Executing Declarative Language Definitions

Eelco Visser

TU Delft

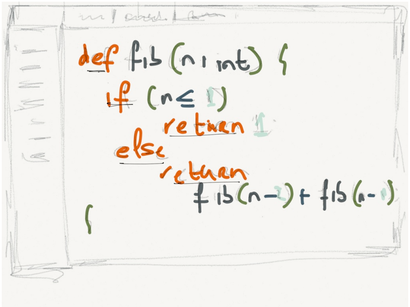
MiniKanren 2020 | August 27, 2020

```
def fib(n : int) {
    if (n ≤ 1)
        return 1
    else
        return
            fib(n-2) + fib(n-1)
}
```

```
def fib (n : int) {
  if (n ≤ 1)
    return 1
  else
    return
      fib(n-2) + fib(n-1)
}
```

Desktop — bash — 37×16

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

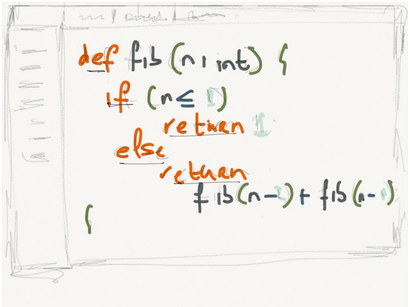


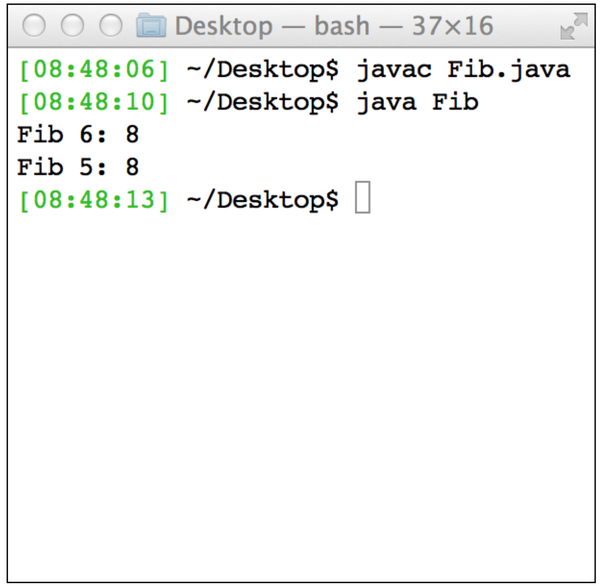
Fib.java

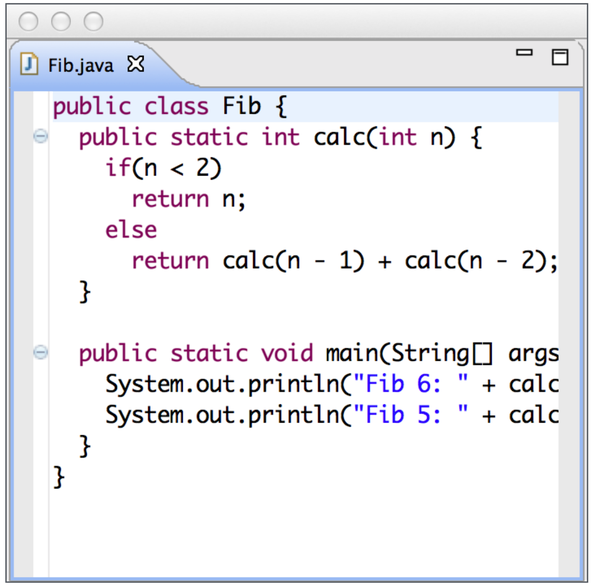
```
public class Fib {
  public static int calc(int n) {
    if(n < 2)
      return n;
    else
      return calc(n - 1) + calc(n - 2);
  }

  public static void main(String[] args
    System.out.println("Fib 6: " + calc
    System.out.println("Fib 5: " + calc
  }
}
```

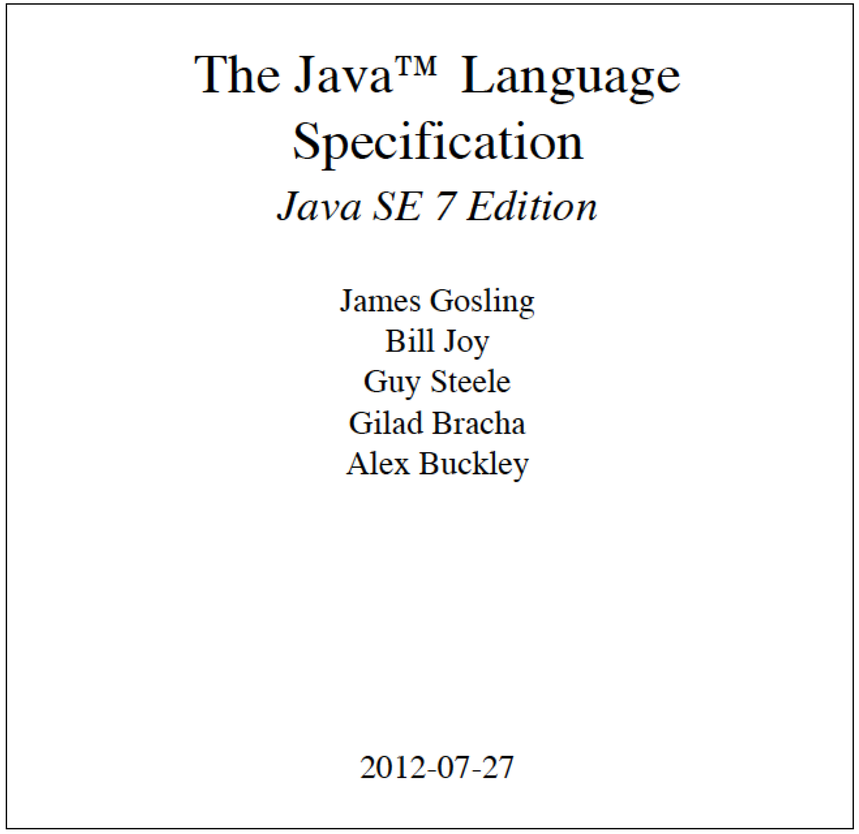
Desktop — bash — 37×16

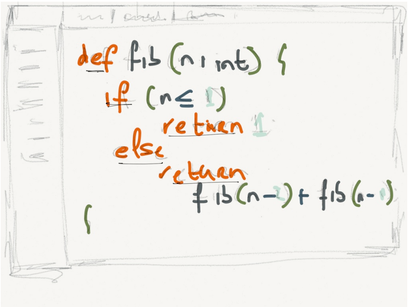
```
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$
```

