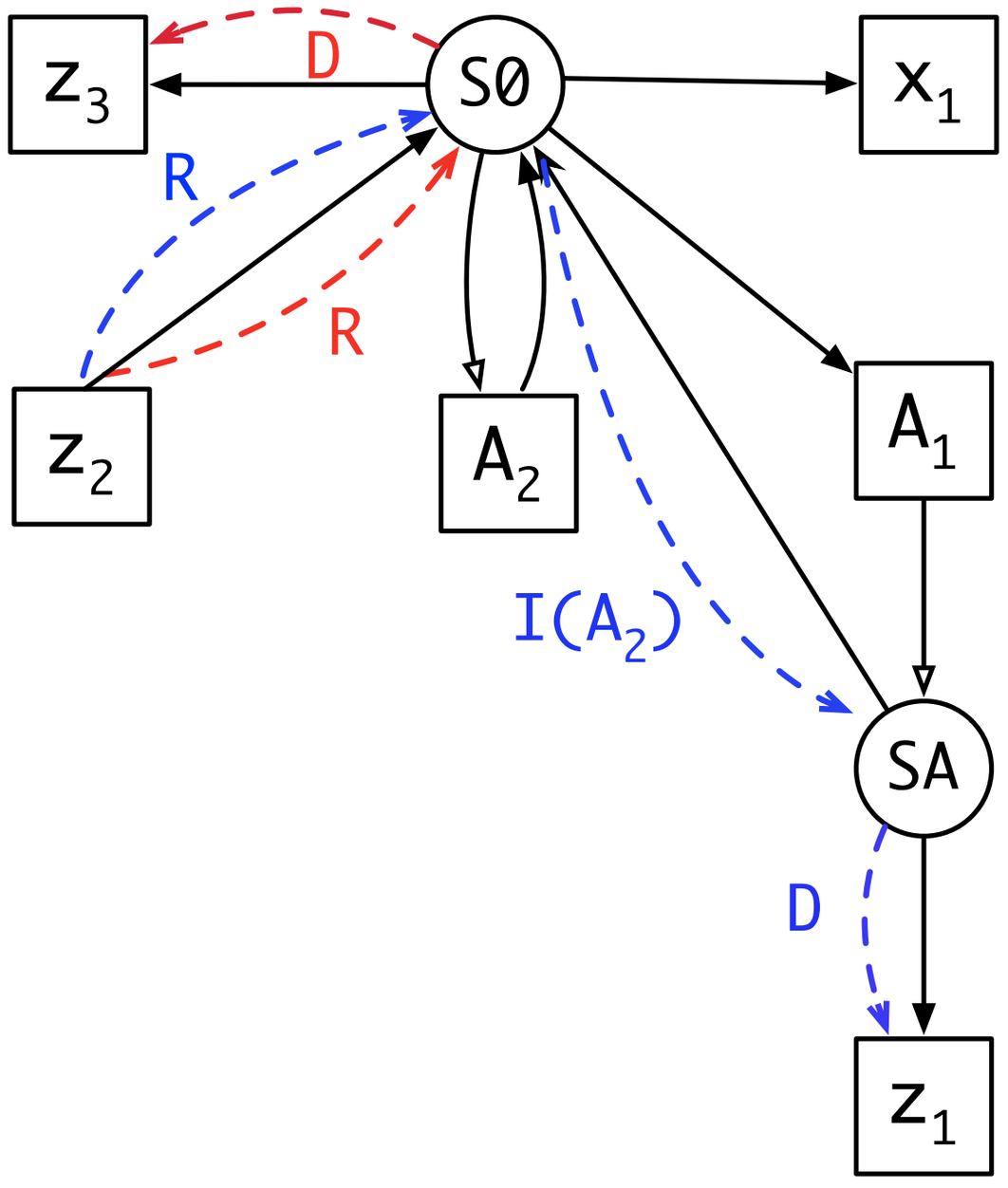
```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



D < I(_).p’

Imports vs. Includes

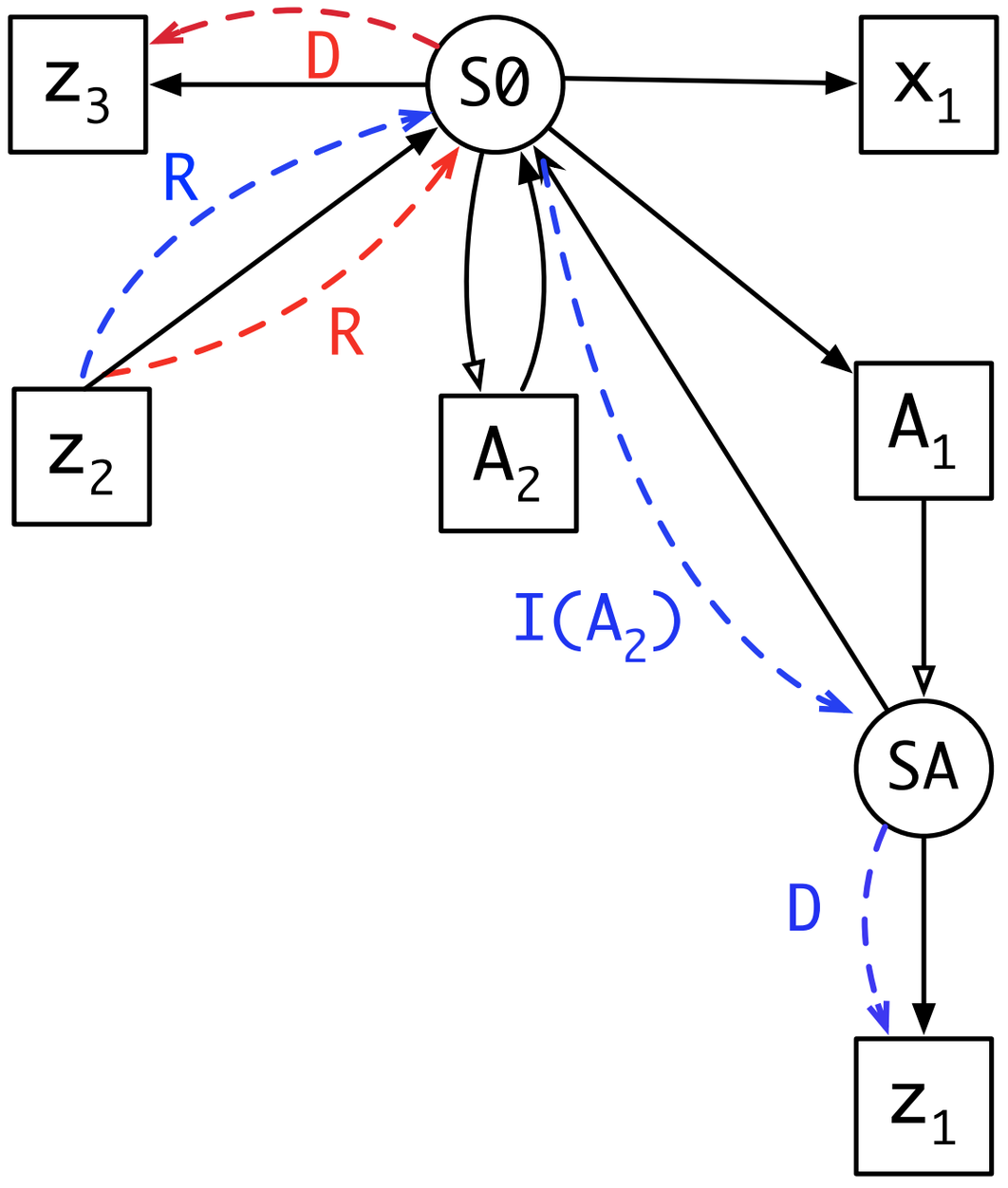
```
                                  S0
def z3 = 2
module A1 {
  def z1 = 5                      SA
}

import A2
def x1 = 1 + z2
```



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

Imports vs. Includes

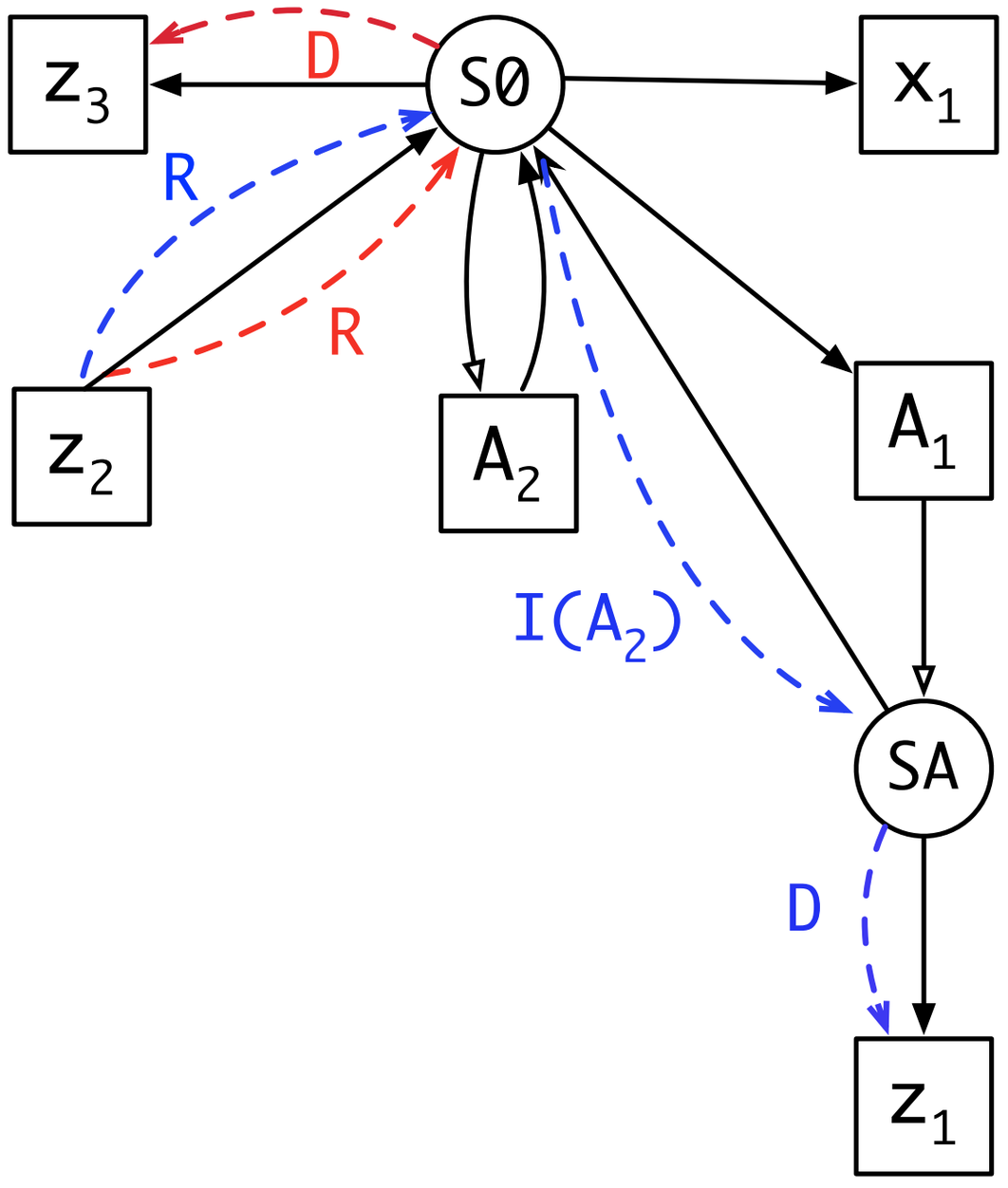
```
def z3 = 2                    S0
module A1 {
  def z1 = 5                  SA
}
include A2
def x1 = 1 + z2
```

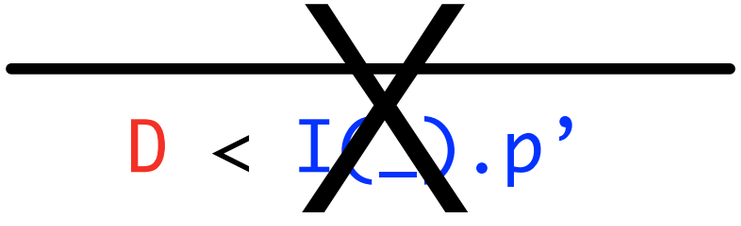


$$\frac{}{D < I(\_).p'} \quad \Rightarrow \quad R.D < R.I(A_2).D$$

Imports vs. Includes

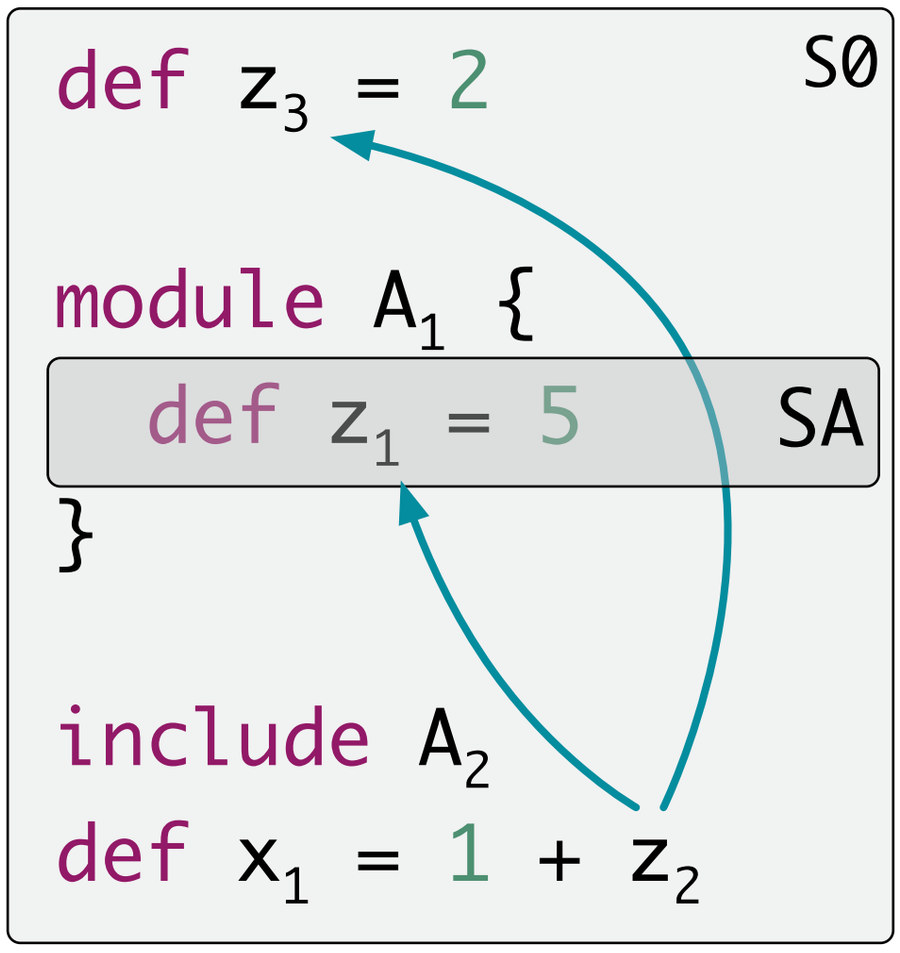
```
def z3 = 2                    S0
module A1 {
  def z1 = 5                  SA
}
include A2
def x1 = 1 + z2
```

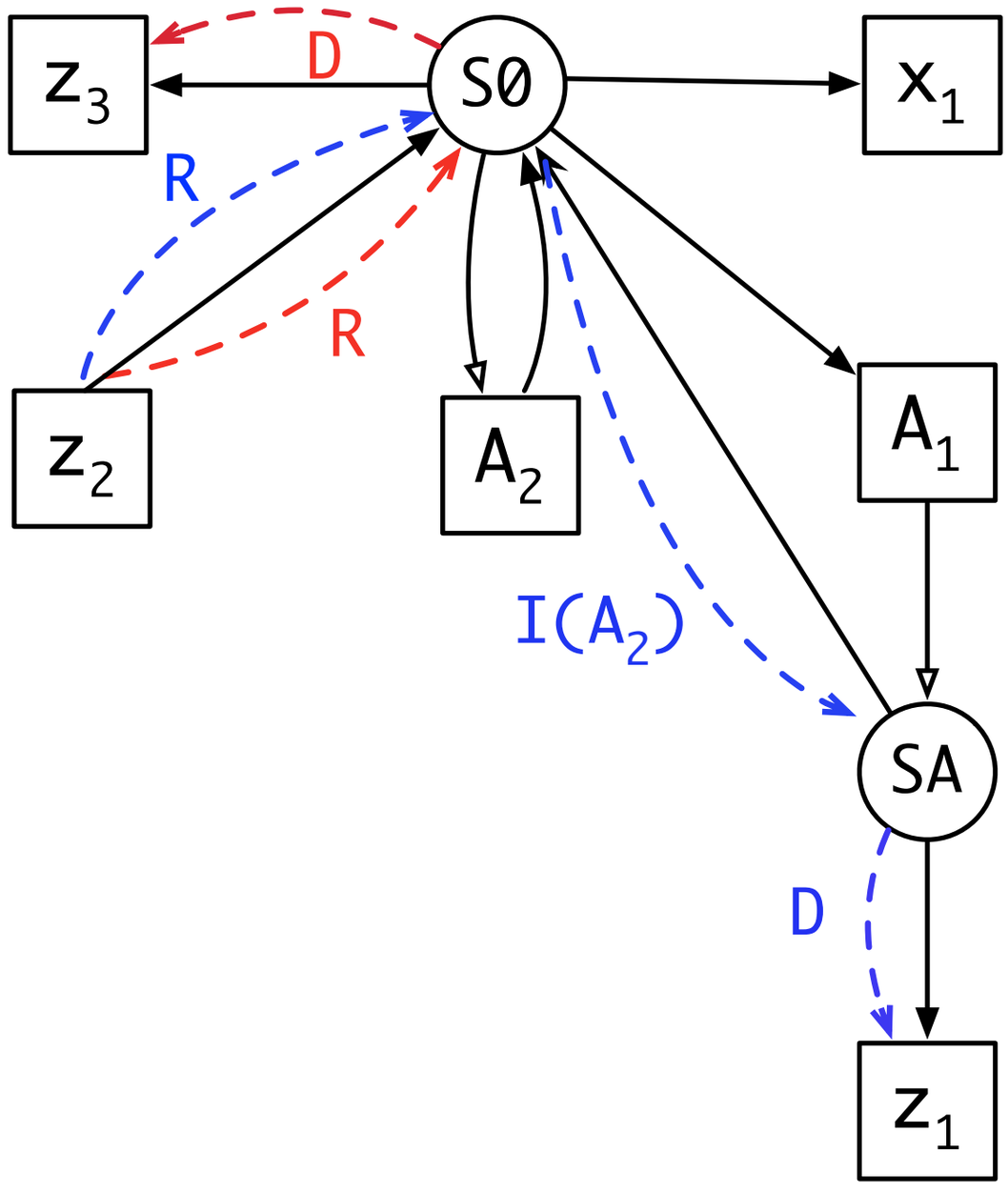


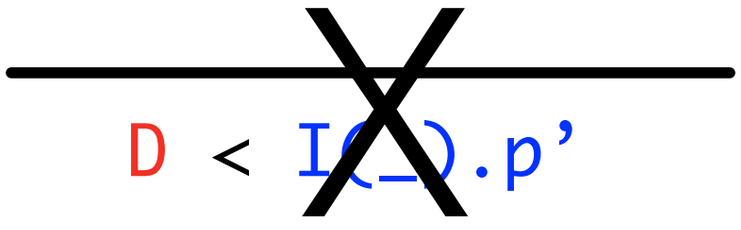


$\Rightarrow$ R.D < R.I(A_2).D

Imports vs. Includes







Import Parents

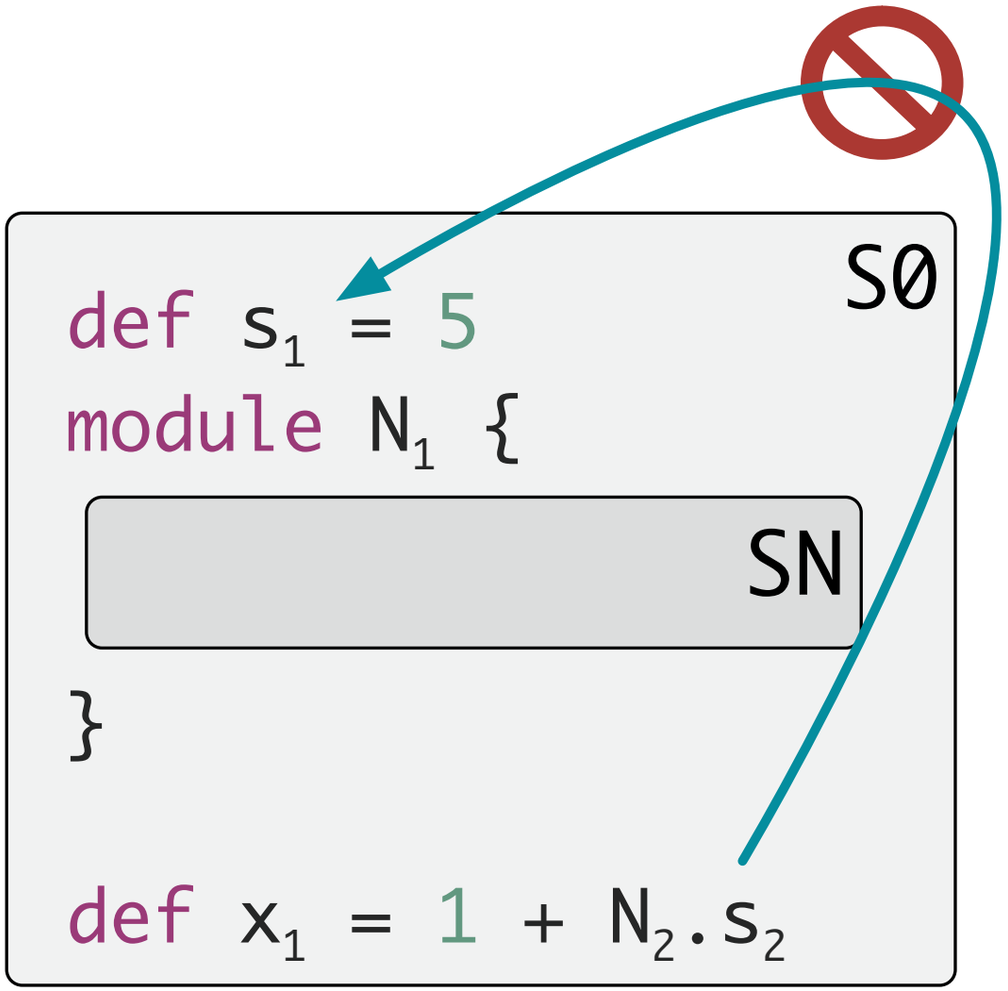
S0