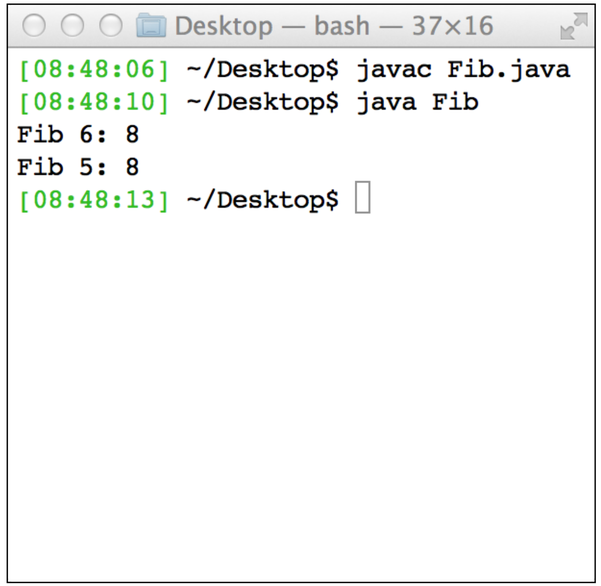

def fib (n: int) {
if (n ≤ 1)
return 1
else
return
fib(n-2) + fib(n-1)
}

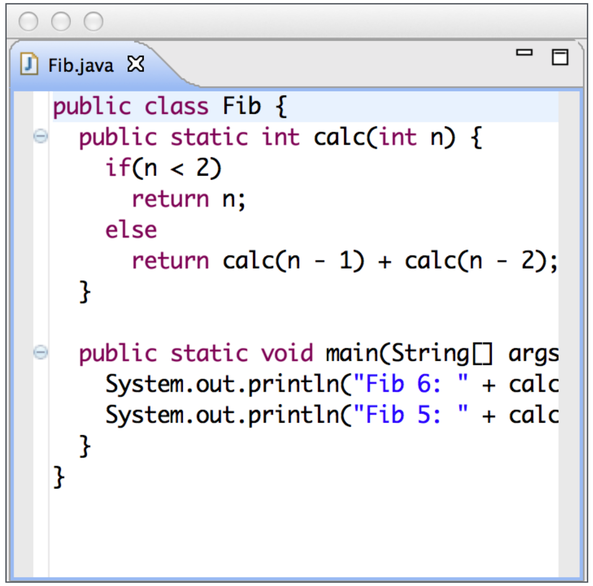

Desktop — bash — 37×16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$

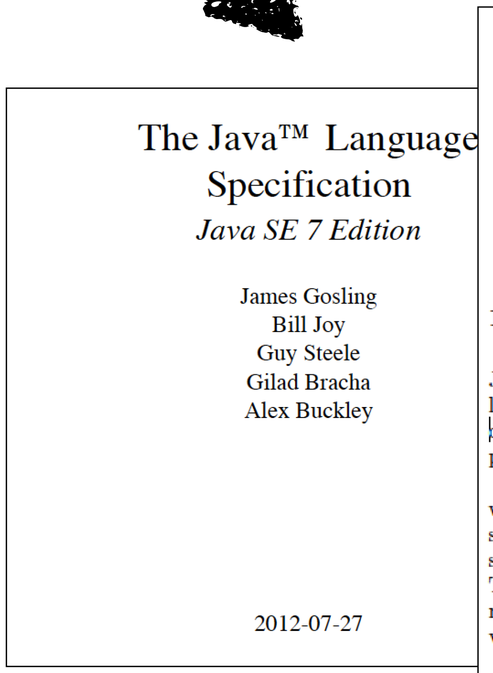

Fib.java
public class Fib {
public static int calc(int n) {
if(n < 2)
return n;
else
return calc(n - 1) + calc(n - 2);
}
public static void main(String[] args
System.out.println("Fib 6: " + calc
System.out.println("Fib 5: " + calc
}
}


The Java™ Language Specification
Java SE 7 Edition
James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
2012-07-27







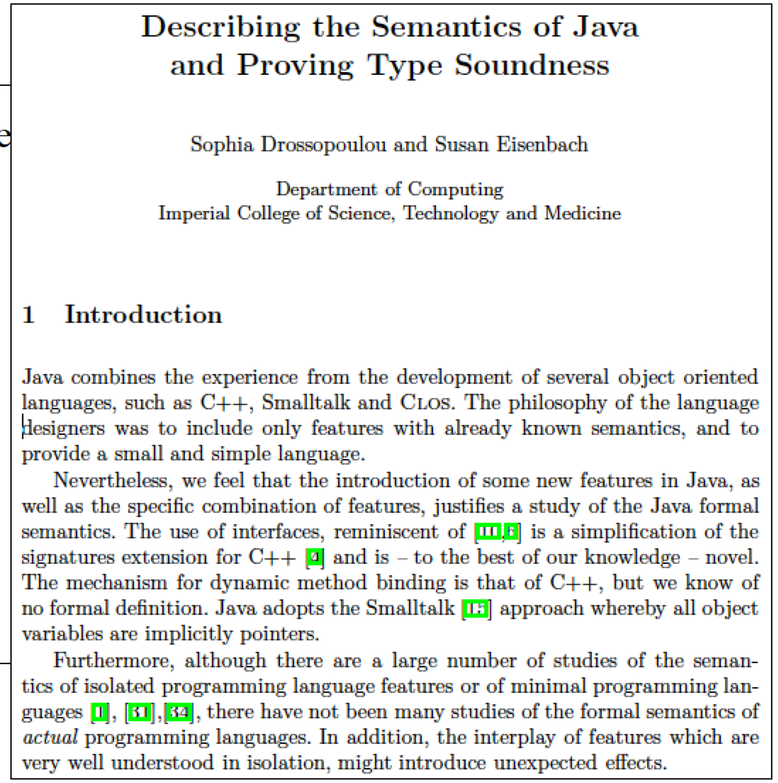


The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27



Describing the Semantics of Java and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach

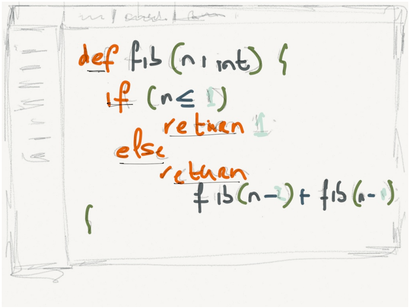
Department of Computing
Imperial College of Science, Technology and Medicine

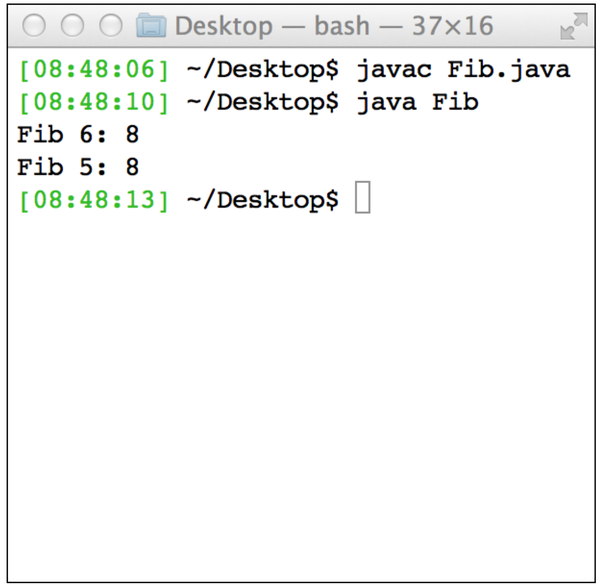
1 Introduction

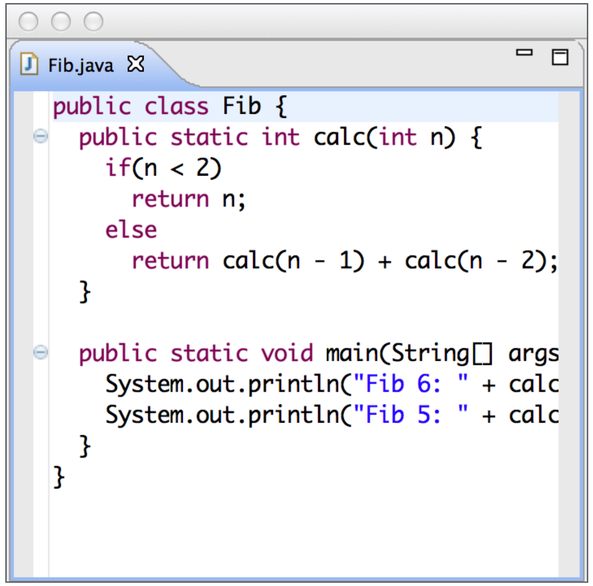
Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and CLOS. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.

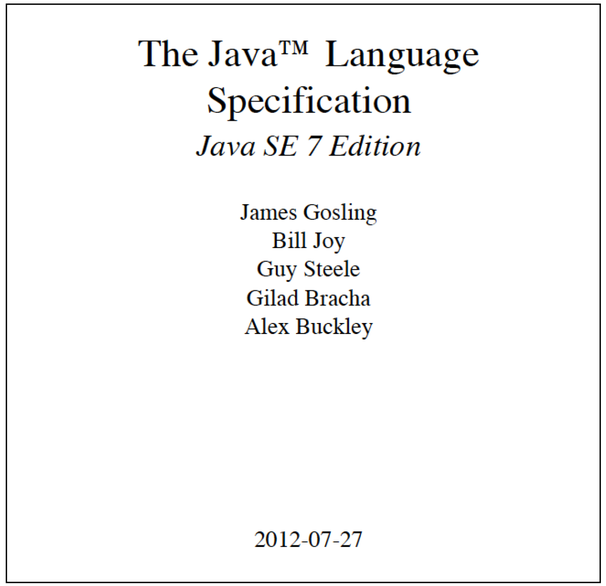
Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,8] is a simplification of the signatures extension for C++ [9] and is – to the best of our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

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







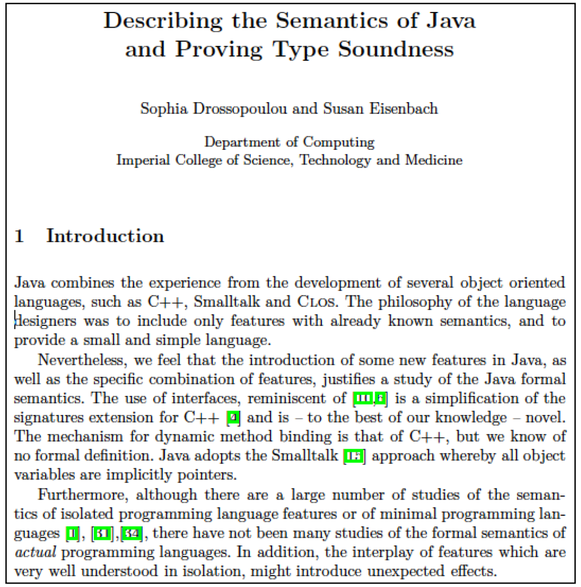


The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27



Describing the Semantics of Java and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine

1 Introduction

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

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,6] is a simplification of the signatures extension for C++ [9] and is – to the best of our knowledge – novel. The mechanism for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk [13] approach whereby all object variables are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], [31],[32], there have not been many studies of the formal semantics of *actual* programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.

Desktop — bash — 37×16

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

Fib.java

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

  public static void main(String[] args
    System.out.println("Fib 6: " + calc
    System.out.println("Fib 5: " + calc
  }
}
```

The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine

1 Introduction

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

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,8] is a simplification of the signatures extension for C++ [9] and is – to the best of our knowledge – novel. The mechanism for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk [12] approach whereby all object variables are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], [31],[32], there have not been many studies of the formal semantics of *actual* programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.

parser

type checker

code generator

interpreter

parser

error recovery

syntax highlighting

outline

code completion

navigation

type checker

debugger

syntax definition

static semantics

dynamic semantics

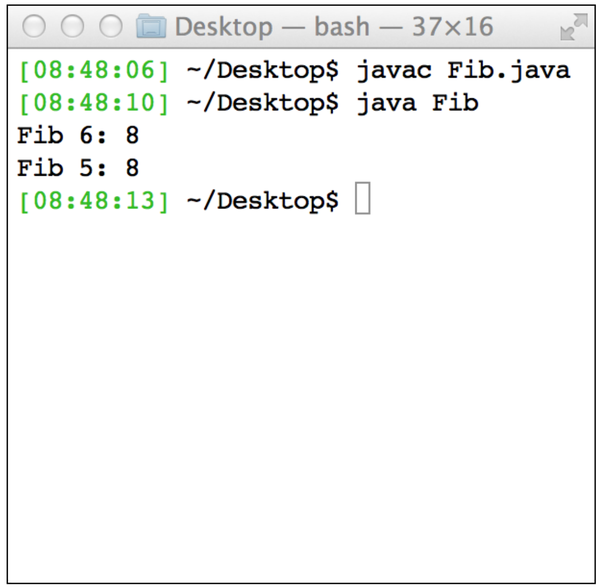
abstract syntax

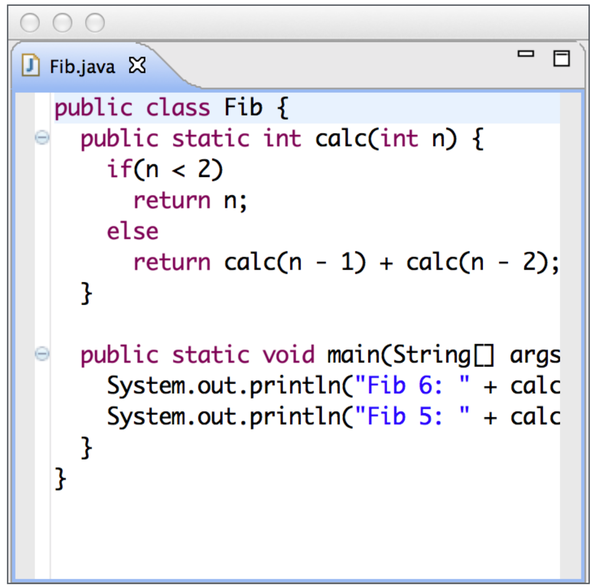
type system

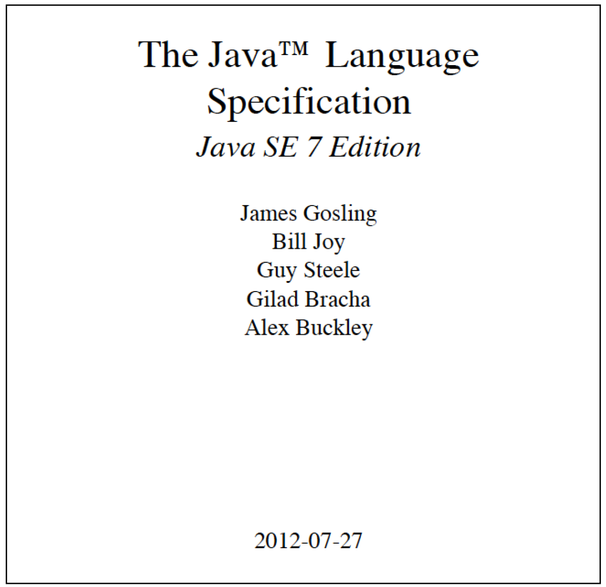
operational semantics

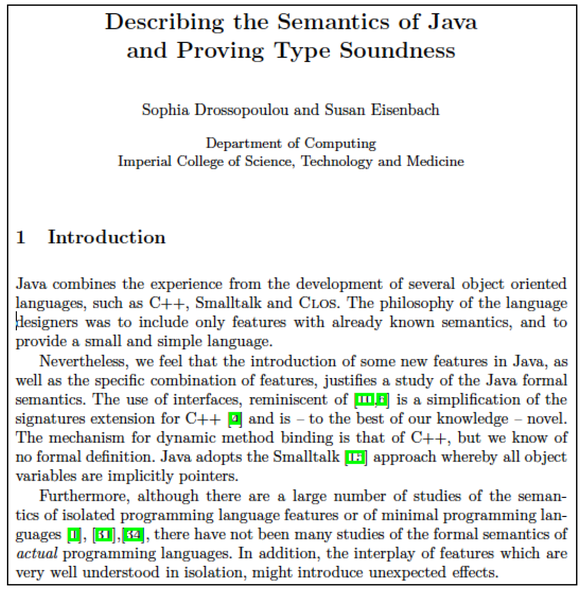
type soundness proof

Language Design







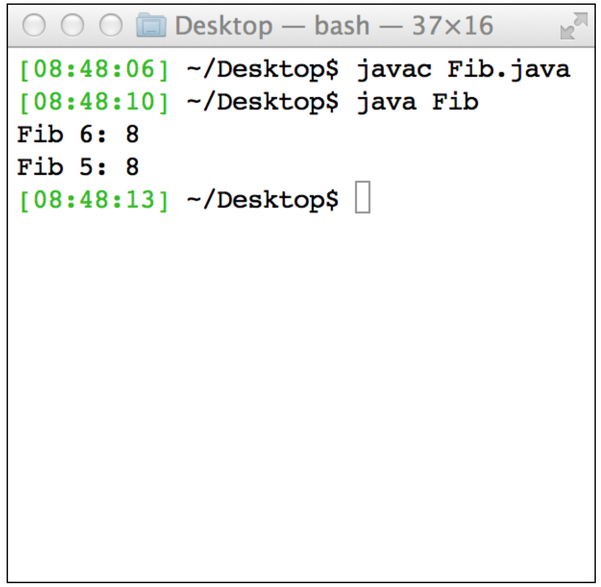


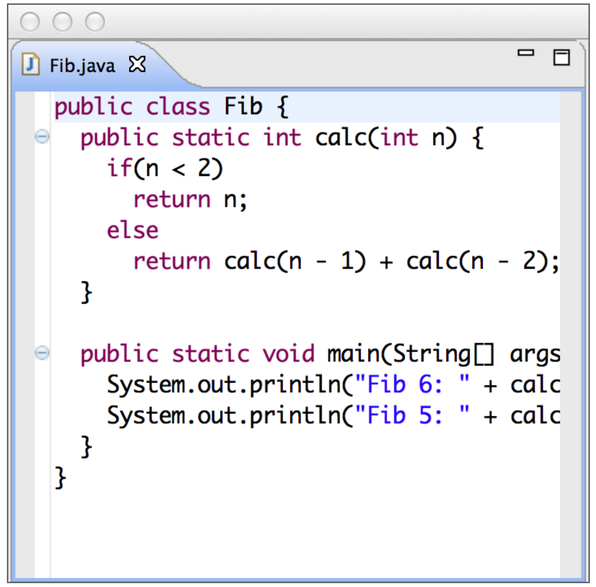
Language Engineering

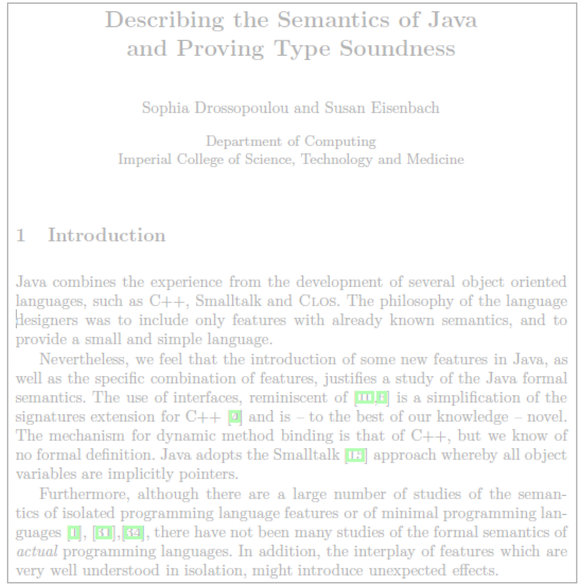
Syntax Checker

Type Checker

Code Generator







Semantics Engineering

Abstract Syntax

Static Semantics

Dynamic Semantics

Transform

```
Desktop — bash — 37×16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
[08:48:13] ~/Desktop$
```

Fib.java

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

  public static void main(String[] args
    System.out.println("Fib 6: " + calc
    System.out.println("Fib 5: " + calc
  }
}
```