```
def s₁ = 5
module N₁ {
```

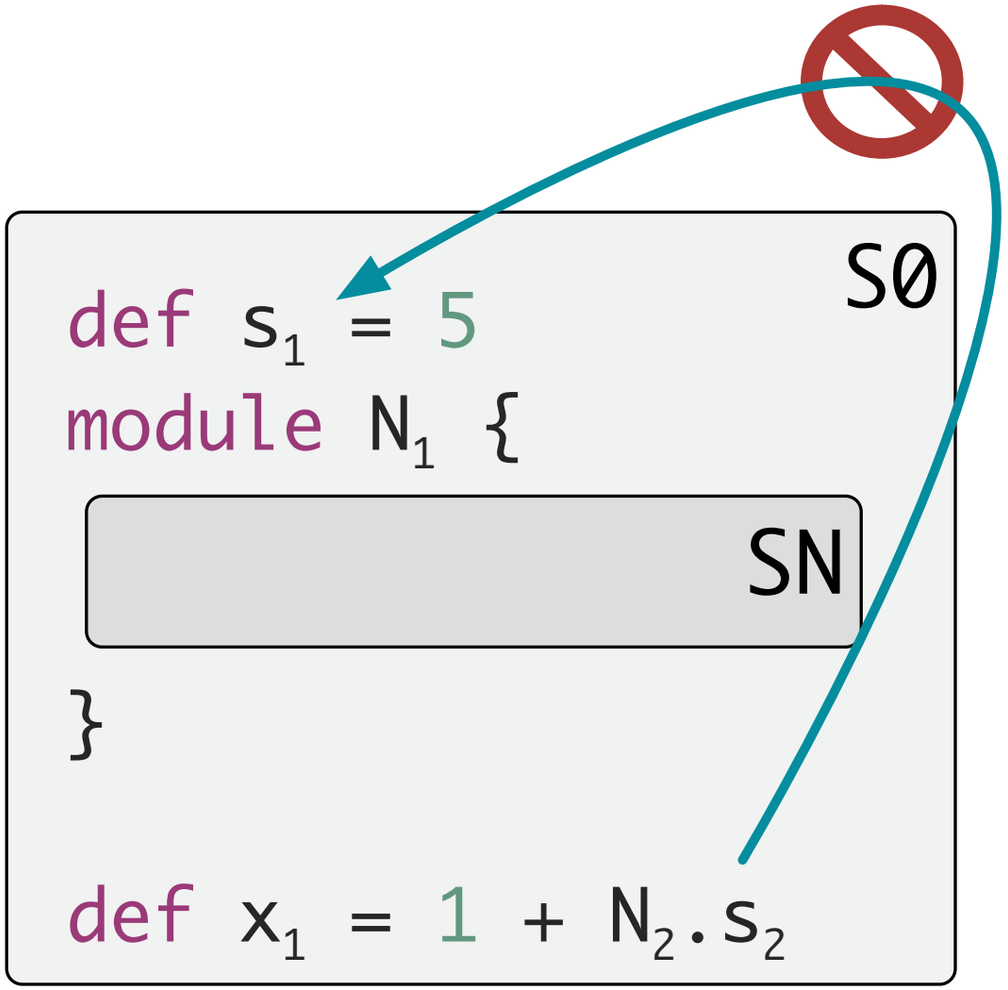
SN

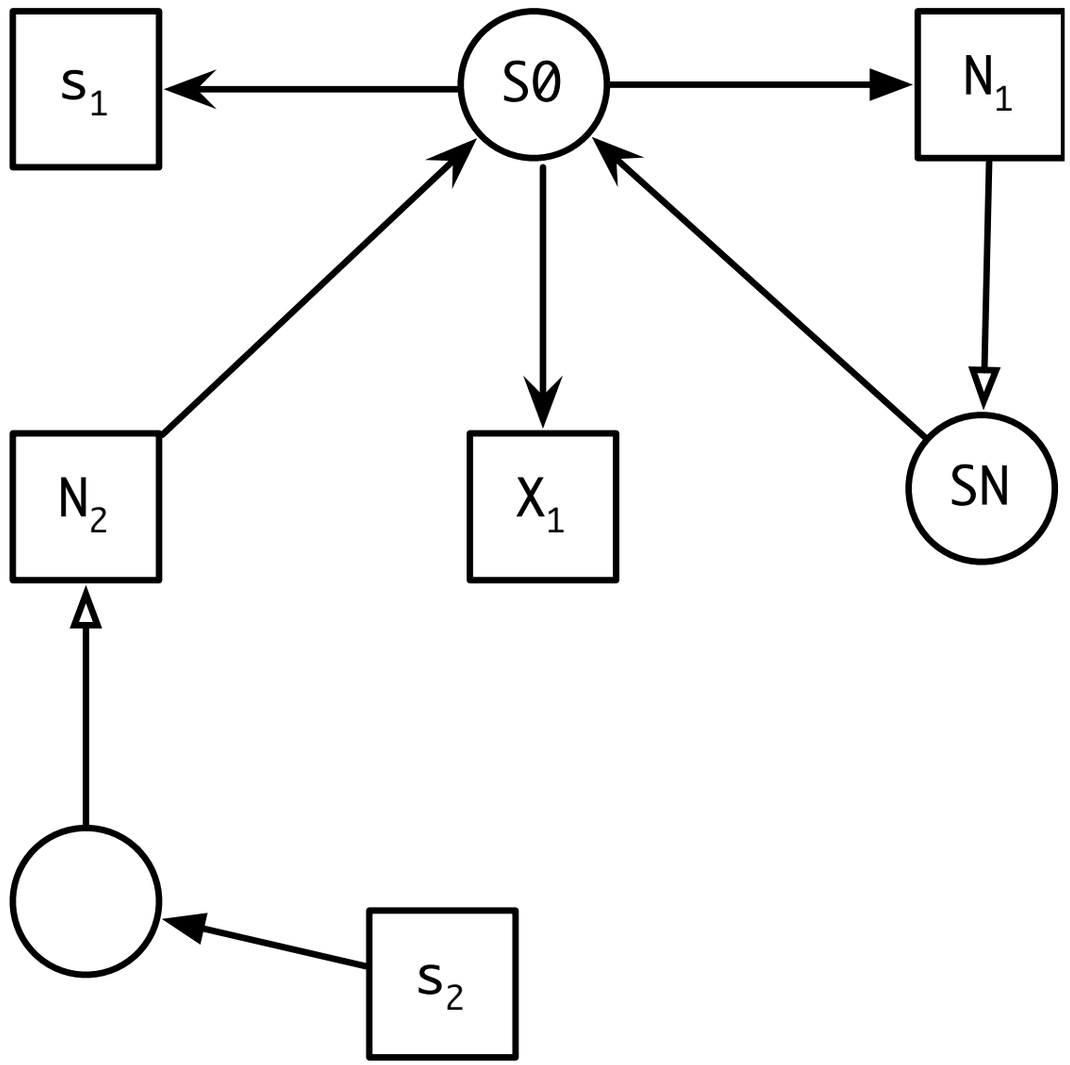
```
}

def x₁ = 1 + N₂.s₂
```

Import Parents

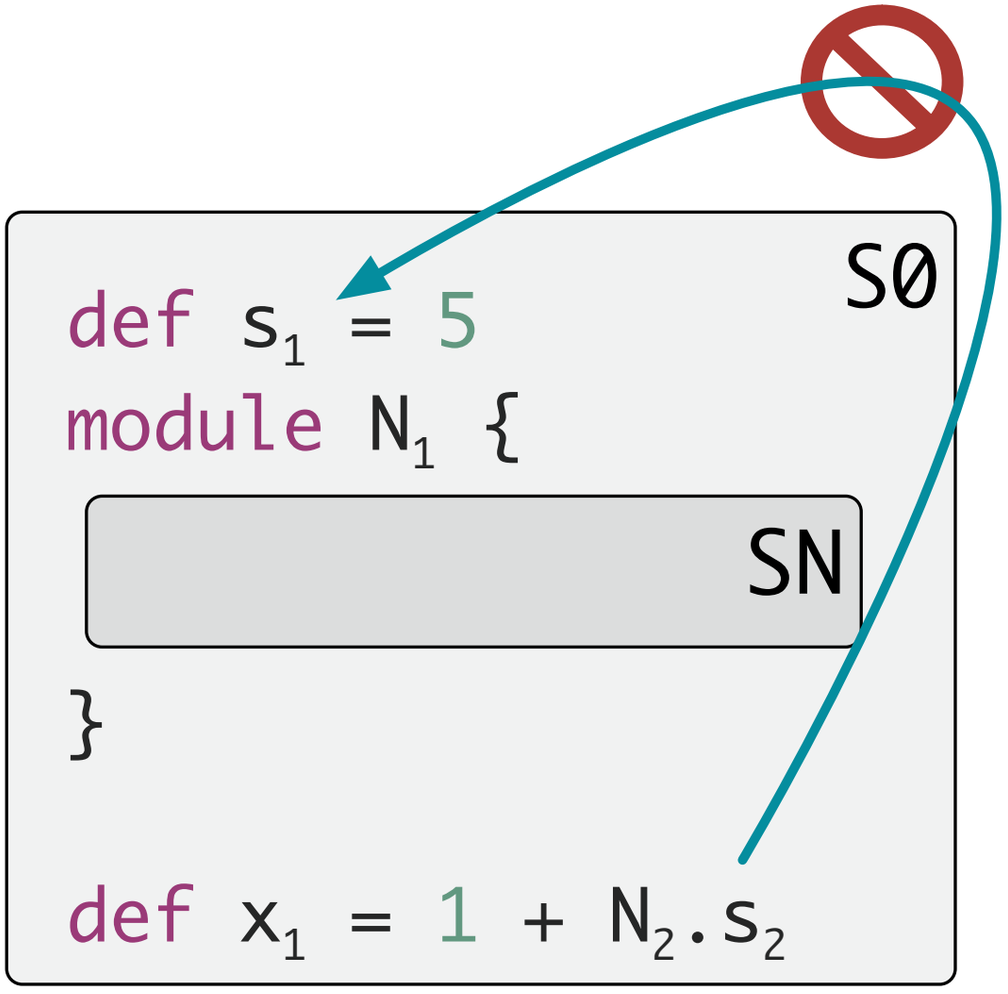


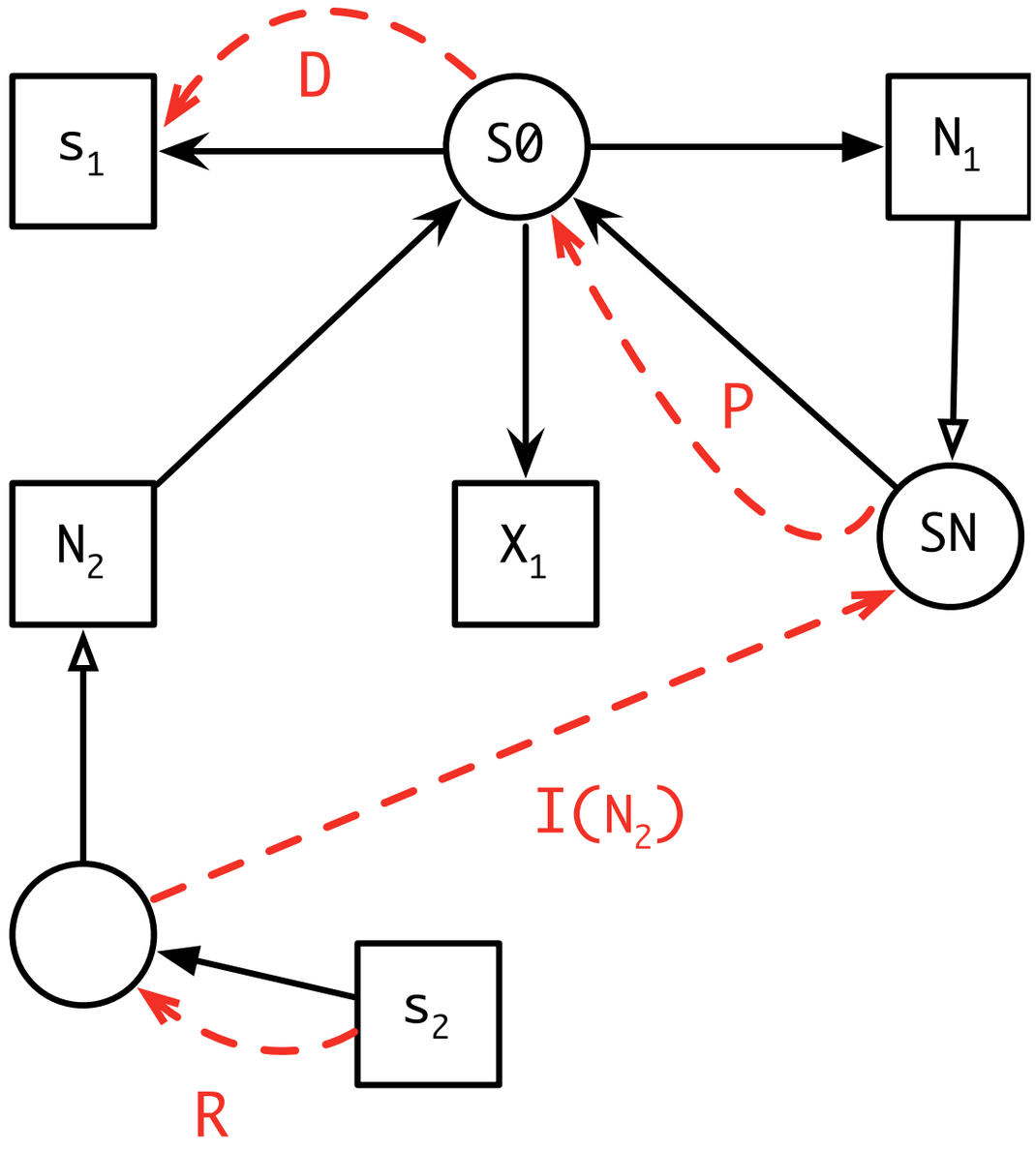
Import Parents



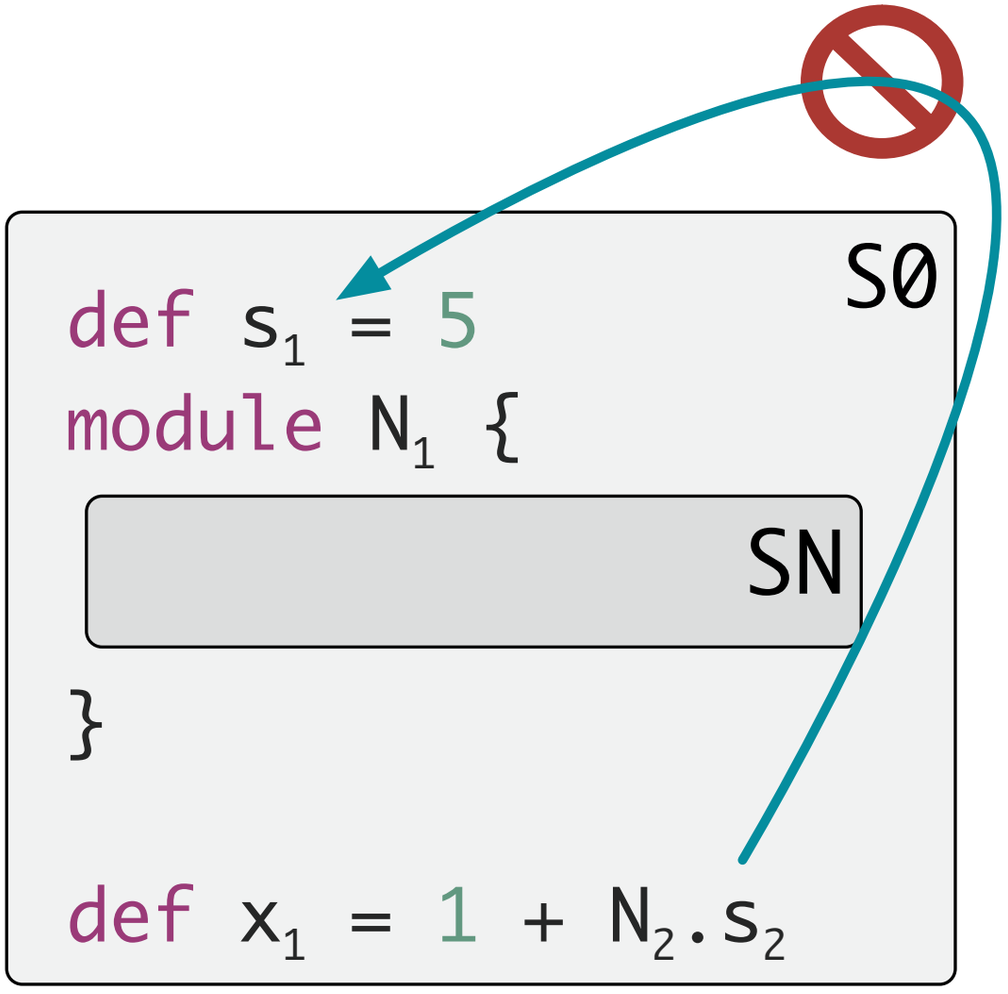


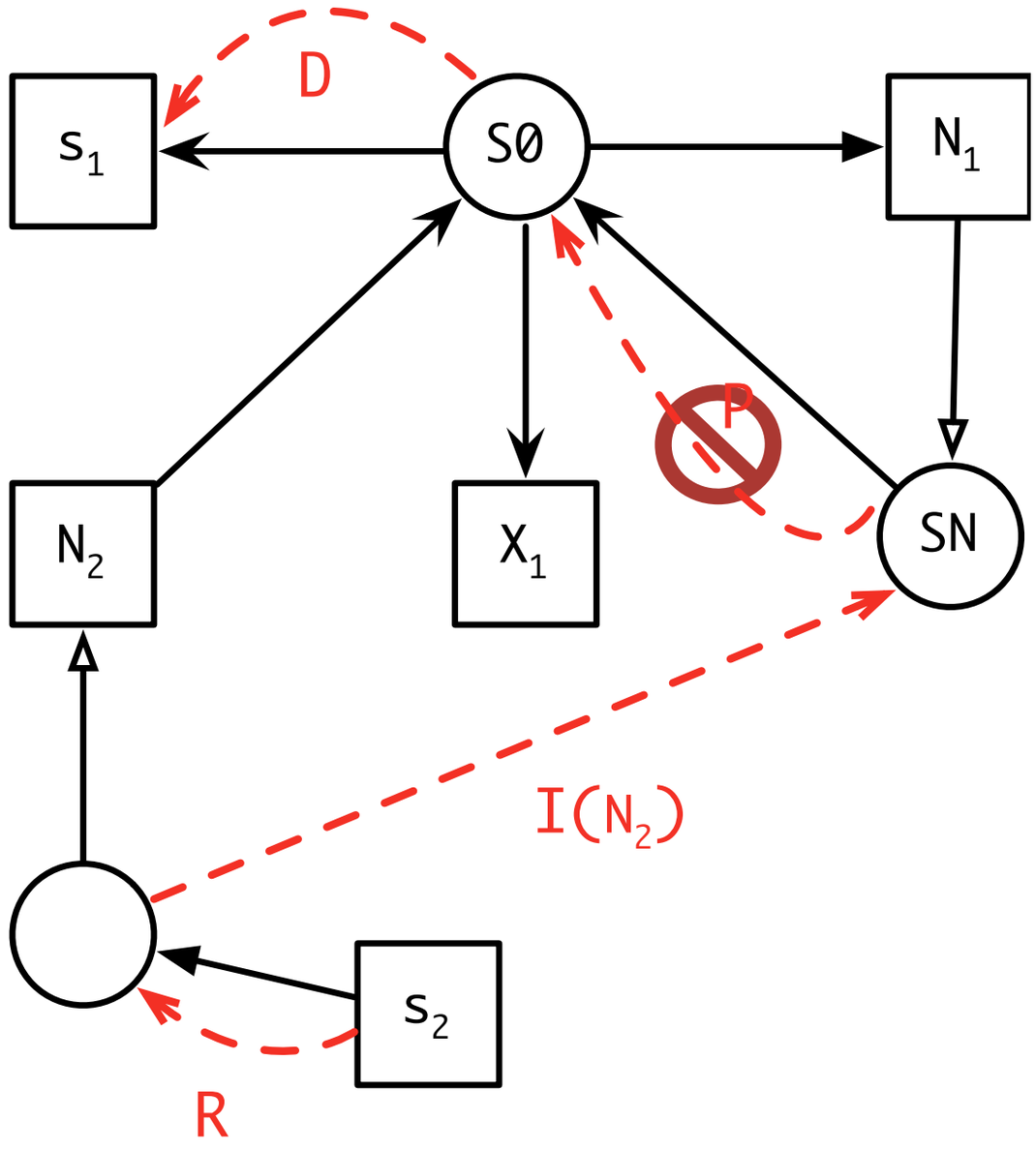
Import Parents



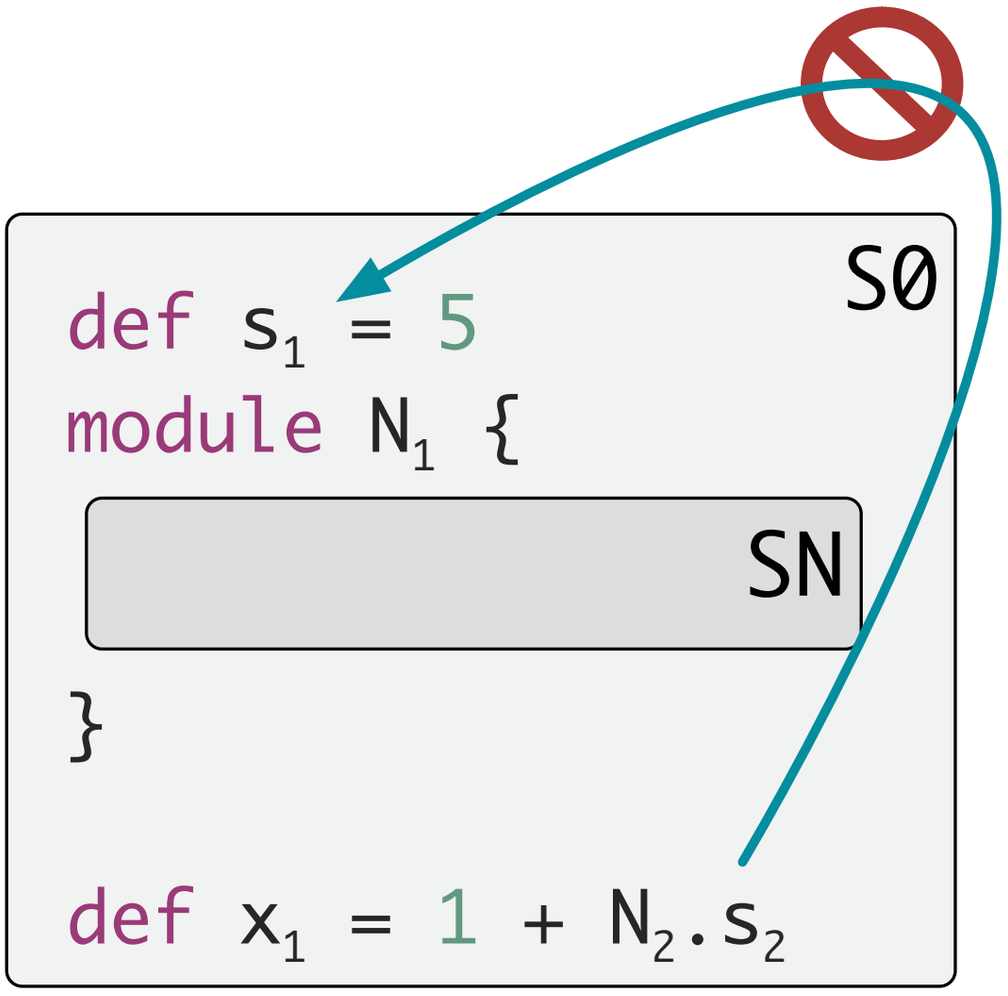


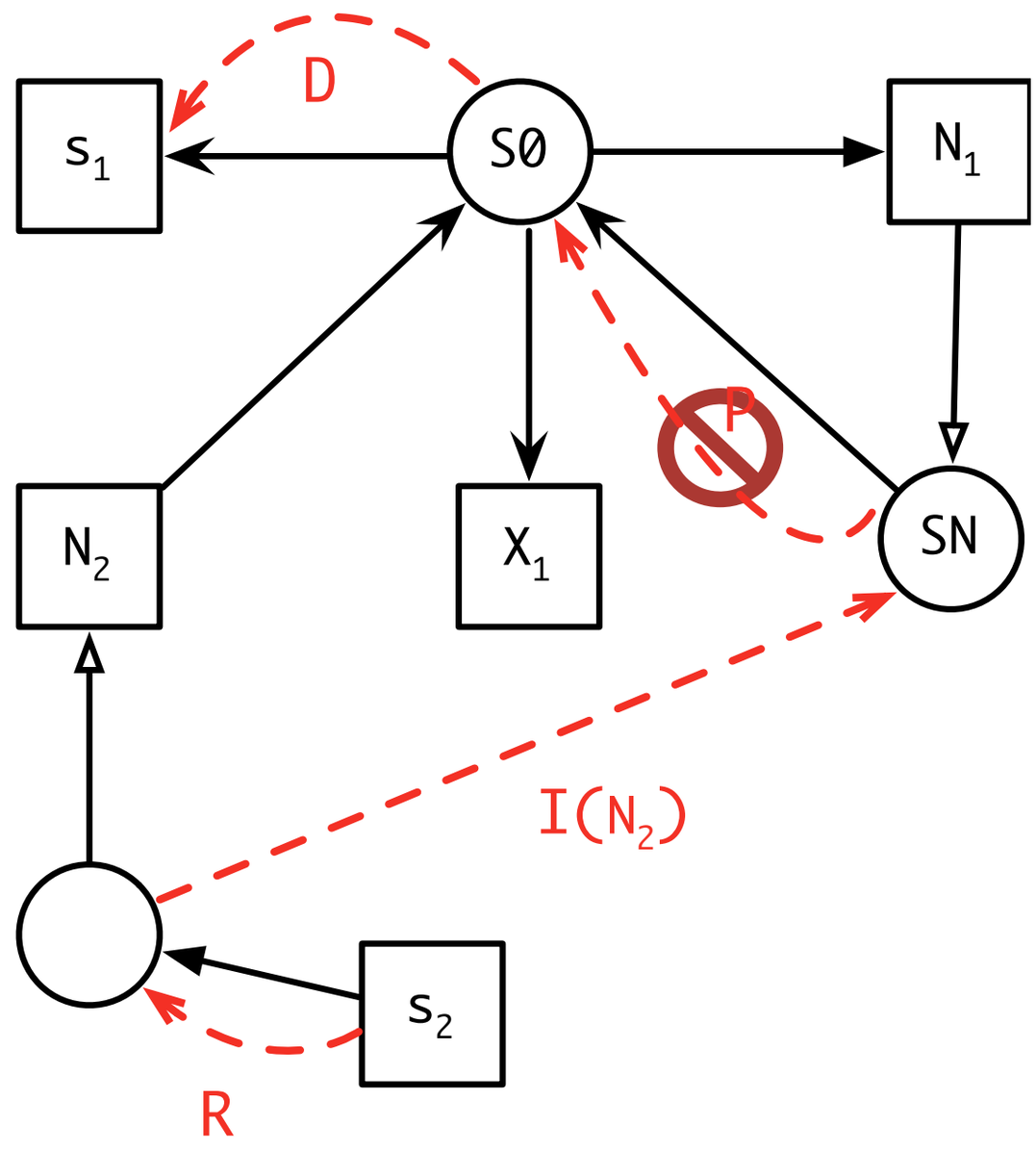
Import Parents





Import Parents



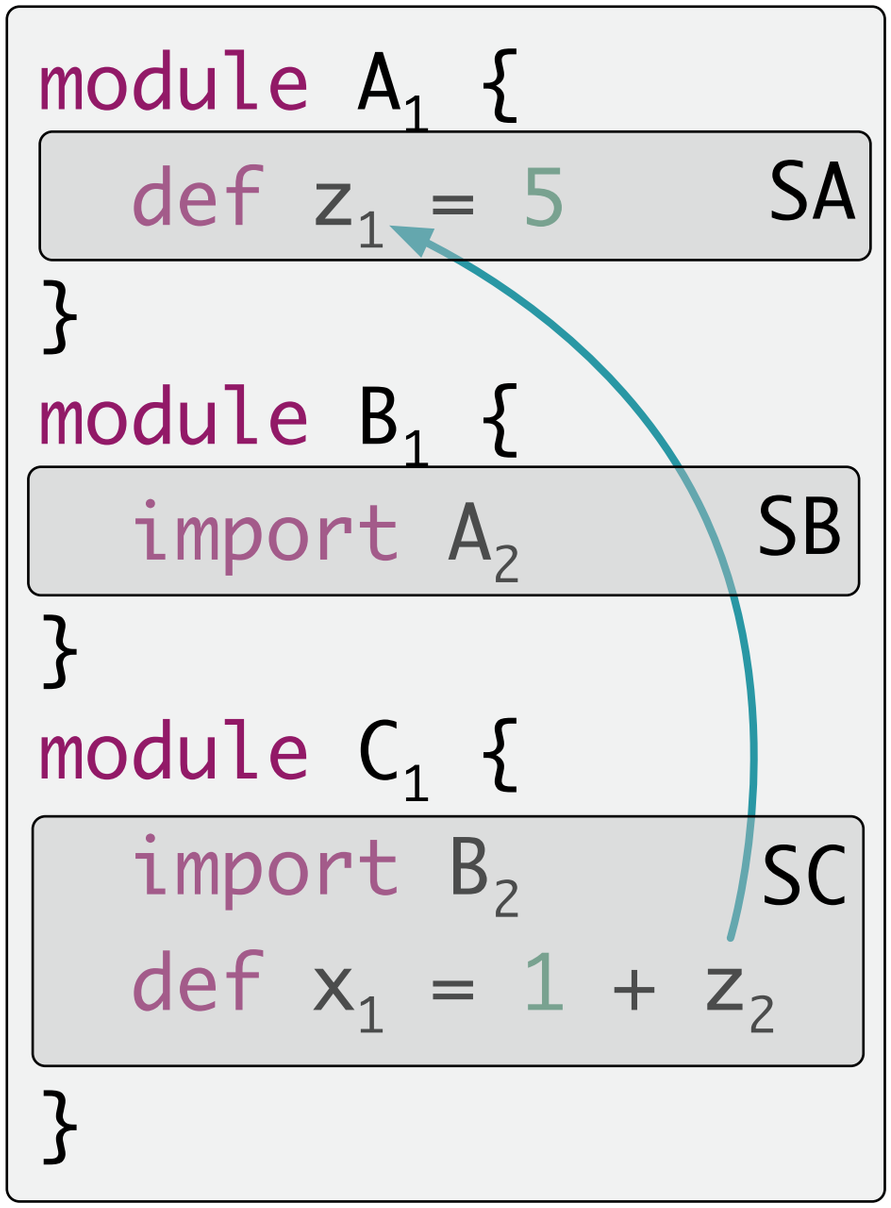


Well formed path: R.P*.I(_)*.D

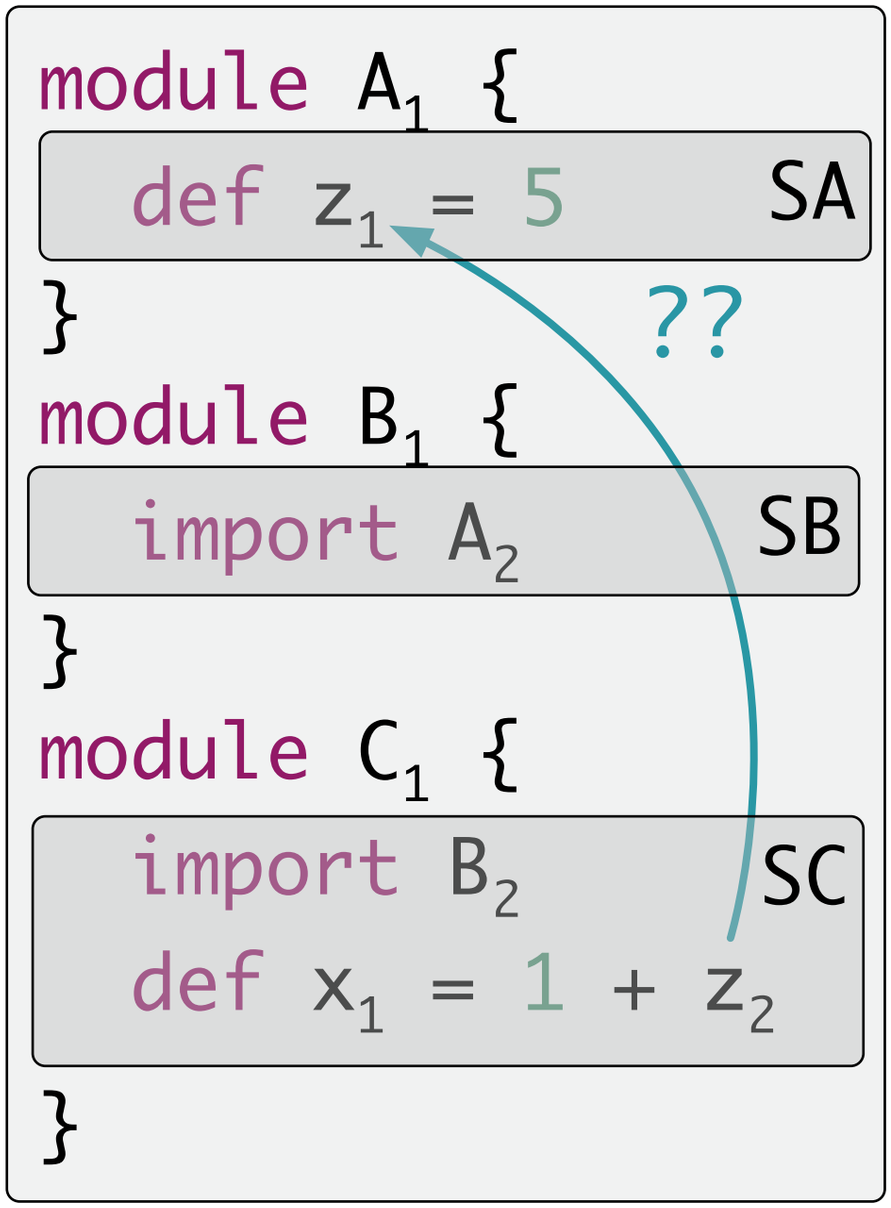
Transitive vs. Non-Transitive

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

Transitive vs. Non-Transitive

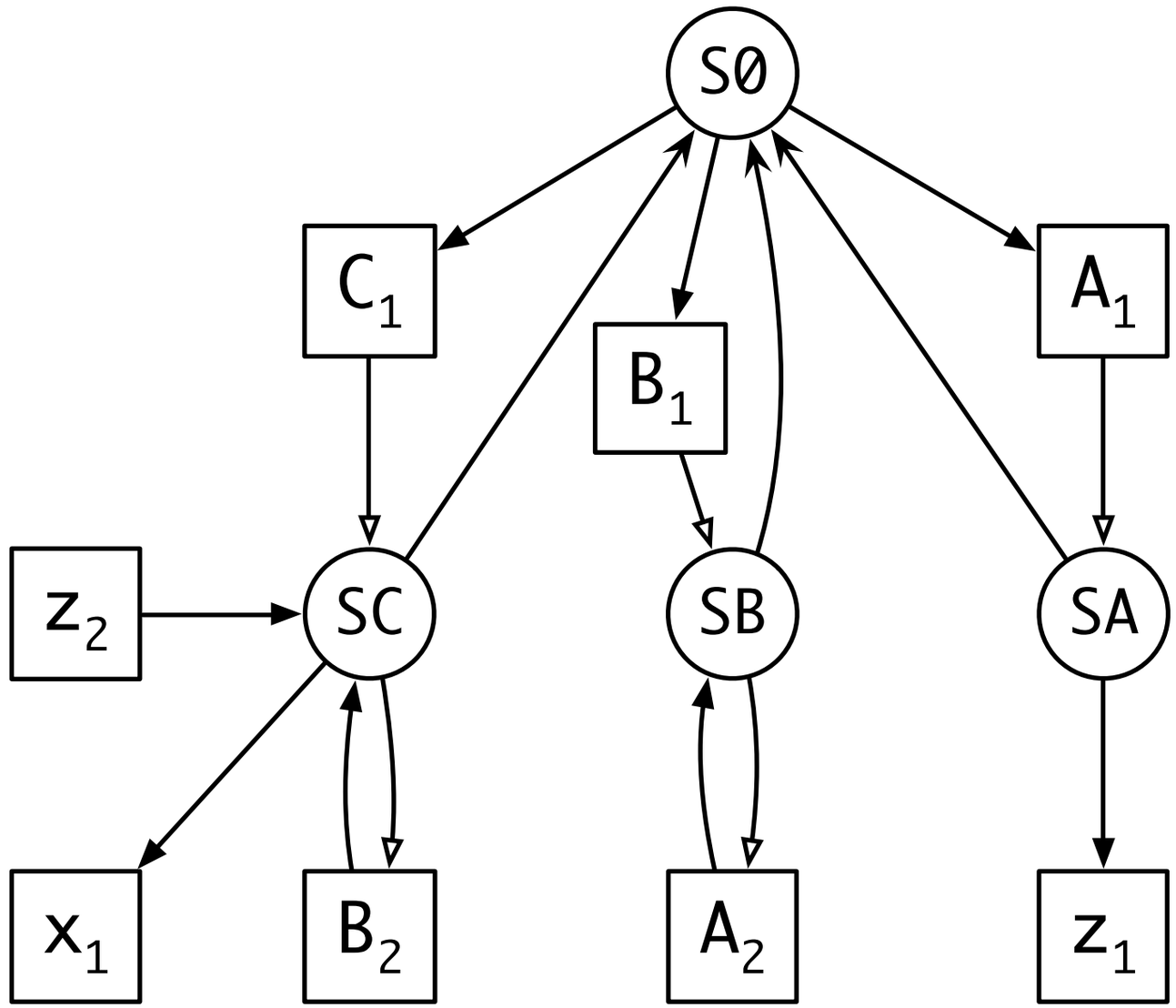


Transitive vs. Non-Transitive



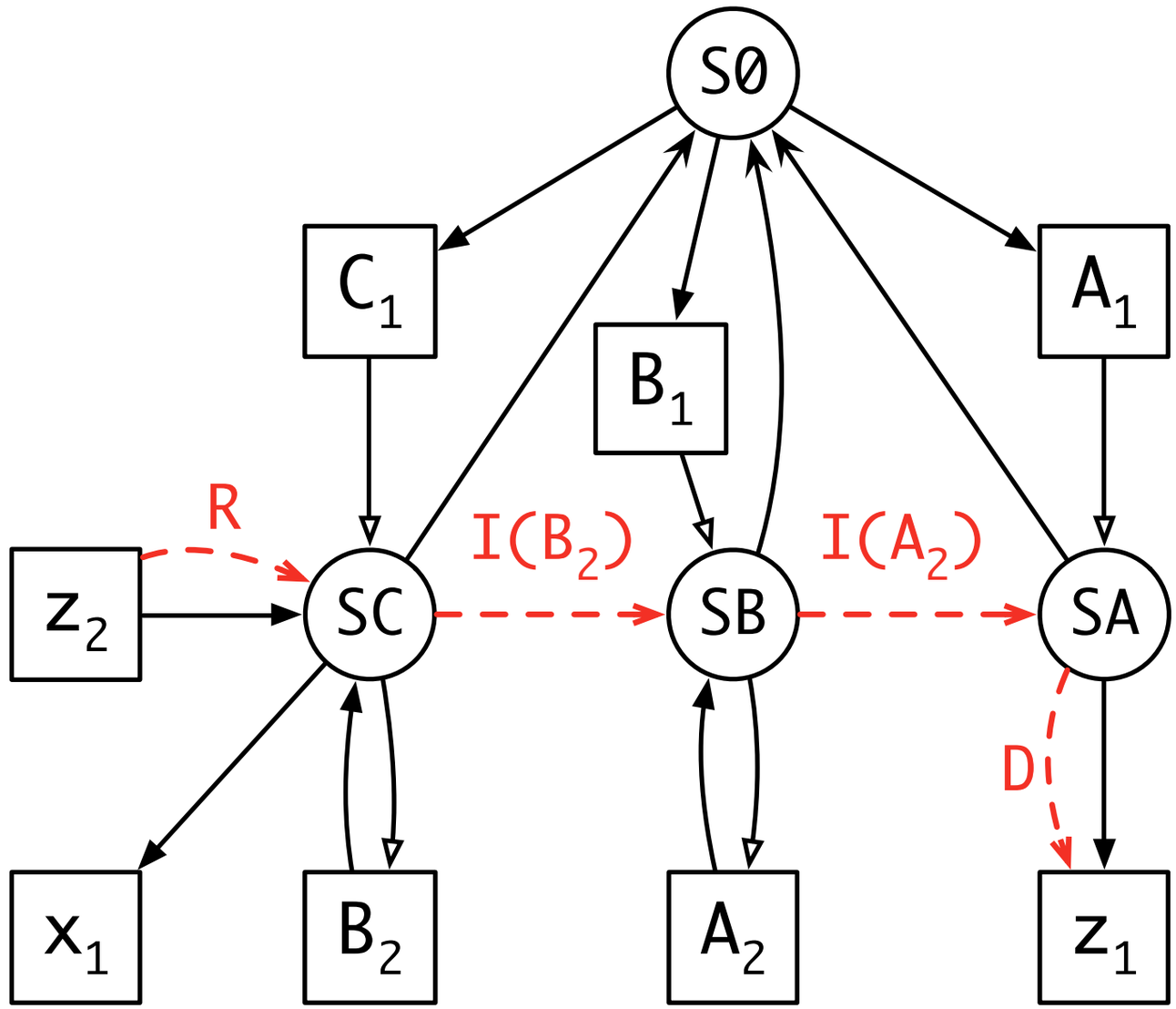
Transitive vs. Non-Transitive

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



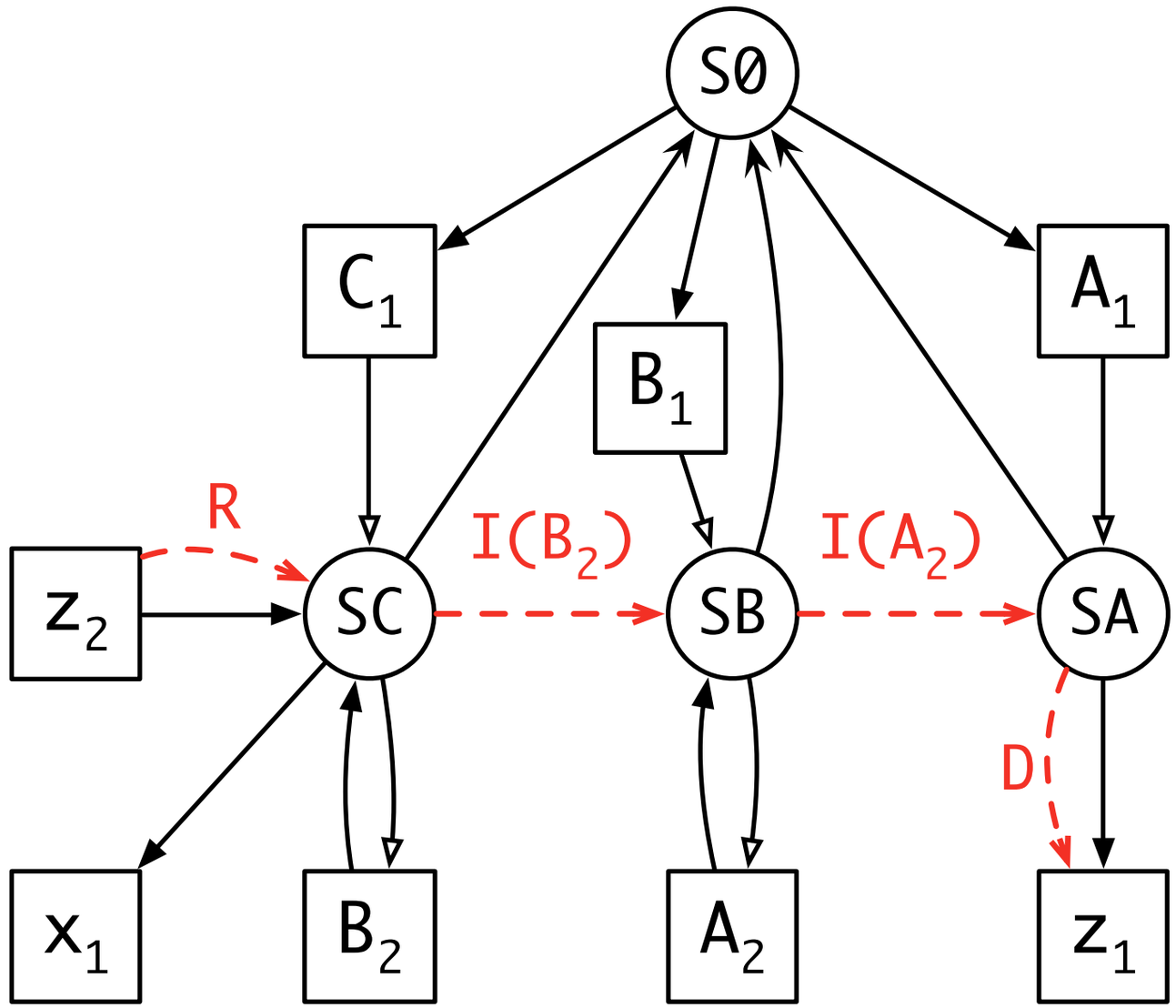
Transitive vs. Non-Transitive