The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach

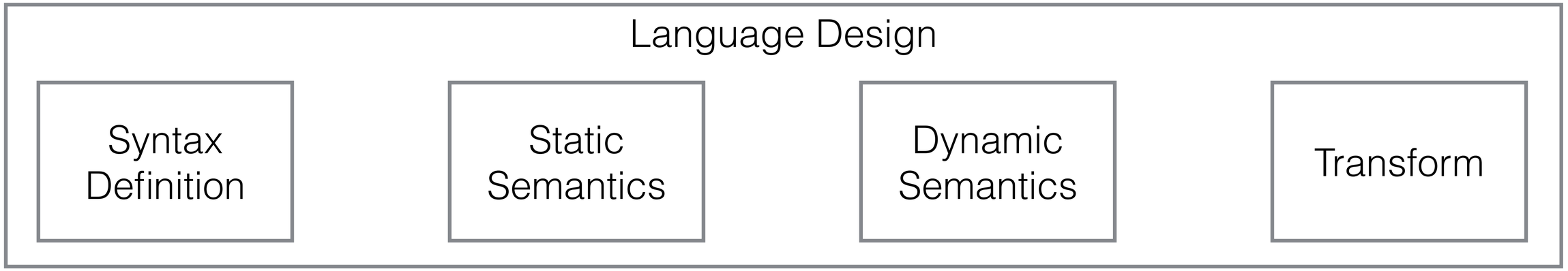
Department of Computing
Imperial College of Science, Technology and Medicine

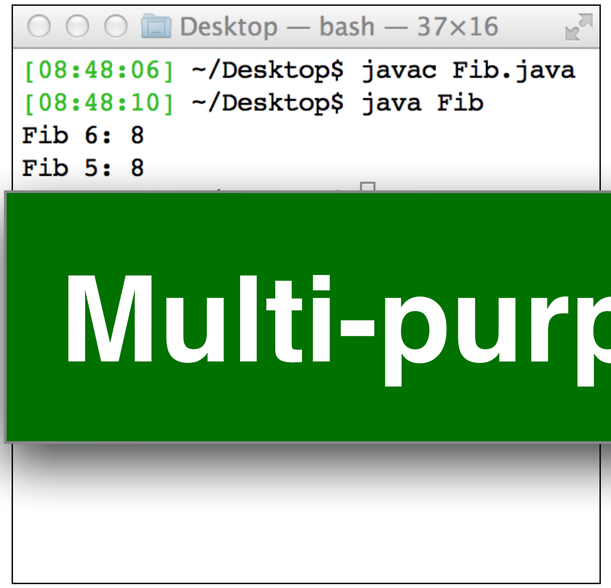
1 Introduction

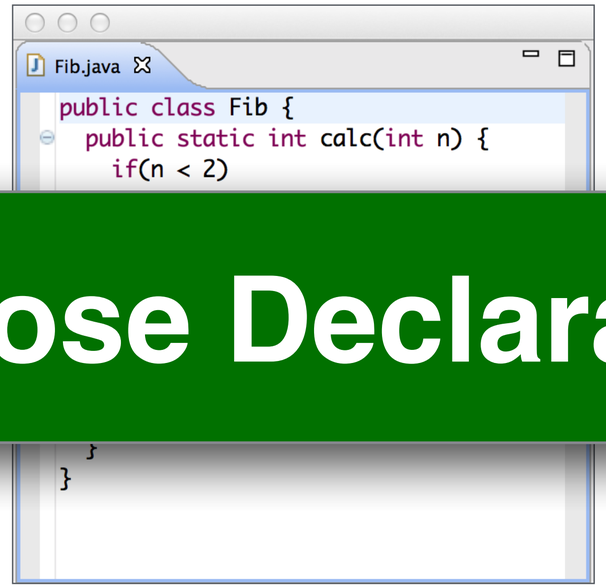
Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and CLOS. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.

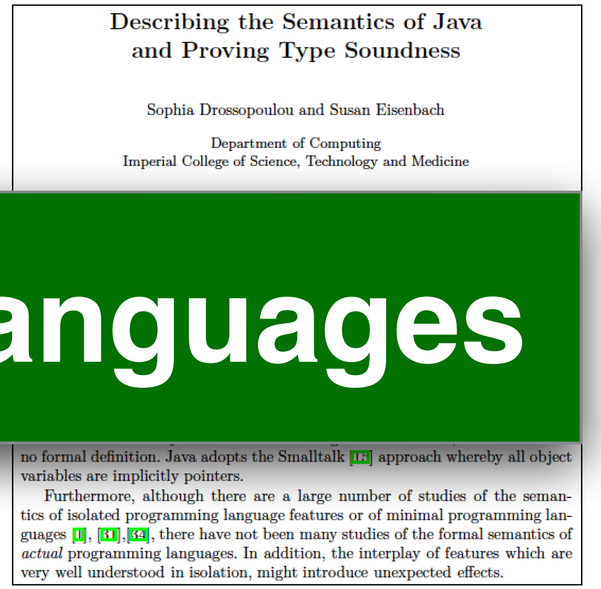
Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,8] is a simplification of the signatures extension for C++ [9] and is – to the best of our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

Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], [31],[32], there have not been many studies of the formal semantics of *actual* programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.









Multi-purpose Declarative Meta-Languages

A Recipe for Declarative Meta-Languages

Representation

- Language-independent representation

Specification Formalism

- Language-specific rules

Declarative Semantics

- Model (instance of representation) satisfies specification

Language-Independent Tools

- Reusable language-independent tools operate on representation
- Provide multiple interpretations
- Sound wrt declarative semantics

The Spoofax Language Workbench

What is Spoofax?

A tool for implementing programming languages

- Open source and freely available
- Used in education, research, and industry
- Requires a lot of software engineering to maintain

A long term research project

- Incubator for language engineering research
- Basis for implementation and evaluation of 100+ papers
- Imperfect approximation of a language designer's workbench

Spoofax History

Stratego/XT [1997-2005]

- Command-line language processors with SDF + Stratego (C/Linux)

Spoofax [2006-2011]

- Eclipse (IMP) IDEs from SDF + Stratego + ESV (Java)

Spoofax 2 [2012-2020]

- Spoofax-Core library with bindings for Eclipse, IntelliJ, command-line
- Meta-Languages: SDF3, Stratego, NaBL, NaBL2, Statix, DynSem, …

Spoofax 3 [2019-…]

- Live language development based on PIE build system

Spoofax in Action

Research

- Language Engineering, Language Prototyping

Education

- Compiler Construction (MiniJava)
- Language Engineering Project (2020: Ada, C, ChocoPy, FlowSpec)

Academic Workflow Engineering

- WebDSL (researchr.org, WebLab, …)

Industry

- Oracle Labs: Graph Analytics
- Canon: Several DSLs
- Philips: Software Restructuring (we're hiring PhD students!)

Meta-Languages for Language Definition

Meta-languages

- **Syntax definition with SDF3**
- **Static semantics with Statix**
- Data-flow analysis with FlowSpec
- Transformation with Stratego
- Dynamic Semantics with DynSem/Dynamix
- Editor service definition with ESV

Declarative Syntax Definition with SDF3

Declarative Syntax Definition

Representation

- Syntax trees

Specification Formalism: SDF3

- Productions + Constructors + Templates + Disambiguation

Declarative Semantics

- Well-formedness of syntax trees wrt syntax definition

Language-Independent Tools

- Parser
- Formatting based on layout hints in grammar
- Syntactic completion

Syntax = Structure

```
module structure

imports Common

context-free start-symbols Exp

context-free syntax

  Exp.Var = ID

  Exp.Int = INT

  Exp.Add = Exp "+" Exp

  Exp.Fun = "function" "(" {ID ","}* ")" "{" Exp "}"

  Exp.App = Exp "(" {Exp ","}* ")"

  Exp.Let = "let" Bnd* "in" Exp "end"

  Bnd.Bnd = ID "=" Exp
```

```
let
  inc = function(x) { x + 1 }
 in
  inc(3)
end
```

```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), [Int("3")])
)
```

Parsing = Formatting^{-1}

```
context-free syntax

  Exp.Var = <<ID>>

  Exp.Int = <<INT>>

  Exp.Add = <<Exp> + <Exp>>

  Exp.Fun = <
    function(<{ID ","}*>){
      <Exp>
    }
  >

  Exp.App = <<Exp>(<{Exp ","}*>)>

  Exp.Let = <
    let
      <Bnd*>
     in
      <Exp>
    end
  >

  Bnd.Bnd = <<ID> = <Exp>>
```

```
let
  inc = function(x) { x + 1 }
 in
  inc(3)
end
```

```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), [Int("3")])
)
```

```
let
  inc = function(x){
    x + 1
  }
 in
  inc(3)
end
```

Completion = Rewrite(Incomplete Structure)

```
class A {

  public int m() {
    int x;
    x = $Exp;
    return
  }
}
```