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



Transitive vs. Non-Transitive

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

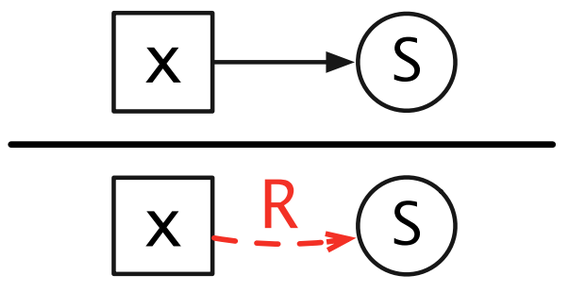


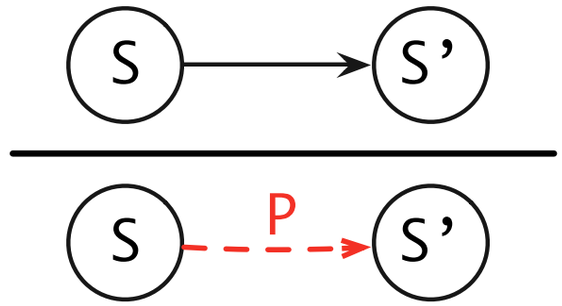
With transitive imports, a well formed path is `R.P*.I(_)*.D`

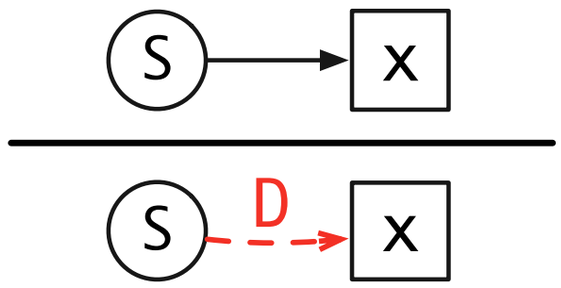
With non-transitive imports, a well formed path is `R.P*.I(_)?.D`

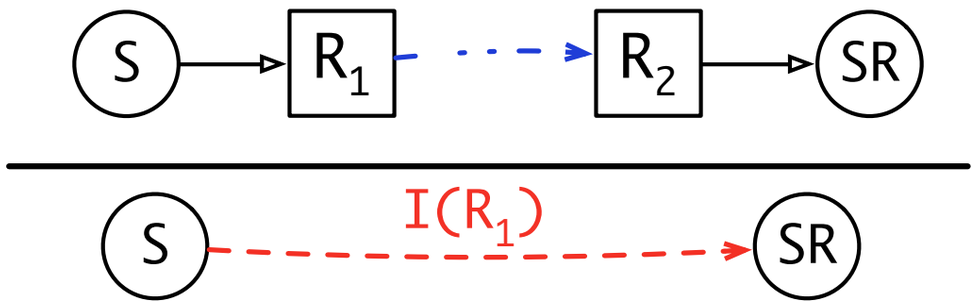
A Calculus for Name Resolution

Reachability









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

Visibility

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

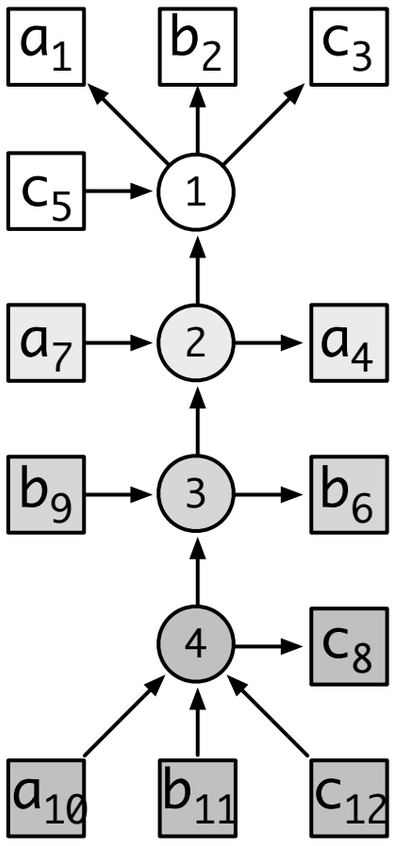
$$\frac{p < p'}{s.p < s.p'} \qquad \frac{}{D < I(\_).p'}$$

Examples

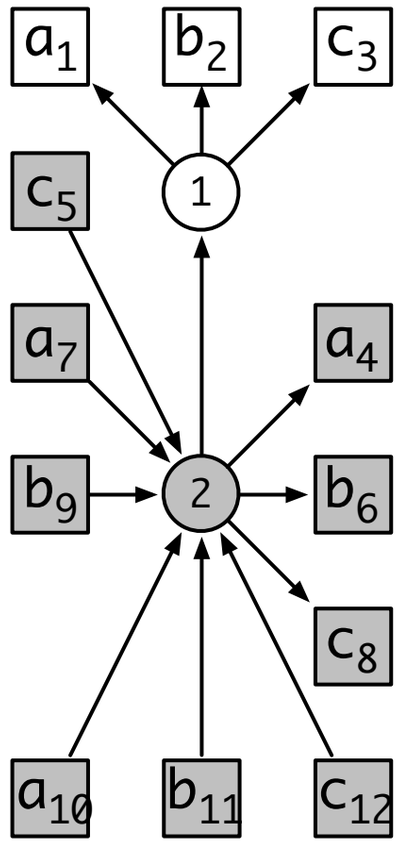
(From TUD-SERG-2015-001. Adapted for this screen)

Let Bindings

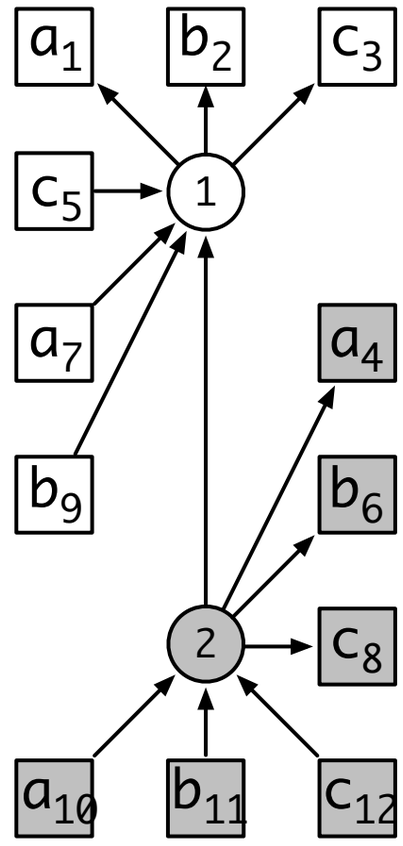
```
def a1 = 0
def b2 = 1
def c3 = 2

let
  a4 = c5
  b6 = a7
  c8 = b9
in
  a10+b11+c12
```




```
def a1 = 0
def b2 = 1
def c3 = 2

letrec
  a4 = c5
  b6 = a7
  c8 = b9
in
  a10+b11+c12
```

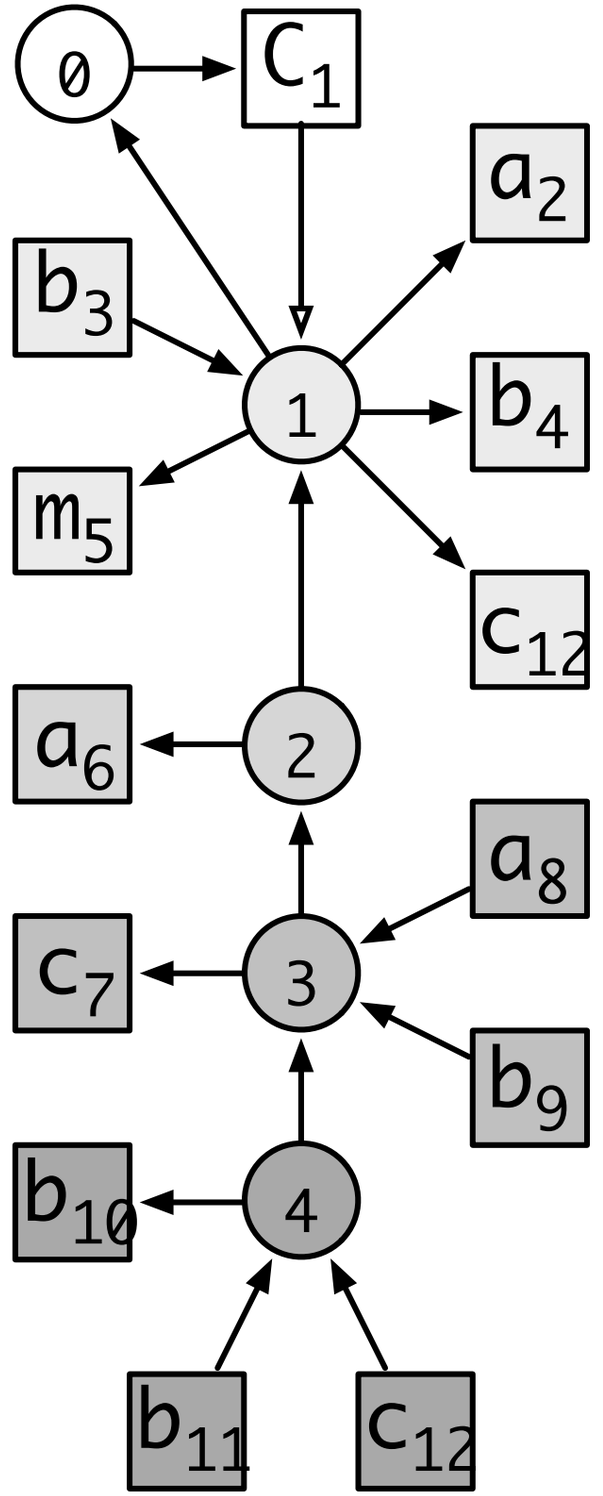



```
def a1 = 0
def b2 = 1
def c3 = 2

letpar
  a4 = c5
  b6 = a7
  c8 = b9
in
  a10+b11+c12
```



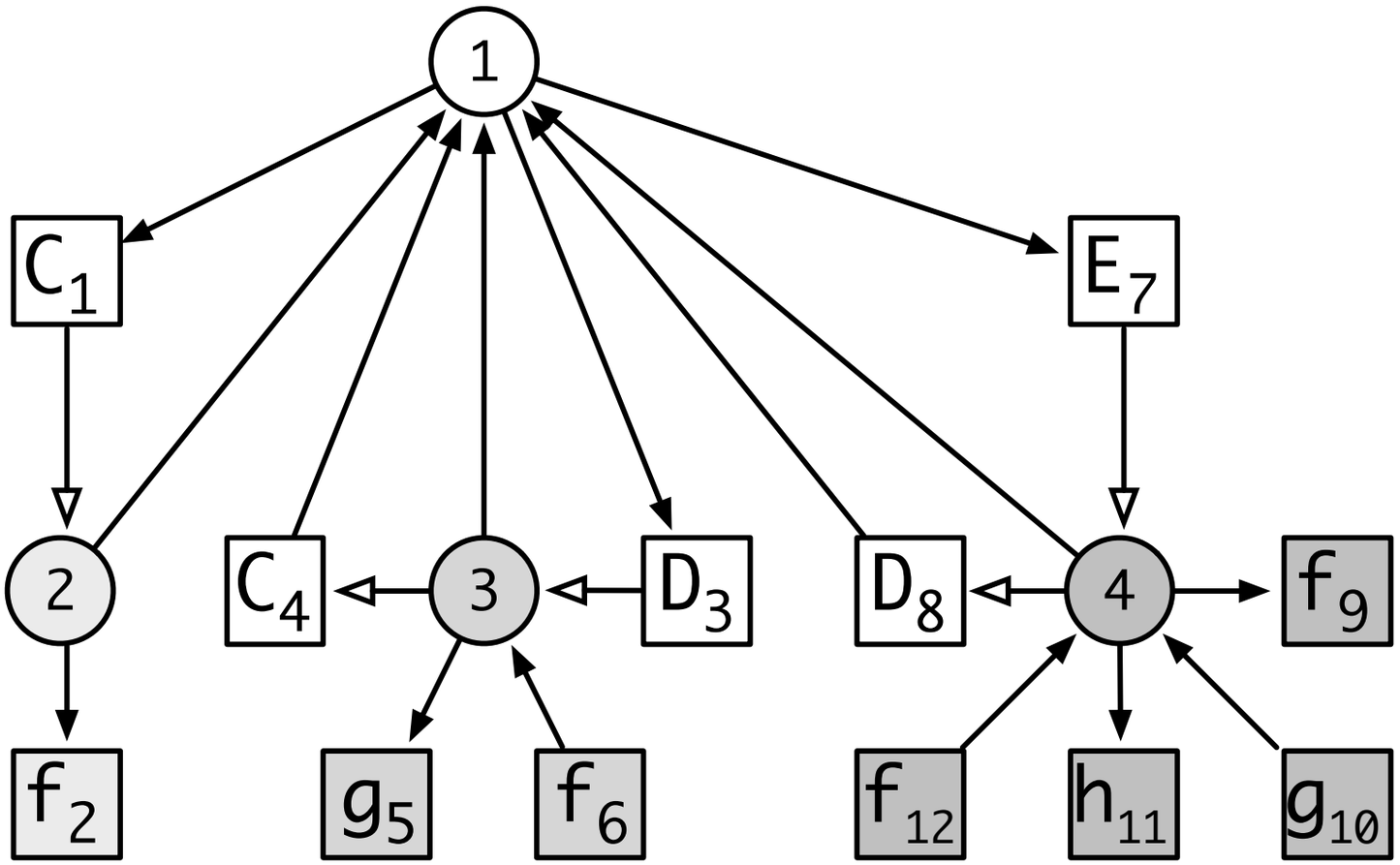
Definition before Use / Use before Definition

```
class C1 {
  int a2 = b3;
  int b4;
  void m5 (int a6) {

    int c7  = a8 + b9;
    int b10 = b11 + c12;

  }
  int c12;
}
```



Inheritance

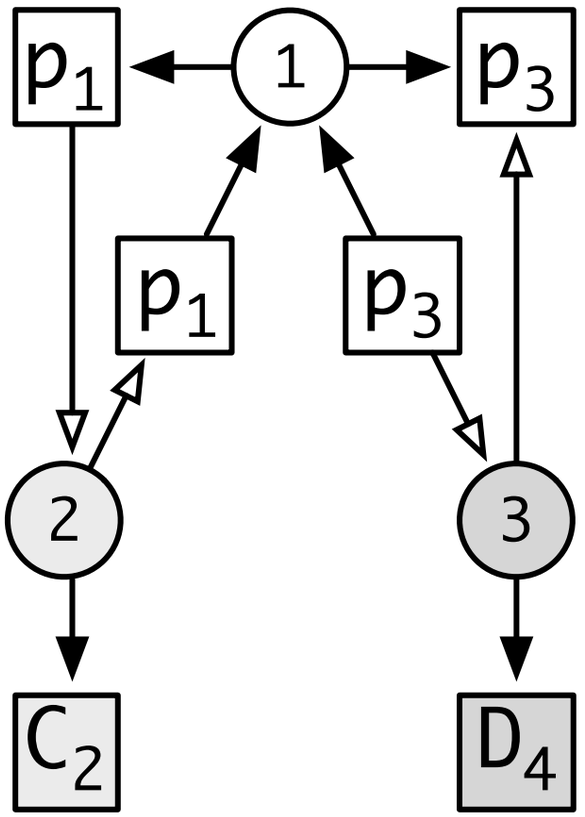
```
class C1 {
  int f2 = 42;
}
class D3 extends C4 {
  int g5 = f6;
}
class E7 extends D8 {
  int f9  = g10;
  int h11 = f12;
}
```



Java Packages

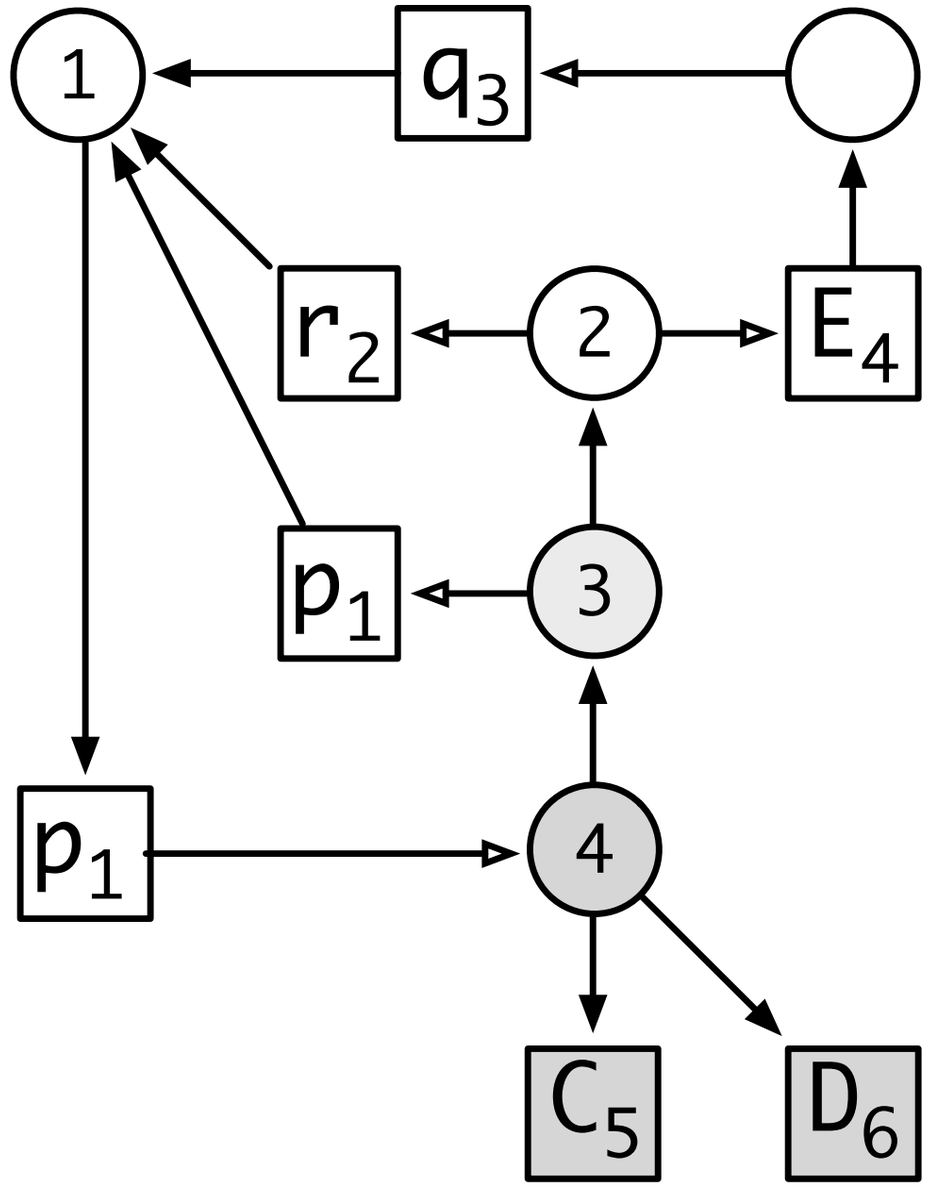
```
package p1;

class C2 {}
```

```
package p3;