+Add
+Sub
+Mul
+Lt
+VarRef

$Exp + $Exp

```
class A {

  public int m() {
    int x;
    x = $Exp + $Exp;
    retu
  }
}
```

+Add
+Sub
+Mul
+Lt
+VarRef

$Exp + $Exp

```
class A {

  public int m() {
    int x;
    x = 21 + $Exp;
    return x;
  }
}
```

+Add
+Sub
+Mul
+Lt
+VarRef

($Exp + $Exp)

```
class A {

  public int m() {
    int x;
    x = 21 + 21;
    return x;
  }
}
```

Disambiguation

Ambiguity = Multiple Possible Parses

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>>

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>>

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>>
```

```
a + b + c
```

```
amb(
  [ Add(Var("a"), Add(Var("b"), Var("c")))
  , Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)
```

Disambiguation = Select(Structure)

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>>

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>>

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>>
```

```
a + b + c
```

```
amb(
  [ Add(Var("a"), Add(Var("b"), Var("c")))
  , Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)
```

```
Add(Add(Var("a"), Var("b")), Var("c"))
```

Declarative Disambiguation = Separate Concern

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

Associativity = Solve Intra Operator Ambiguity

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
a + b + c
```

```
amb(
  [ Add(Var("a"), Add(Var("b"), Var("c")))
  , Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)
```

```
Add(Add(Var("a"), Var("b")), Var("c"))
```

Priority = Solve Inter Operator Ambiguity

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
f a + b
```

```
amb(
  [ Add(App(Var("f"), Var("a")), Var("b"))
  , App(Var("f"), Add(Var("a"), Var("b")))
  ]
)
```

```
Add(App(Var("f"), Var("a")), Var("b"))
```

Dangling Else = Operators with Overlapping Prefix

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
if(1) if(2) 3 else 4
```

```
amb(
  [ IfElse(
      Int("1")
    , If(Int("2"), Int("3"))
    , Int("4")
    )
  , If(
      Int("1")
    , IfElse(Int("2"), Int("3"), Int("4"))
    )
  ]
)
```

```
If(
  Int("1")
, IfElse(Int("2"), Int("3"), Int("4"))
)
```

Parenthesize

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
(a + b) + c
```

↓

```
Add(Add(Var("a"), Var("b")), Var("c"))
```

↓

```
a + b + c
```

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
a + (let x = b in (c + d))
```

```
Add(
  Var("a")
, Let(
    [Bnd("x", Var("b"))]
  , Add(Var("c"), Var("d"))
  )
)
```

```
a + let
  x = b
in
  c + d
```

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
(a + (let x = b in c)) + d
```

```
Add(
  Add(
    Var("a")
  , Let([Bnd("x", Var("b"))], Var("c"))
  )
, Var("d")
)
```

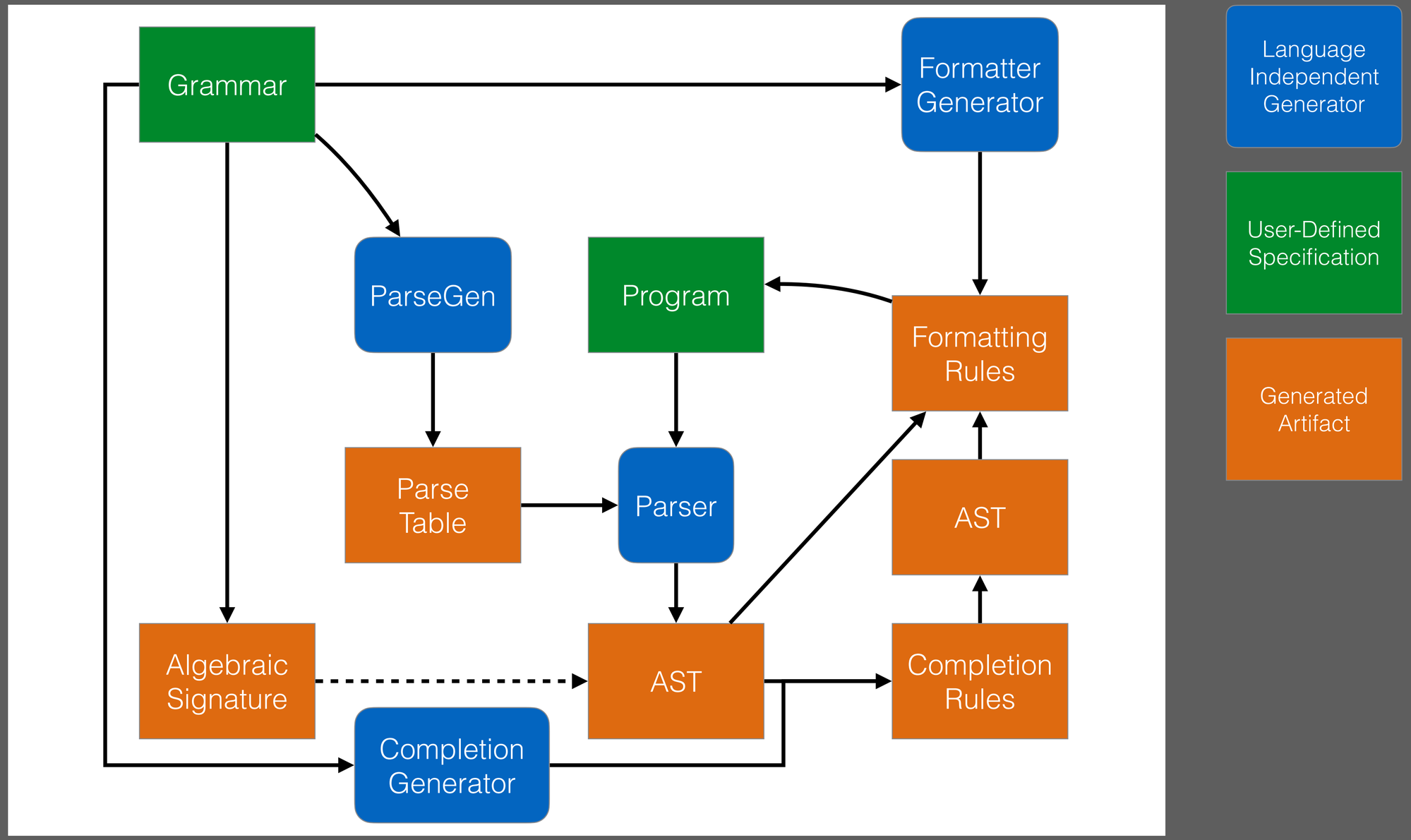
```
a + (let
  x = b
in
  c) + d
```

SDF3 Interpretations

```
Statement.If = <
  if(<Exp>)
    <Statement>
  else
    <Statement>
>
```

- Parser
- Error recovery
- Pretty-printer
- Abstract syntax tree schema
- Syntactic coloring
- Syntactic completion
- Folding rules
- Outline rules

Generating Artifacts from Syntax Definitions



Declarative Type System Specification with Statix

Declarative Static Semantics Definition with Statix

Representation

- Scope graph

Specification Formalism: Statix

- Type constraints + scope graph constraints + resolution policies

Declarative Semantics

- Scope graph of program satisfies specification

Language-Independent Tools

- Type checking
- Refactoring / Renaming
- Code completion

Demonstration

workspace-2020-07-29-LEP - london.example/07-modules.mod - Eclipse IDE

Package Ex

- type.stx
- > variable.stx
- syntax
- target
- trans
- metaborg.yaml
- pom.xml
- README.md
- > london.example [statix-s
- JRE System Library [Jav
- Maven Dependencies
- modules
  - module01.aterm
  - module01.mod
  - module02.aterm
  - module02.mod
  - module03.mod
- target
- 00-sandbox.aterm
- > 00-sandbox.mod
- 00-sandbox2.aterm
- 00-sandbox2.mod
- 01-numbers.aterm
- 01-numbers.mod
- 02-booleans.aterm
- 02-booleans.mod
- 03-variables.mod
- 04-types.mod
- 05-functions.mod
- 06-records.aterm
- 06-records.mod
- 07-modules.aterm
- > 07-modules.mod
- 08-extend.mod
- metaborg.yaml
- pom.xml
- london.syntax.test [statix-s
- london.test [statix-sandbox

07-modules.mod | 03-variables.mod | 05-functions.mod | 06-records.mod | 08-extend.mod | 02-booleans.mod

```
module Library{

  module Sig {
    type Pred = Int -> Bool
  }

  type P = Sig.Pred

  module Odd {
    import Even
    def odd : P =
        fun(x) if x == 0 then false else even (x - 1)
  }

  module Even {
    import Odd
    def even : Sig.Pred =
        fun(x) if x == 0 then true else odd (x - 1)
  }
}

module Application {

  import Library.Even
//  def alias = Library  // error

  $ even 42
  $ Library.Odd.odd 45
}
```