class D4 {}
```



Java Import

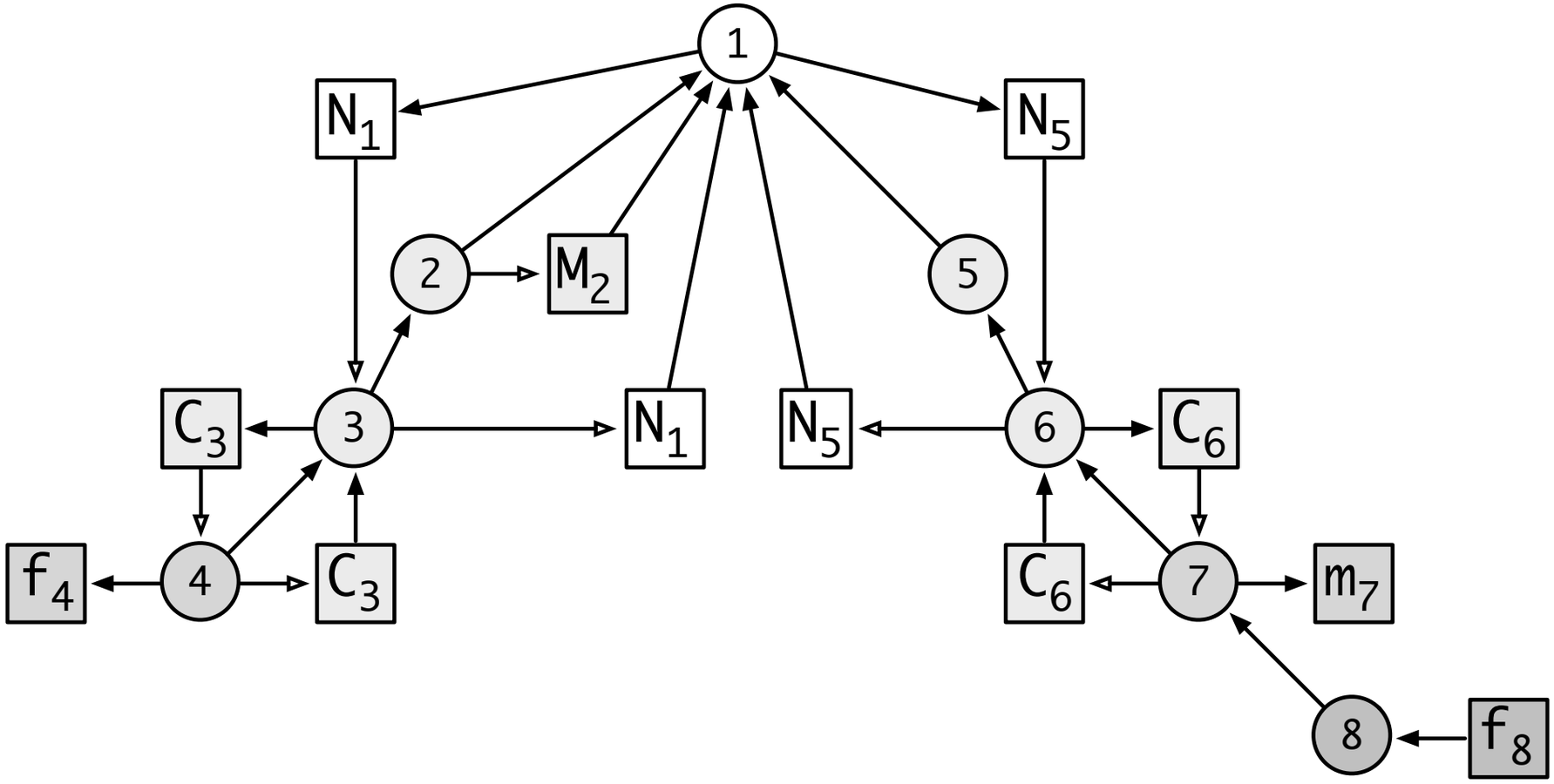
```
package p1;

imports r2.*;
imports q3.E4;

public class C5 {}

class D6 {}
```



C# Namespaces and Partial Classes

```
namespace N1 {
  using M2;

  partial class C3 {
    int f4;
  }
}
```

```
namespace N5 {
  partial class C6 {
    int m7() {
      return f8;
    }
  }
}
```



More Examples?

Here the audience proposes binding patterns that they think are not covered by the scope graph framework (and they may be right?)

Blocks in Java

```
class Foo {
  void foo() {
   int x = 1;
   {
     int y = 2;
   }
   x = y;
  }
}
```

What is the scope graph for this program?

Is the y declaration visible to the y reference?

Constraint Language

Constraint Generation

```
module constraint-gen

imports signatures/-

rules
```

```
[[ Constr(t1, ..., tn) ^ (s1, …, sn) ]] :=
   C1,
   ...,
   Cn.
```

one rule per abstract syntax constructor

Name Binding Constraints

```
false              // always fails
true               // always succeeds

C1, C2             // conjunction of constraints

[[ e ^ (s) ]]      // generate constraints for sub-term

new s              // generate a new scope

NS{x} <- s         // declaration of name x in namespace NS in scope s
NS{x} -> s         // reference of name x in namespace NS in scope s

s -> s'            // unlabeled scope edge from s to s'
s -L-> s'          // scope edge from s to s' labeled L

NS{x} |-> d        // resolve reference x in namespace NS to declaration d


C | error $[something is wrong]     // custom error message
C | warning $[does not look right] // custom warning
```

Tiger Name Binding

Let Bindings

```
let
  var x : int := 5
  var f : int := 1
in
  for y := 1 to x do (
    f := f * y
  )
end
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in fact(10)
end
```

Let Bindings are Sequential

```
let
  var x : int := 0 + z  // z not in scope
  var y : int := x + 1
  var z : int := x + y + 1
 in
   x + y + z
end
```

Adjacent Function Declarations are Mutually Recursive

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

Namespaces

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
 in foo(56) + foo // both refer to the variable foo
end
```

Tiger in NaBL2

Set Up

```
module static-semantics

imports signatures/-
imports nabl-lib

signature
  name resolution
    labels
      D, P, I
    well-formedness
      P* . I*
    order
      D < P,
      D < I,
      I < P
rules

  init ^ (s) :=
    new s,                  // generate the root scope
    Type{"int"} <- s. // declare primitive type int
```

Library

```
module nabl-lib

rules // auxiliary

  [[ None() ^ (s) ]] := true.
  [[ Some(e) ^ (s) ]] := [[ e ^ (s) ]].

  Map[[ [] ^ (s) ]] := true.
  Map[[ [ x | xs ] ^ (s) ]] :=
    [[ x ^ (s) ]], Map[[ xs ^ (s) ]].

  Map2[[ [] ^ (s, s') ]] := true.
  Map2[[ [ x | xs ] ^ (s, s') ]] :=
    [[ x ^ (s, s') ]], Map2[[ xs ^ (s, s') ]].
```

Variable Declarations and References

```
rules // variable declarations

  [[ VarDec(x, t, e) ^ (s, s_outer) ]] :=
     Var{x} <- s,
     [[ t ^ (s_outer) ]],
     [[ e ^ (s_outer) ]].

  [[ VarDecNoInit(x, t) ^ (s, s_outer) ]] :=
     Var{x} <- s,
     [[ t ^ (s_outer) ]].

rules // variable references

  [[ Var(x) ^ (s) ]] :=
     Var{x} -> s,  // declare x as variable reference
     Var{x} |-> d. // check that x resolves to a declaration
```

```
let
  var x : int := 5
  var f : int := 1
in
  for y := 1 to x do (
    f := f * y
  )
end
```

For Binds Loop Index Variable

```
rules // binding statements

  [[ For(Var(x), e1, e2, e3) ^ (s) ]] :=
     new s',
     s' -P-> s,
     Var{x} <- s',
     [[ e1 ^ (s) ]], // x not bound in loop bounds
     [[ e2 ^ (s) ]],
     [[ e3 ^ (s') ]]. // x bound in body
```

```
let
  var x : int := 5
  var f : int := 1
in
  for y := 1 to x do (
    f := f * y
  )
end
```

Function Declarations and References

```
rules // function declarations

  [[ FunDecs(fdecs) ^ (s, s_outer) ]] :=
     Map2[[ fdecs ^ (s, s_outer) ]].

  [[ FunDec(f, args, t, e) ^ (s, s_outer) ]] :=
     Var{f} <- s, // declare f
     new s',      // declare a new scope for body of function
     s' -P-> s,   // make lexical environment accessible in body
     Map2[[ args ^ (s', s_outer) ]],
     [[ t ^ (s_outer) ]],
     [[ e ^ (s') ]].

  [[ FArg(x, t) ^ (s, s_outer) ]] :=
     Var{x} <- s, // declare argument x as a variable declaration
     [[ t ^ (s_outer) ]].

rules // function calls

  [[ Call(Var(f), exps) ^ (s) ]] :=
     Var{f} -> s,  // declare f as a function reference
     Var{f} |-> d | error $[Function [f] not declared],
     // check that f resolves to a declaration
     Map[[ exps ^ (s) ]].
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in fact(10)
end
```

Let Bindings are Sequential

```
rules // declarations

  [[ Let(blocks, exps) ^ (s) ]] :=
     new s',      // declare a new scope for the names introduced by the let
     s' -P-> s,   // declare s as its parent scope
     new s_body, // scope for body of the let
     Decs[[ blocks ^ (s', s_body) ]],
     Map[[ exps ^ (s_body) ]].

  Decs[[ [] ^ (s, s_body) ]] :=
    s_body -P-> s.

  Decs[[ [block | blocks] ^ (s_outer, s_body) ]] :=
    new s', s' -P-> s_outer,
    [[ block ^ (s', s_outer) ]],
    Decs[[ blocks ^ (s', s_body) ]].
```

```
let
  var x : int := 0 + z  // z not in scope
  var y : int := x + 1
  var z : int := x + y + 1
 in
   x + y + z
end
```

Adjacent Function Declarations are Mutually Recursive

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

Namespaces

```
rules // type declarations

  [[ TypeDecs(tdecs) ^ (s, s_outer) ]] :=
     Map2[[ tdecs ^ (s, s_outer) ]].

  [[ TypeDec(x, t) ^ (s, s_outer) ]] :=
     Type{x} <- s,
     [[ t ^ (s_outer) ]].

rules // types

  [[ Tid(x) ^ (s) ]] :=
     Type{x} -> s,
     Type{x} |-> d | error $[Type [x] not declared].
```

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
 in foo(56) + foo // both refer to the variable foo
end
```

Full Definition Online

https://github.com/MetaBorgCube/metaborg-tiger

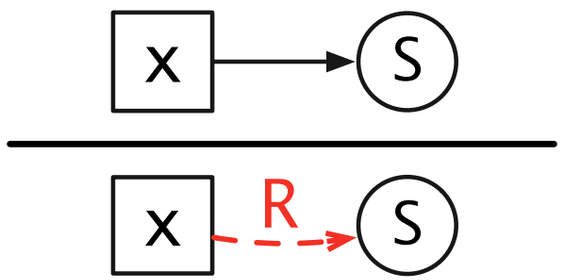
https://github.com/MetaBorgCube/metaborg-tiger/blob/master/org.metaborg.lang.tiger/trans/static-semantics/static-semantics.nabl2

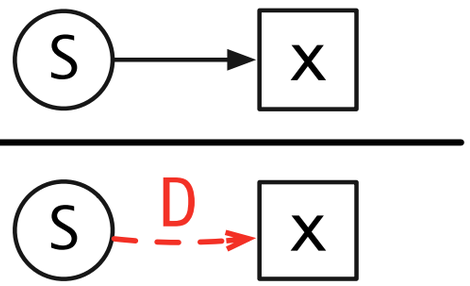
Formal Framework

(with labeled scope edges)

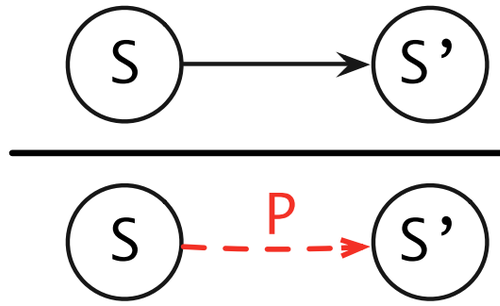
Issues with the Reachability Calculus

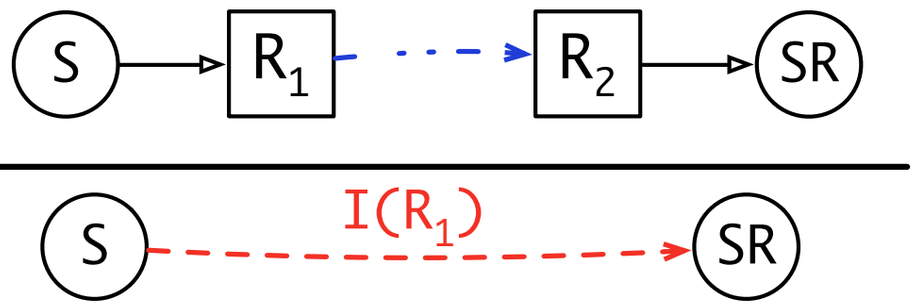
Disambiguating import paths





Cyclic Import Paths





Multi-import interpretation

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

Fixed visibility policy

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

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

Resolution Calculus with Edge Labels

$$\mathrm{R} \longrightarrow \mathrm{S} \mid \mathrm{S} \longrightarrow \mathrm{D} \mid \mathrm{S} \xrightarrow{l} \mathrm{S} \mid \mathrm{D} \longrightarrow\!\!\triangleright \mathrm{S} \mid \mathrm{S} \xrightarrow{l}\!\!\triangleright \mathrm{R}$$

Resolution paths

$$\begin{array}{rcl} s & := & \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{E}(l, S) \mid \mathbf{N}(l, x_i^{\mathsf{R}}, S) \\ p & := & [] \mid s \mid p \cdot p \qquad \text{(inductively generated)} \end{array}$$

$$[] \cdot p = p \cdot [] = p$$
$$(p_1 \cdot p_2) \cdot\ p_3 = p_1 \cdot (p_2 \cdot p_3)$$

Well-formed paths

$$WF(p) \Leftrightarrow labels(p) \in \mathcal{E}$$

Visibility ordering on paths

$$\frac{label(s_1) < label(s_2)}{s_1 \cdot p_1 < s_2 \cdot p_2} \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2}$$

Edges in scope graph

$$\frac{S_1 \xrightarrow{l} S_2}{\mathbb{I} \vdash \mathbf{E}(l, S_2) : S_1 \longrightarrow S_2} \quad (E)$$

$$\frac{S_1 \xrightarrow{l}\!\!\triangleright\ y_i^{\mathsf{R}} \quad y_i^{\mathsf{R}} \notin \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}} \quad y_j^{\mathsf{D}} \longrightarrow\!\!\triangleright S_2}{\mathbb{I} \vdash \mathbf{N}(l, y_i^{\mathsf{R}}, S_2) : S_1 \longrightarrow S_2} \quad (N)$$

Transitive closure

$$\frac{}{\mathbb{I}, \mathbb{S} \vdash [] : A \twoheadrightarrow A} \quad (I)$$

$$\frac{B \notin \mathbb{S} \quad \mathbb{I} \vdash s : A \longrightarrow B \quad \mathbb{I}, \{B\} \cup \mathbb{S} \vdash p : B \twoheadrightarrow C}{\mathbb{I}, \mathbb{S} \vdash s \cdot p : A \twoheadrightarrow C} \quad (T)$$

Reachable declarations

$$\frac{\mathbb{I}, \{S\} \vdash p : S \twoheadrightarrow S' \quad WF(p) \quad S' \longrightarrow x_i^{\mathsf{D}}}{\mathbb{I} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \quad (R)$$

Visible declarations

$$\frac{\begin{array}{c}\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \\ \forall j, p' (\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))\end{array}}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}} \quad (V)$$

Reference resolution

$$\frac{x_i^{\mathsf{R}} \longrightarrow S \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \quad (X)$$

Visibility Policies

Lexical scope

$$\mathcal{L} := \{\mathbf{P}\} \quad \mathcal{E} := \mathbf{P}^* \quad \mathbf{D} < \mathbf{P}$$

Non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{I}^? \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P}$$

Transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{TI}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{TI}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}$$

Transitive Includes

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{Inc}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}$$

Transitive includes and imports, and non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}, \mathbf{TI}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot (\mathbf{Inc} \mid \mathbf{TI})^* \cdot \mathbf{I}^?$$

$$\mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P},$$

Seen Imports

$$\dfrac{\dfrac{A_2^{\mathsf{D}}{:}S_{A_2} \in \mathcal{D}(S_{A_1}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{I}(S_{root}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{R}(S_{root}) \qquad A_1^{\mathsf{D}}{:}S_{A_1} \in \mathcal{D}(S_{root})}{A_4^{\mathsf{R}} \longmapsto A_1^{\mathsf{D}}{:}S_{A_1}}}{S_{root} \longrightarrow S_{A_1} \quad (*)}}{S_{root} \rightarrowtail A_2^{\mathsf{D}}{:}S_{A_2}} \qquad A_4^{\mathsf{R}} \in \mathcal{R}(S_{root}) \quad S_{root} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}}{A_4^{\mathsf{R}} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}}$$

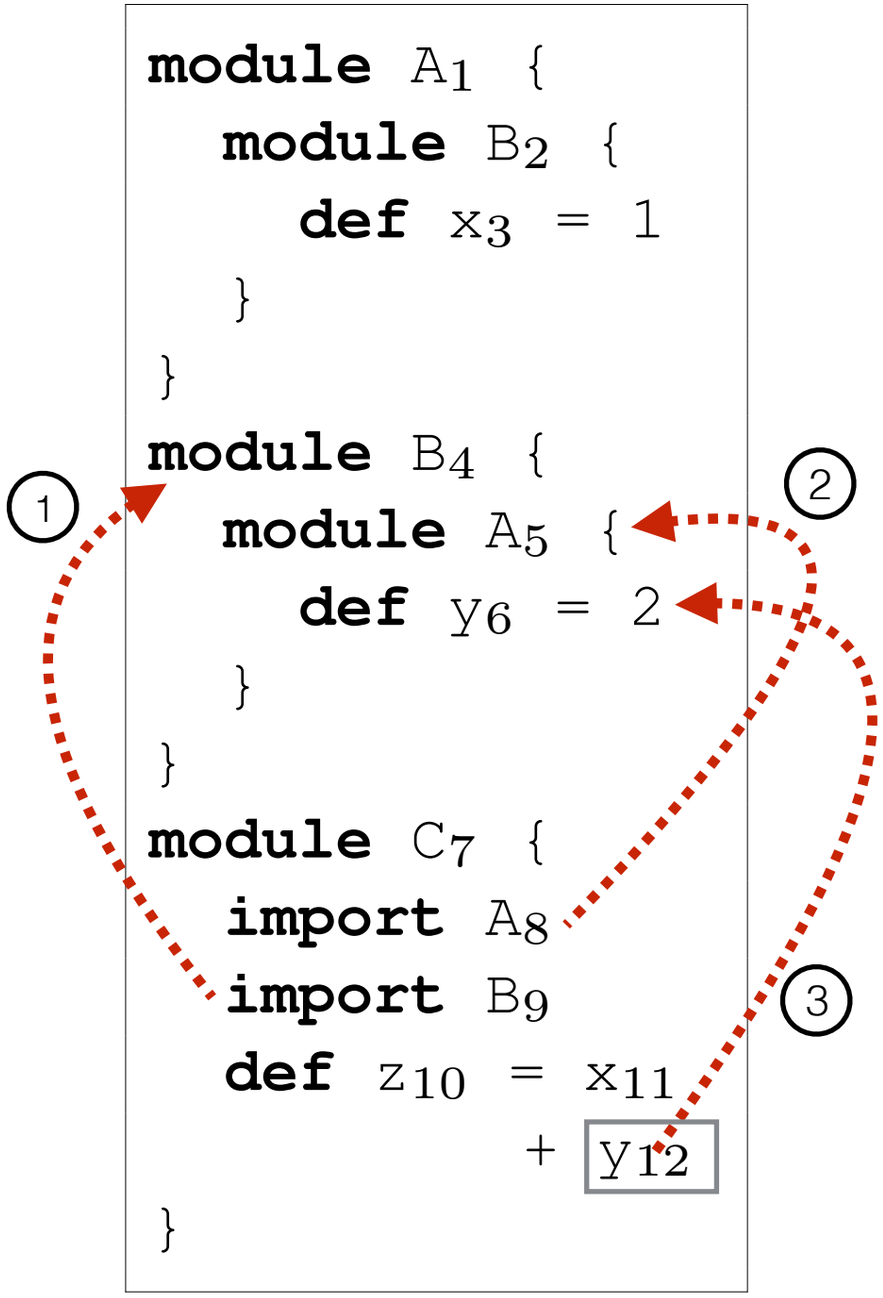
```
module A1 {
   module A2 {
      def a3 = ...
   }
}
import A4
def b5 = a6
```

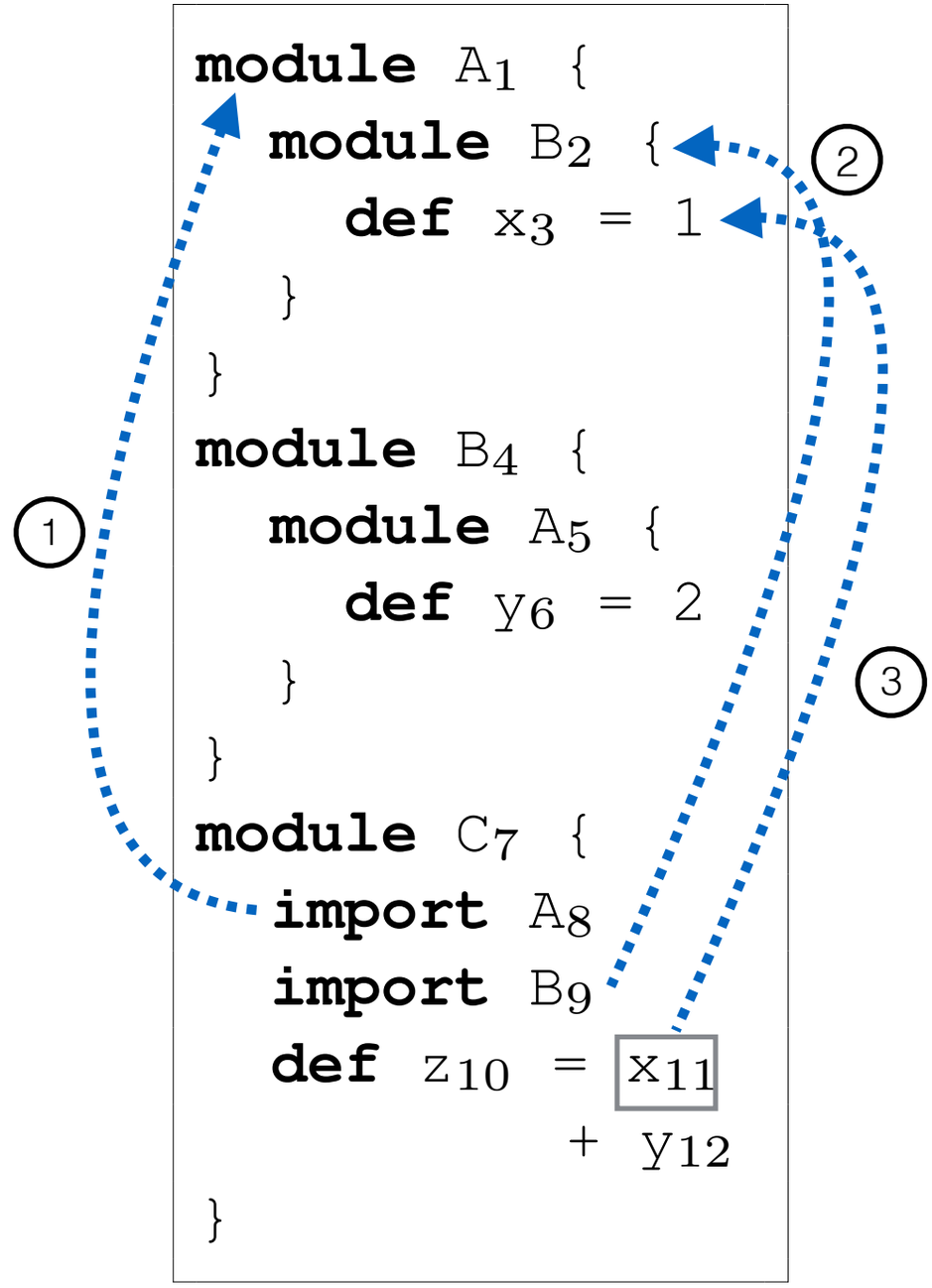
?

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}}$$

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \setminus \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbb{I} \vdash \mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2}$$

Anomaly


module A1 {
module B2 {
def x3 = 1
}
}
module B4 {
module A5 {
def y6 = 2
}
}
module C7 {
import A8
import B9
def z10 = x11
+ y12
}
1
2
3


module A1 {
module B2 {
def x3 = 1
}
}
module B4 {
module A5 {
def y6 = 2
}
}
module C7 {
import A8
import B9
def z10 = x11
+ y12
}
1
2
3

Resolution Algorithm

$$R[\mathbb{I}](x^{\mathbf{R}}) := \text{let } (r, s) = \mathit{Env}_{\mathcal{E}}[\{x^{\mathbf{R}}\} \cup \mathbb{I}, \emptyset](\mathcal{S}c(x^{\mathbf{R}}))\} \text{ in } \begin{cases} \mathsf{U} & \text{if } r = \mathsf{P} \text{ and } \{x^{\mathbf{D}} | x^{\mathbf{D}} \in s\} = \emptyset \\ \{x^{\mathbf{D}} | x^{\mathbf{D}} \in s\} \end{cases}$$

$$\mathit{Env}_{re}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{T},\ \emptyset) & \text{if } S \in \mathbb{S} \text{ or } re = \varnothing \\ \mathit{Env}_{re}^{\mathcal{L} \cup \{\mathbf{D}\}}[\mathbb{I}, \mathbb{S}](S) \end{cases}$$

$$\mathit{Env}_{re}^{L}[\mathbb{I}, \mathbb{S}](S) := \bigcup_{l \in Max(L)} \left(\mathit{Env}_{re}^{\{l' \in L | l' < l\}}[\mathbb{I}, \mathbb{S}](S) \lhd \mathit{Env}_{re}^{l}[\mathbb{I}, \mathbb{S}](S) \right)$$

$$\mathit{Env}_{re}^{\mathbf{D}}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{T},\ \emptyset) & \text{if } [] \notin re \\ (\mathsf{T},\ \mathcal{D}(S)) \end{cases}$$

$$\mathit{Env}_{re}^{l}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{P},\ \emptyset) \text{ if } S_l^{\blacktriangleright} \text{ contains a variable or } \mathit{IS}^l[\mathbb{I}](S) = \mathsf{U} \\ \bigcup_{S' \in \left(\mathit{IS}^l[\mathbb{I}](S) \cup S_l^{\blacktriangleright}\right)} \mathit{Env}_{(l^{-1}\, re)}[\mathbb{I}, \{S\} \cup \mathbb{S}](S') \end{cases}$$

$$\mathit{IS}^l[\mathbb{I}](S) := \begin{cases} \mathsf{U} & \text{if } \exists y^{\mathbf{R}} \in (S_l^{\triangleright} \setminus \mathbb{I}) \ s.t.\ R[\mathbb{I}](y^{\mathbf{R}}) = \mathsf{U} \\ \{S' \mid y^{\mathbf{R}} \in (S_l^{\triangleright} \setminus \mathbb{I}) \wedge y^{\mathbf{D}} \in R[\mathbb{I}](y^{\mathbf{R}}) \wedge y^{\mathbf{D}} \longrightarrow S'\} \end{cases}$$

Alpha Equivalence

Language-independent α-equivalence

Program similarity

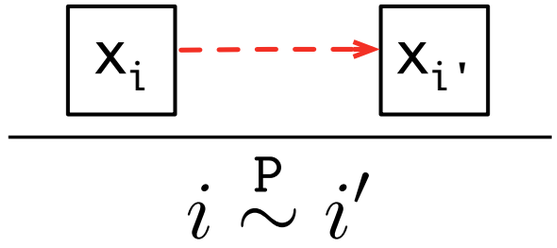
`P` $\simeq$ `P'` if have same AST ignoring identifier names

Language-independent α-equivalence

Program similarity

$\texttt{P} \simeq \texttt{P'}$ if have same AST ignoring identifier names

Position equivalence



$$\frac{i' \overset{P}{\sim} i}{i \overset{P}{\sim} i'}$$

$$\frac{i \overset{P}{\sim} i' \quad i' \overset{P}{\sim} i''}{i \overset{P}{\sim} i''}$$

$$\frac{}{i \overset{P}{\sim} i}$$

Language-independent α-equivalence

Program similarity

$\texttt{P} \simeq \texttt{P'}$ if have same AST ignoring identifier names

Position equivalence

$$\frac{}{i \overset{\texttt{P}}{\simeq} i'} \qquad \frac{i' \overset{\texttt{P}}{\simeq} i}{i \overset{\texttt{P}}{\simeq} i'} \qquad \frac{i \overset{\texttt{P}}{\simeq} i' \quad i' \overset{\texttt{P}}{\simeq} i''}{i \overset{\texttt{P}}{\simeq} i''} \qquad \frac{}{i \overset{\texttt{P}}{\simeq} i}$$

Alpha equivalence

$$\texttt{P1} \overset{\alpha}{\approx} \texttt{P2} \triangleq \texttt{P1} \simeq \texttt{P2} \wedge \forall\ e\ e',\ e \overset{\texttt{P1}}{\simeq} e' \Leftrightarrow e \overset{\texttt{P2}}{\simeq} e'$$

(with some further details about free variables)

Preserving ambiguity

```
module A1 {
  def x2 := 1
}
module B3 {
  def x4 := 2
}
module C5 {
  import A6 B7;
  def y8 := x9
}
module D10 {
  import A11;
  def y12 := x13
}
module E14 {
  import B15;
  def y16 := x17
}
```

P1

```
module AA1 {
  def z2 := 1
}
module BB3 {
  def z4 := 2
}
module C5 {
  import AA6 BB7;
  def s8 := z9
}
module D10 {
  import AA11;
  def u12 := z13
}
module E14 {
  import BB15;
  def v16 := z17
}
```

P2

```
module A1 {
  def z2 := 1
}
module B3 {
  def x4 := 2
}
module C5 {
  import A6 B7;
  def y8 := z9
}
module D10 {
  import A11;
  def y12 := z13
}
module E14 {
  import B15;
  def y16 := x17
}
```

P3

$P1 \overset{\alpha}{\approx} P2$ $\quad P2 \not\overset{\alpha}{\approx} P3$

Names and Types

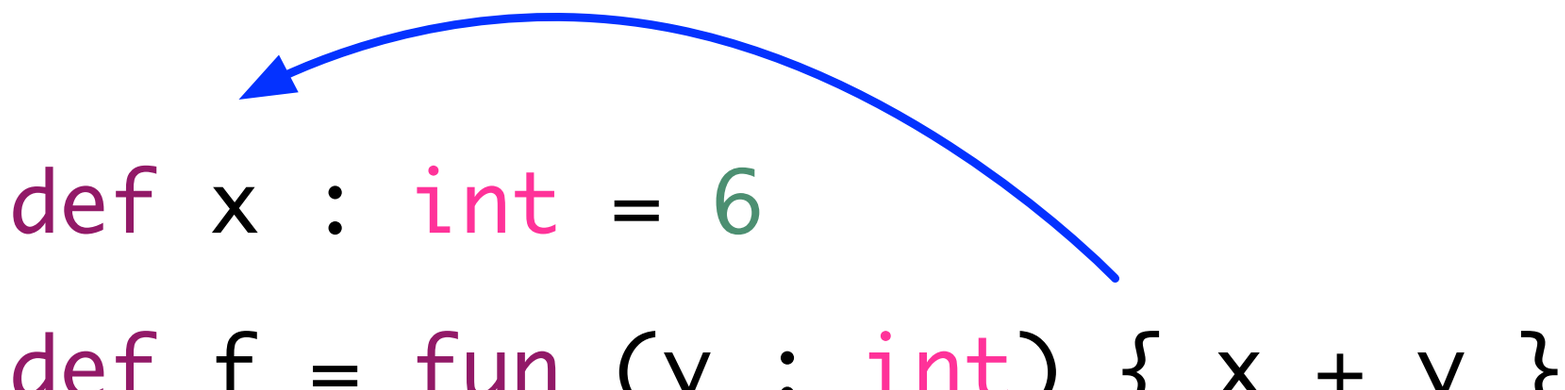
Types from Declaration

```
def x : int = 6

def f = fun (y : int) { x + y }
```

Static type-checking (or inference) is one obvious client for name resolution

In many cases, we can perform resolution **before** doing type analysis

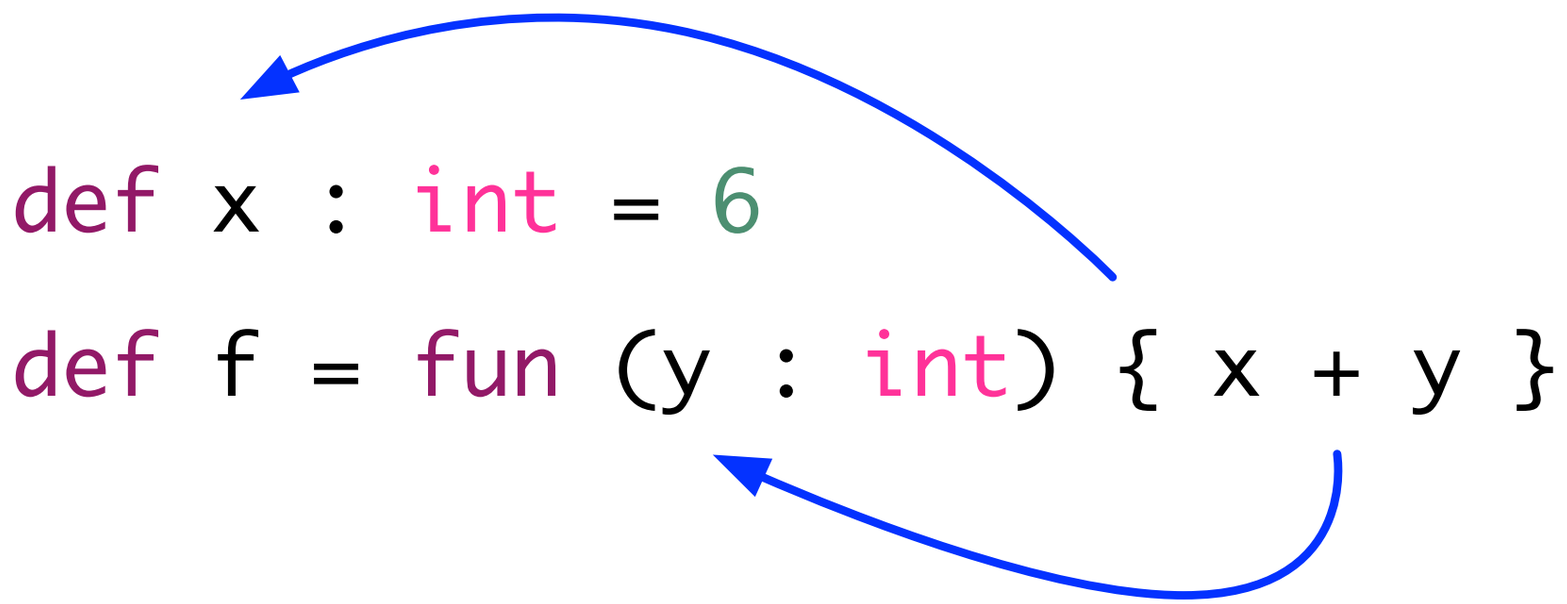
Types from Declaration



Static type-checking (or inference) is one obvious client for name resolution

In many cases, we can perform resolution **before** doing type analysis

Types from Declaration



Static type-checking (or inference) is one obvious client for name resolution

In many cases, we can perform resolution **before** doing type analysis

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A₁ { x₁ : int }
record B₁ { a₁ : A₂ ; x₂ : bool}

def z₁ : B₂ = ...

def y₁ = z₂.x₃

def y₂ = z₃.a₂.x₄
```