Writable | Insert | 30 : 1

Logic Programming

Predicates Represent Program Properties

```
rules // type of ...

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Use `maps` to apply a predicate to all elements of a list

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

`typeOfExp(s, e) == T`
expression `e` has type `T` in scope `s`

`typeOfType(s, t) == T`
syntactic type `t` has semantic type `T` in scope `s`

`declOk(s, d)`
declaration `d` is well-defined (Ok) in scope `s`

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

`typeOfExp(s, e) == T`

expression `e` has type `T` in scope `s`

`typeOfExp(s, e, T)`

expression `e` has type `T` in scope `s`

One expression has one type

One expression can have multiple types

(Solver does not match on type argument)

Predicates are Defined by Rules

Predicate

```
typeOfType : scope * Type → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

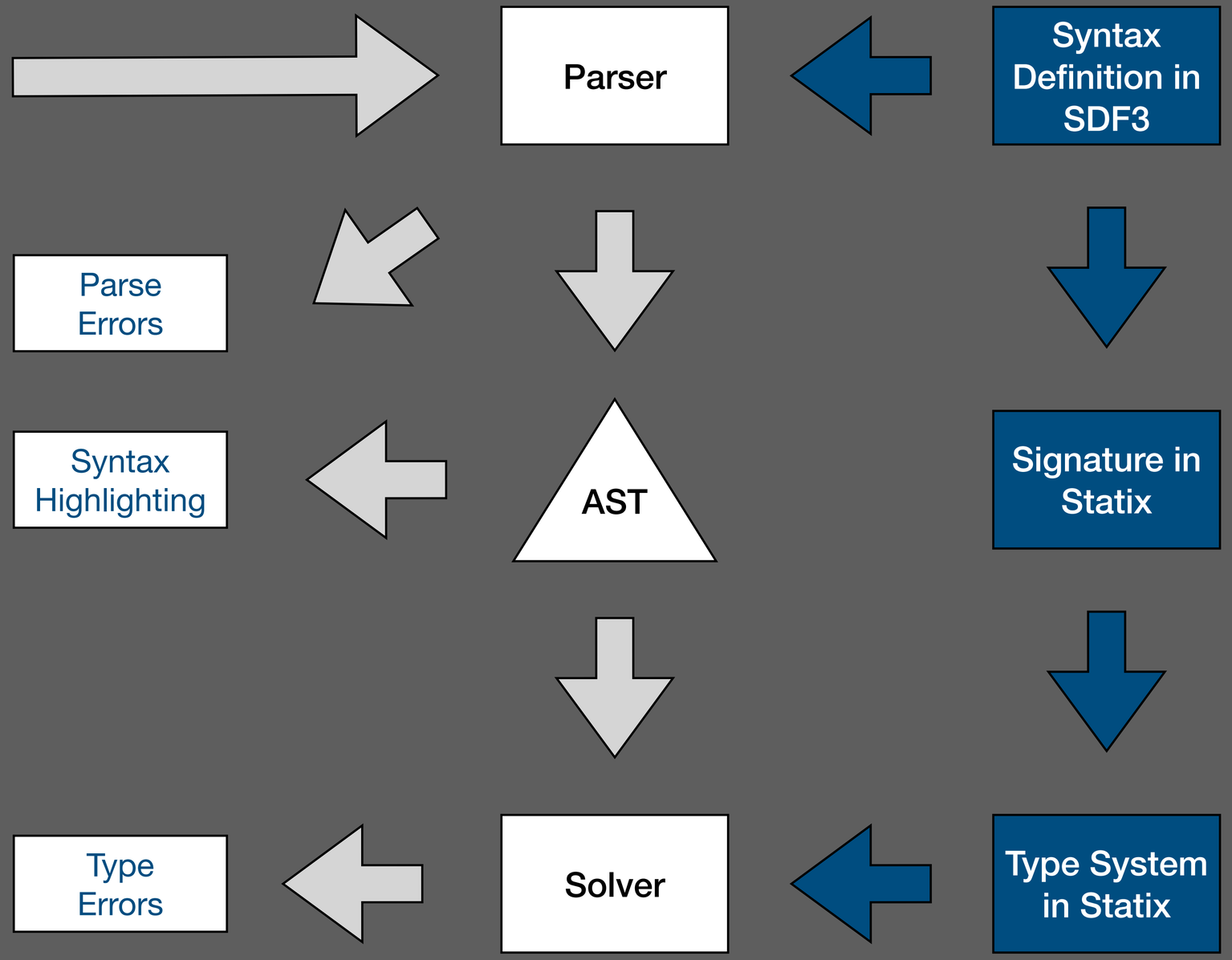
Premises

For all s, e1, e2

If the premises are true, the head is true

From Declarative Definition to Type Checker

```
1> 1 + 2 * 3
2
3> true && false
4
5> 1 ^ 2
6
7> true + 4
8
9> 1 && (true || false)
10
11> if 1 == 1 then
12    true
13  else
14    1 == 3
15
16> if 1 == 1 then
17    true
18  else
19    2
```



Programs with Names

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x)
         if x == 0 then true else odd(x - 1)
  }

  module Odd {
    import Even
    def odd = fun(x)
          if x == 0 then false else even(x - 1)
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x)
           Result{ input = x, output = Odd.odd x }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Name resolution complicates type checkers, compilers

Ad hoc non-declarative treatment

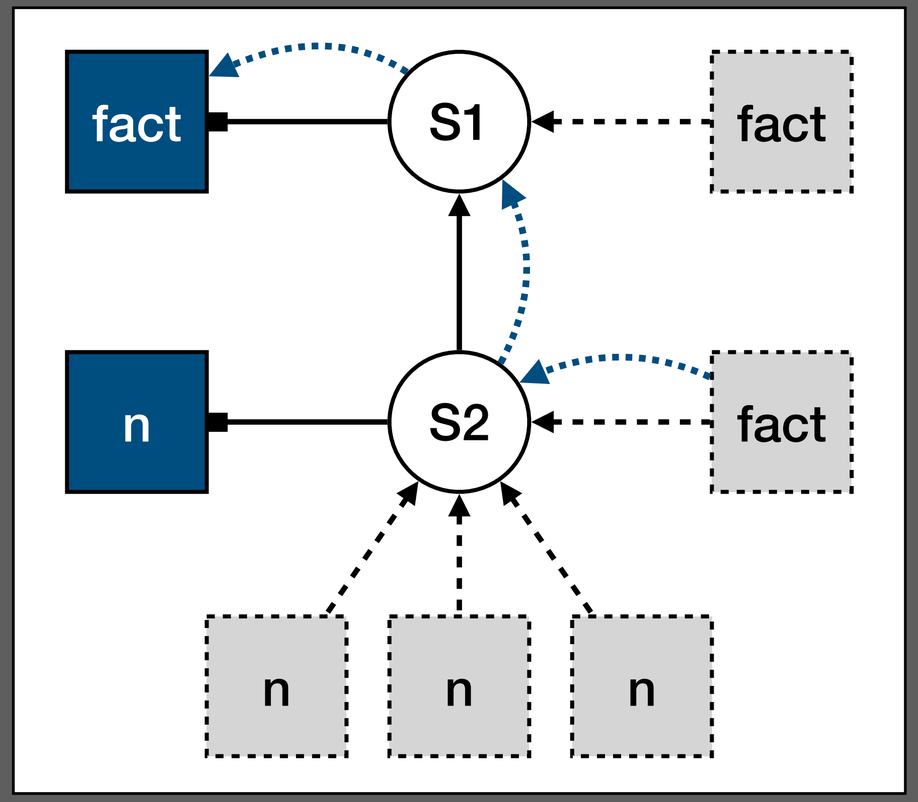
A systematic, uniform approach to name resolution?

Name Resolution with Scope Graphs

Program

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

Scope Graph



Name Resolution

Declaring and Resolving Names

Declarations and References

```
signature
  constructors
    Var   : ID → Exp
    Def   : Bind → Decl
    Bind  : ID * Exp → Bind
```

```
rules

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
```

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  typeOfExp(s, Var(x)) = typeOfVar(s, x).

  declOk(s, Def(bind)) :-
    bindOk(s, s, bind).

  bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
    typeOfExp(s_ctx, e) == T,
    declareVar(s_bnd, x, T).
```

Representing Name Binding with Scope Graphs

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE
```

Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE
```

Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

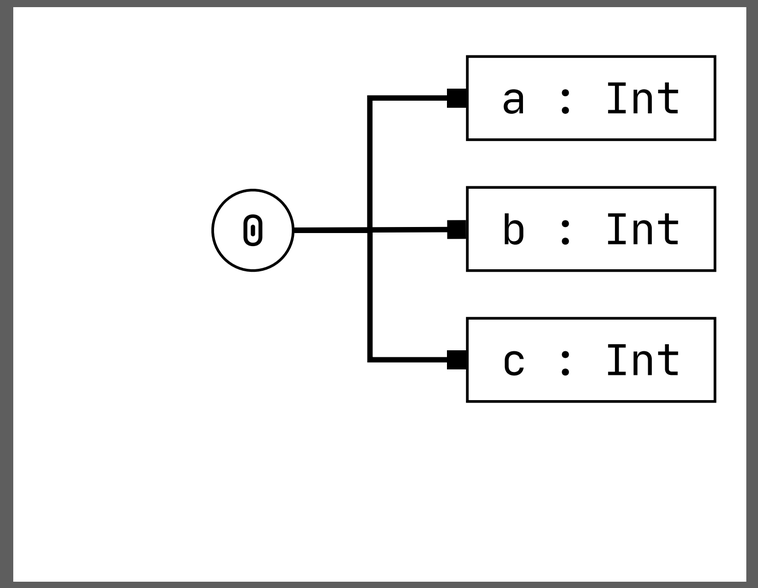
declaration relation

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T.
```

variable x is declared in scope s with type T



Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

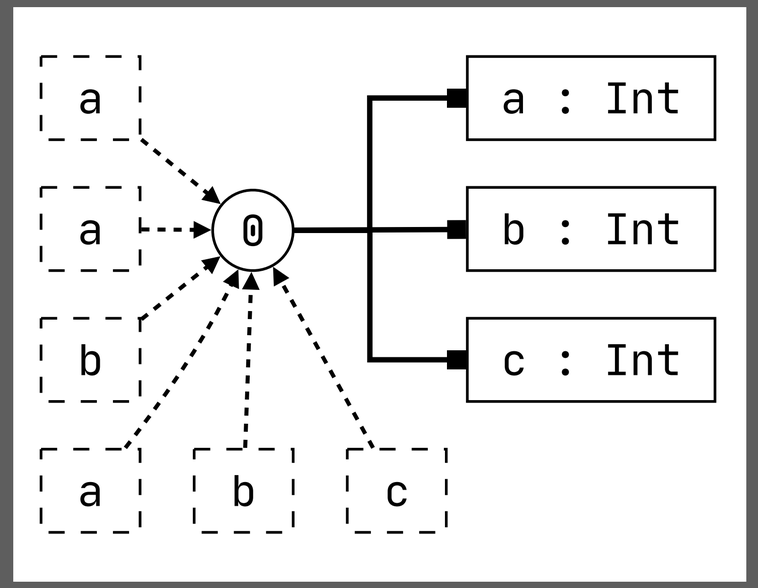
```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T.

  typeOfVar(s, x) = T :- {x'}
    typeOfDecl of Var{x} in s ⟼ [(_, (Var{x'}, T))].
```

variable x is declared in scope s with type T

variable x in scope s resolves to declaration x' with type T



How about shadowing?

Lexical Scope

New Scope and Scope Edge Constraints

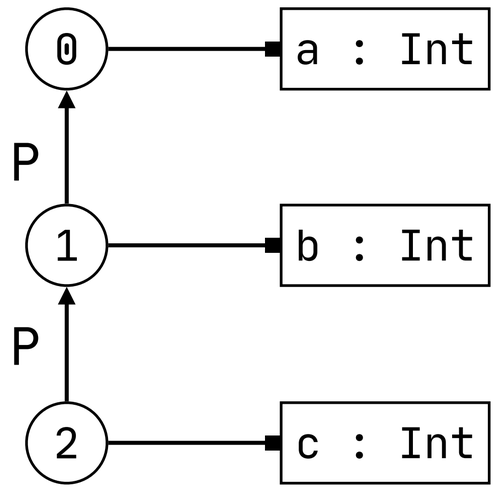
```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let,
     s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

scope edge



Path Wellformedness: Reachability

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let,
     s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

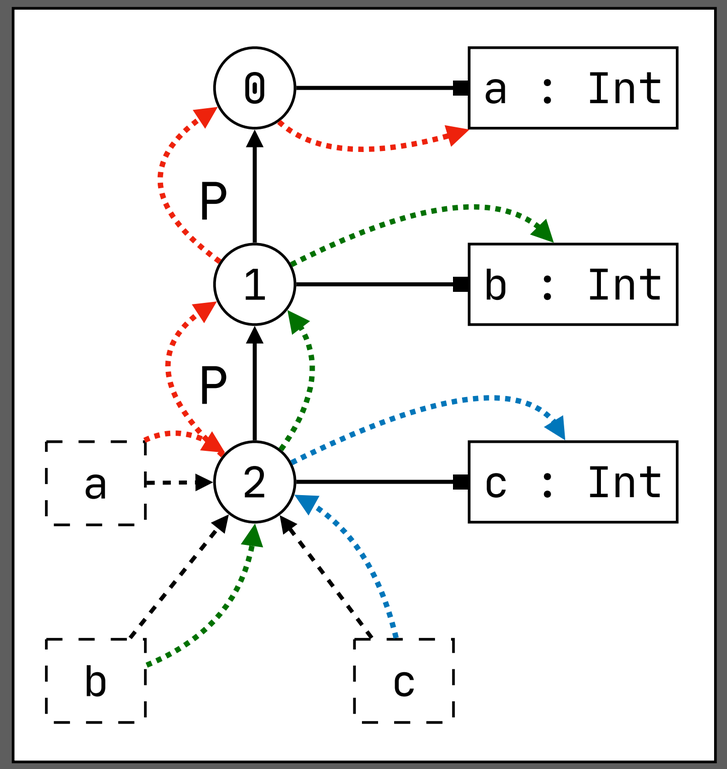
new scope

scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

path *P** allows resolution through zero or more *P* edges

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```



Path Specificity: Visibility (Shadowing)

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let,
     s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

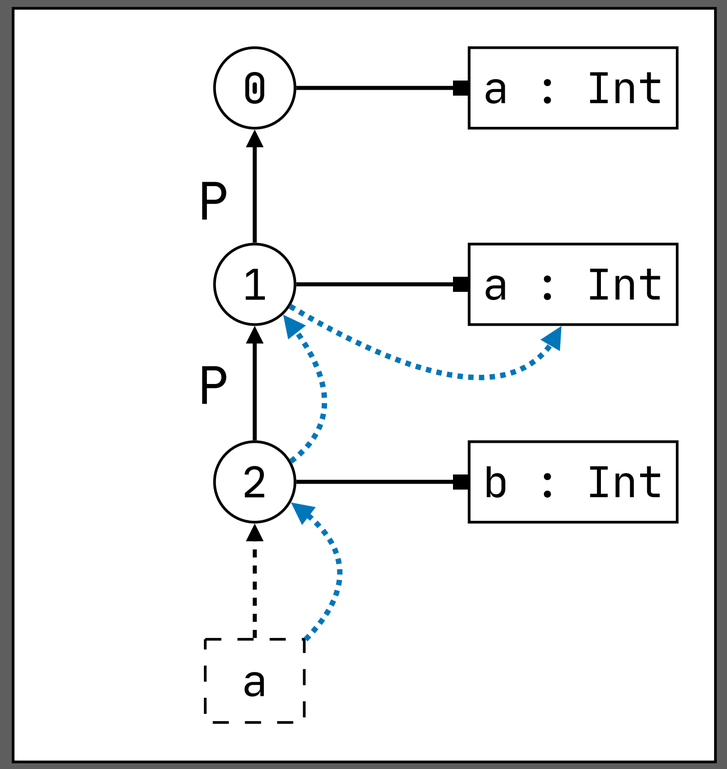
scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P* min $ < P
```

path *P** allows resolution through zero or more *P* edges

prefer local scope ($) over parent scope (*P*)

```
let a = 1 in
let a = 2 in
let b = 3 in
  a
```



How about non-lexical bindings?

Non-Lexical Scope (Modules)