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Type-Dependent Name Resolution

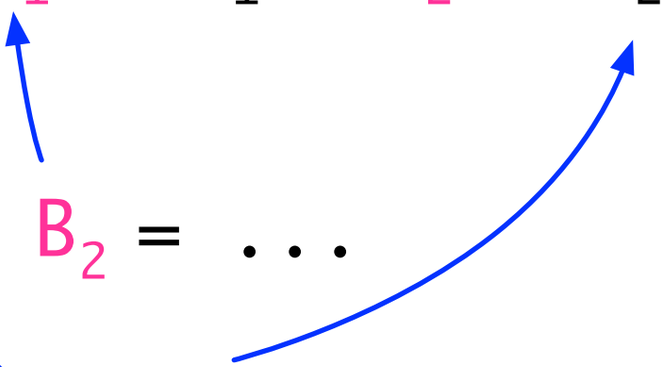
But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



Type-Dependent Name Resolution

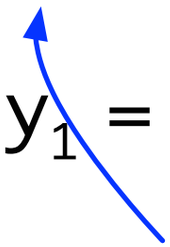
But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

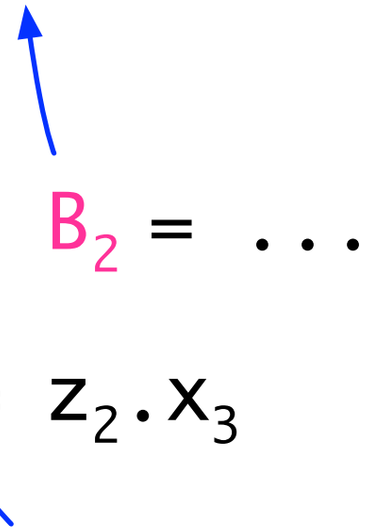
```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



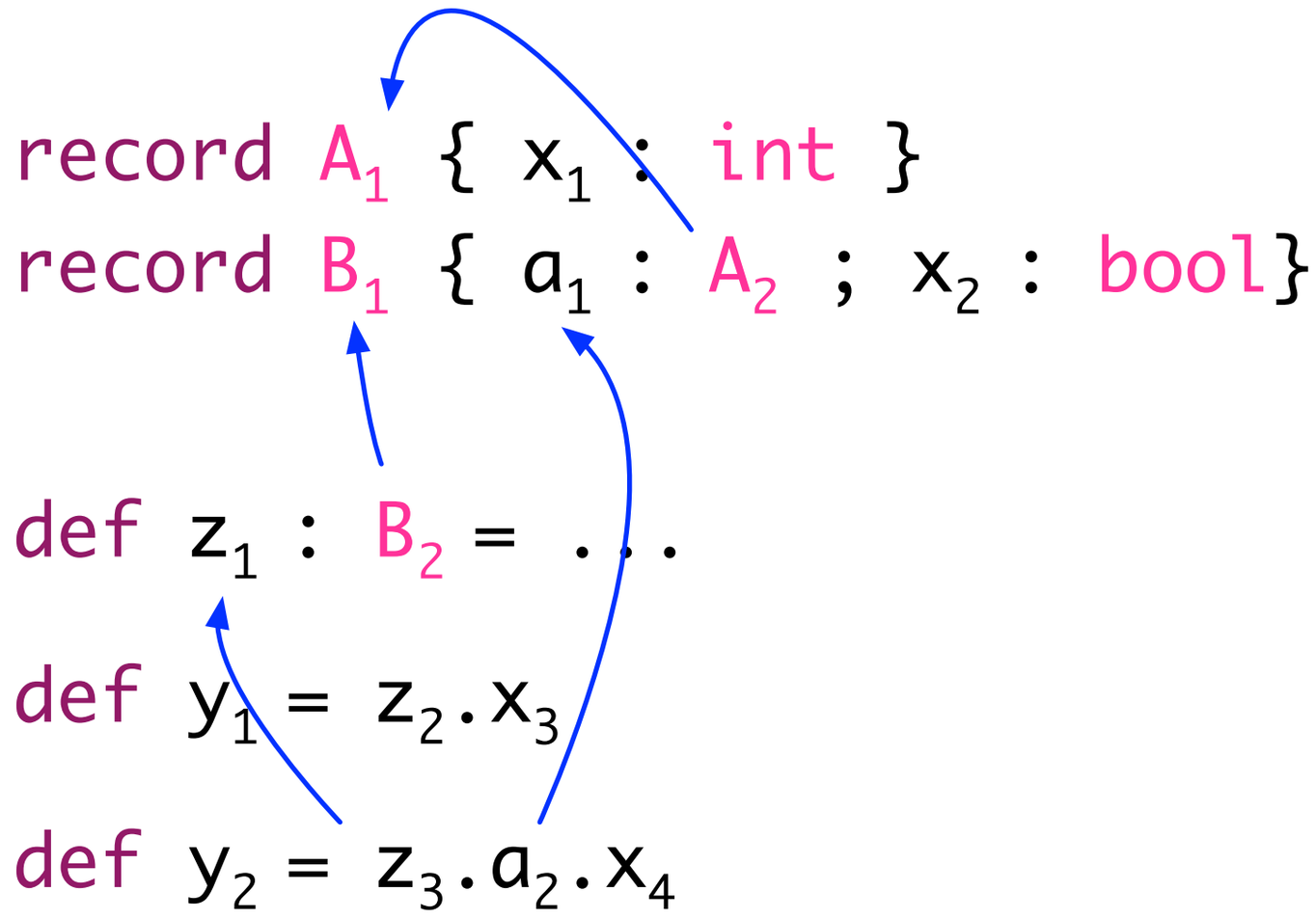
Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

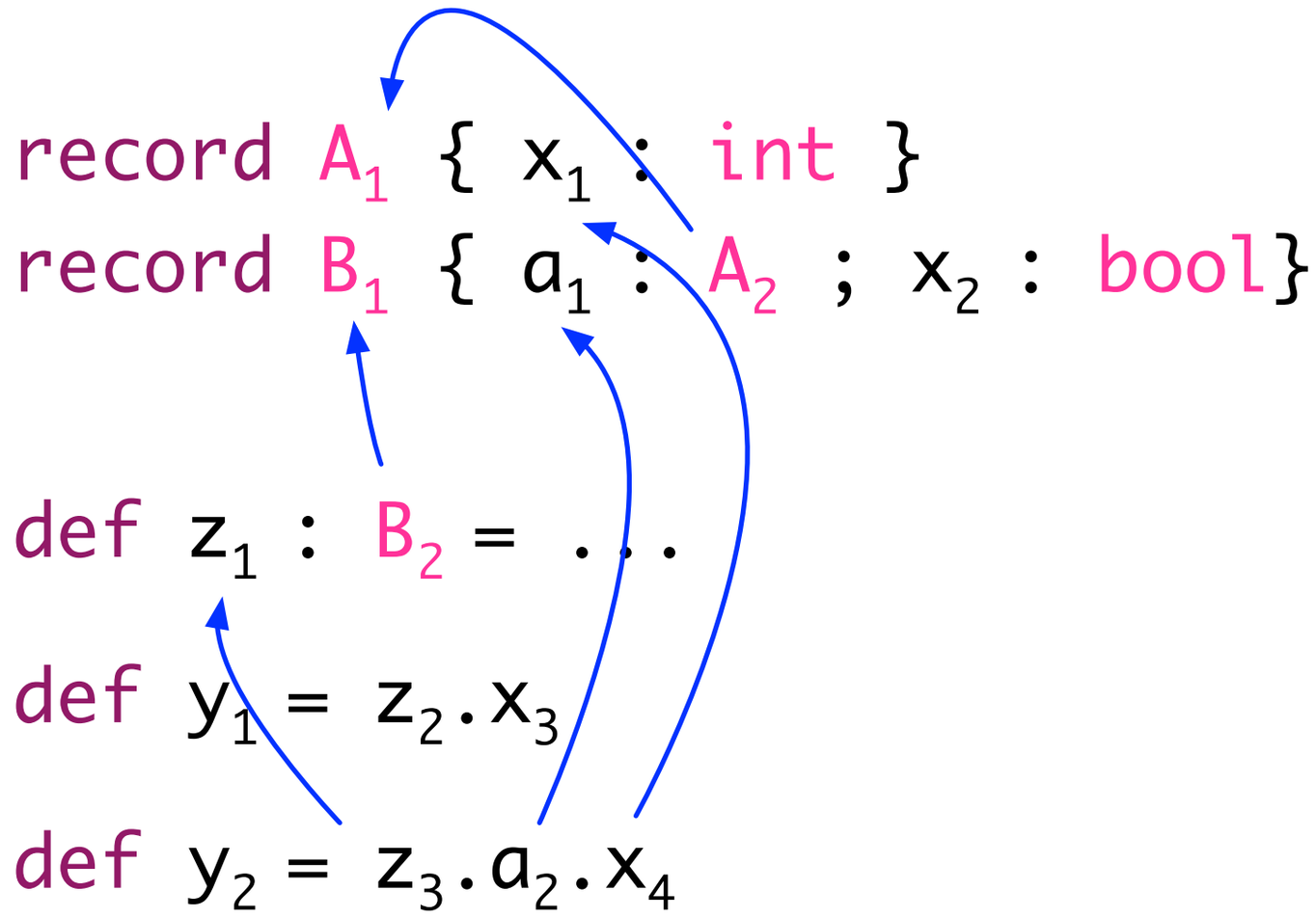
Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution



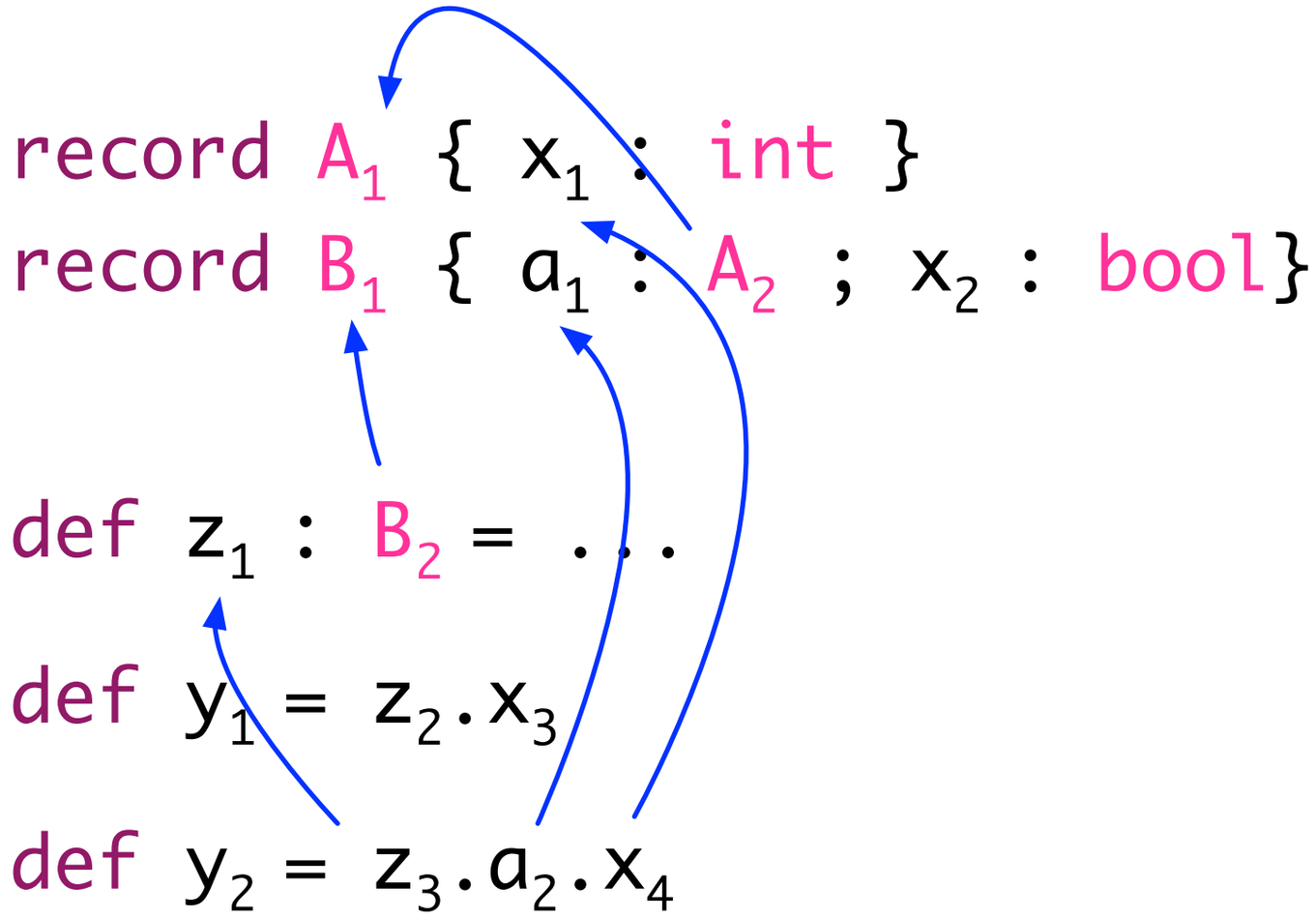
Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution



Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution



Our approach: interleave **partial** name resolution with type resolution (also using constraints)

See PEPM 2016 paper / talk

More on types next week

Closing

Summary: A Theory of Name Resolution

- **Representation: Scope Graphs**
  - Standardized representation for lexical scoping structure of programs
  - Path in scope graph relates reference to declaration
  - Basis for syntactic and semantic operations
- **Formalism: Name Binding Constraints**
  - References + Declarations + Scopes + Reachability + Visibility
  - Language-specific rules map AST to constraints
- **Language-Independent Interpretation**
  - Resolution calculus: correctness of path with respect to scope graph
  - Name resolution algorithm
  - Alpha equivalence
  - Mapping from graph to tree (to text)
  - Refactorings
  - And many other applications

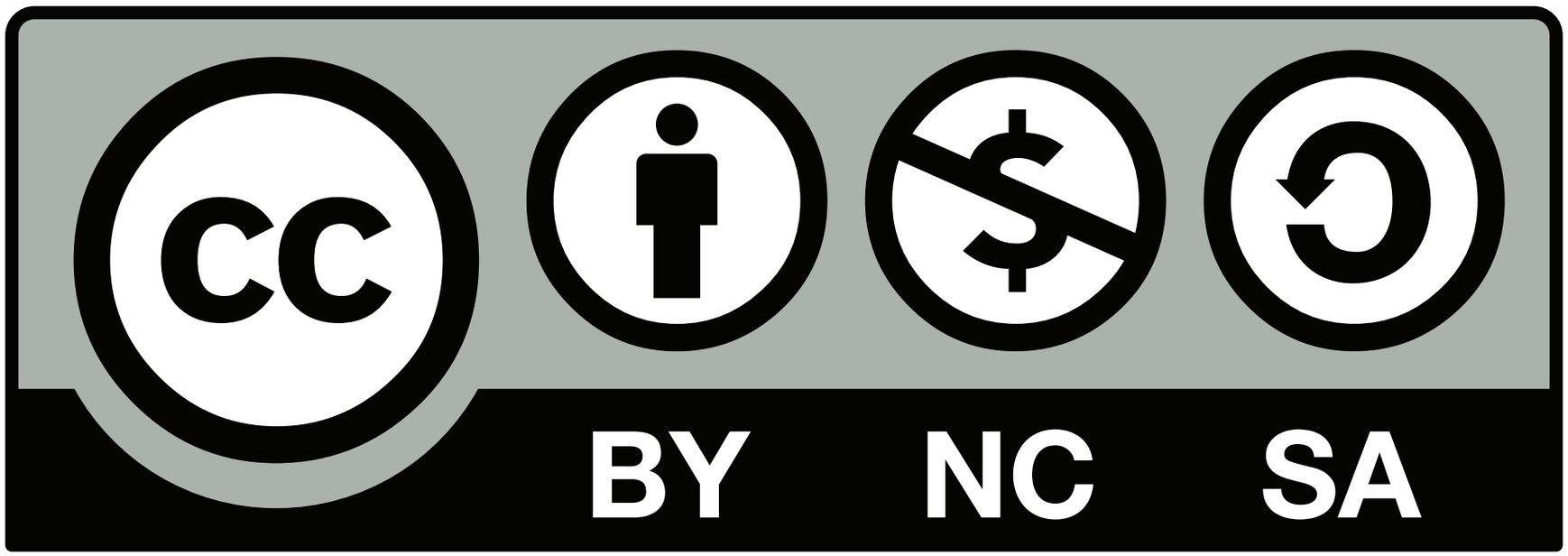
Validation

- **We have modeled a large set of example binding patterns**
  - definition before use
  - different let binding flavors
  - recursive modules
  - imports and includes
  - qualified names
  - class inheritance
  - partial classes
- **Next goal: fully model some real languages**
  - Java
  - ML

Future Work

- **Scope graph semantics for binding specification languages**
  - starting with NaBL
  - or rather: a redesign of NaBL based on scope graphs
- **Resolution-sensitive program transformations**
  - renaming, refactoring, substitution, …
- **Dynamic analogs to static scope graphs**
  - how does scope graph relate to memory at run-time?
- **Supporting mechanized language meta-theory**
  - relating static and dynamic bindings

copyrights


BY
NC
SA

Pictures

copyrights

Slide 1:

Reference by Ian Charleton, some rights reserved