Modules: Scopes as Types

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

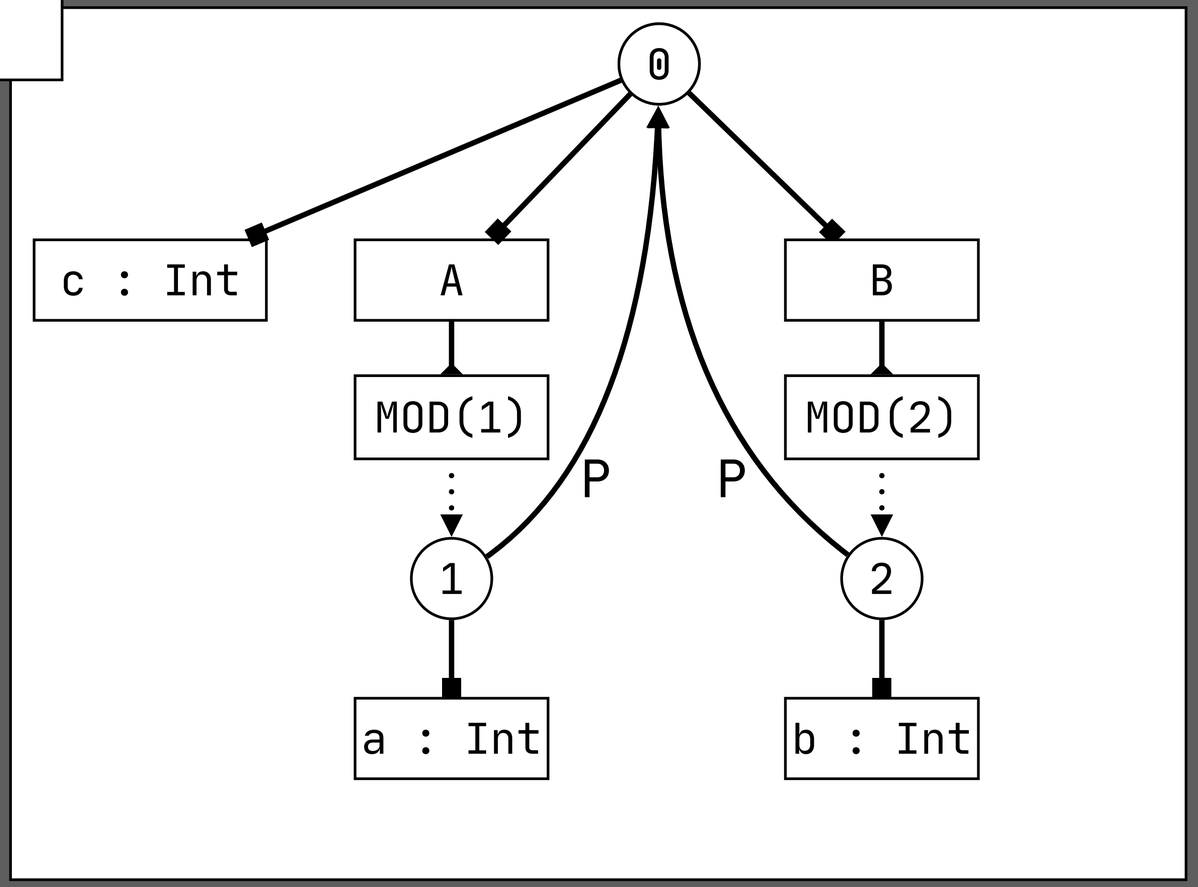
scope as type

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).
```

lexical scope

scope as type




```
signature
  namespaces
    Mod  : string
```

Resolving Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```

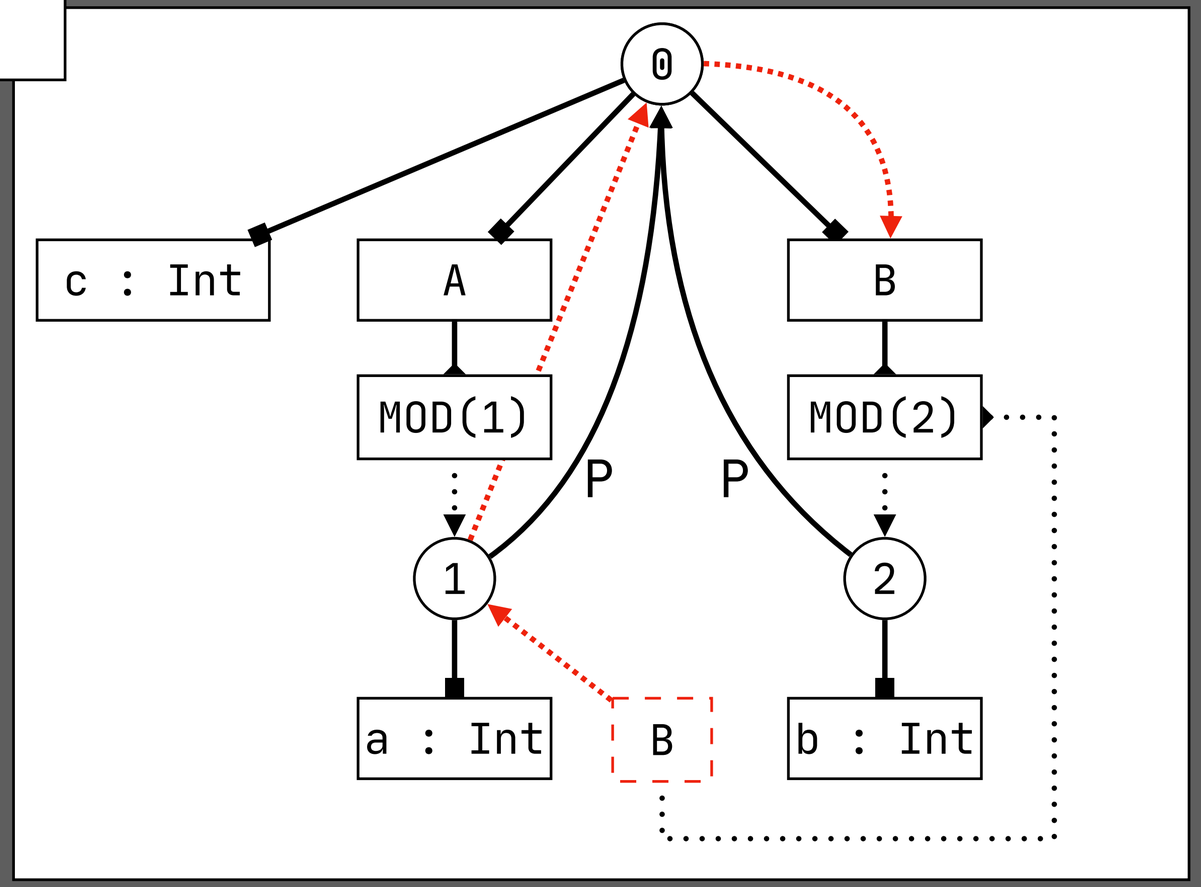
```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

resolve import

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P*
      min $ < I, $ < P, I < P
```



Import Edge

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

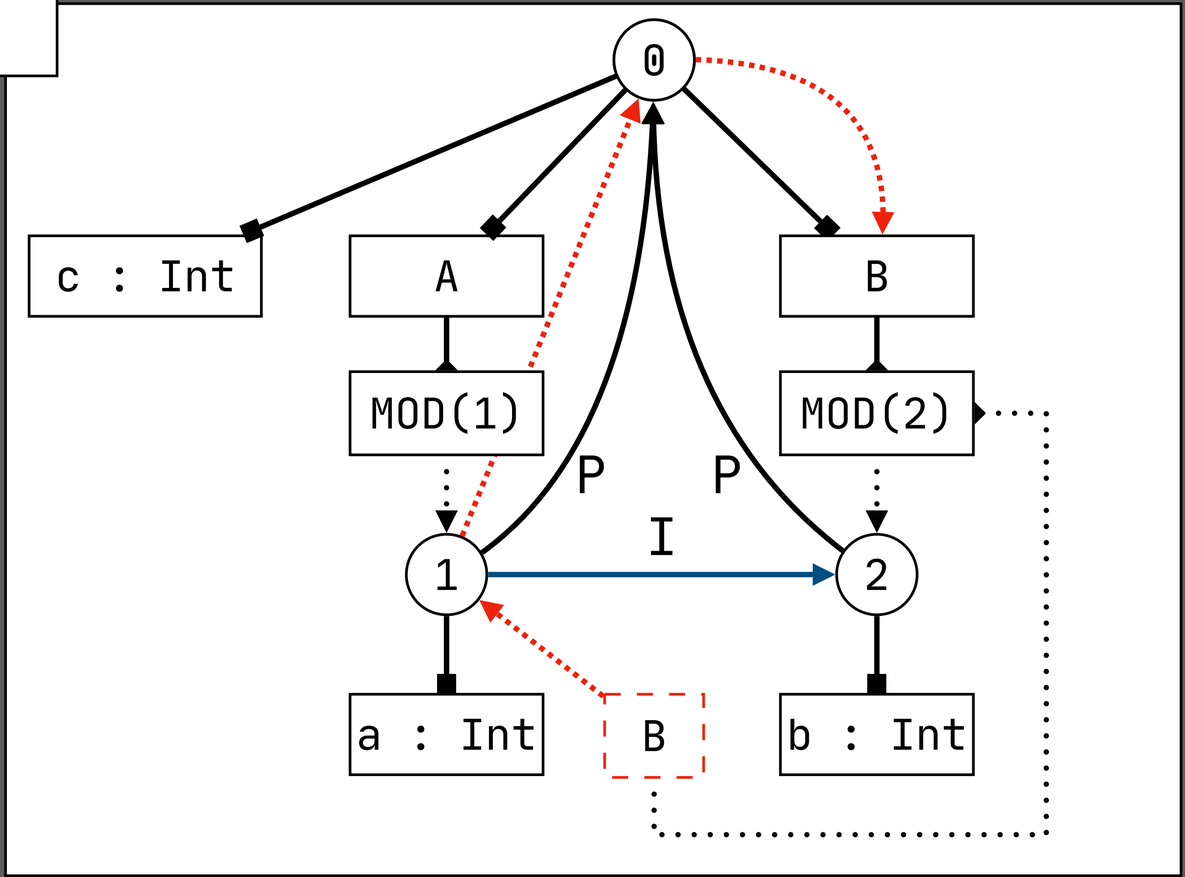
scope as type

resolve import

import edge

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P*
      min $ < I, $ < P, I < P
```

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Resolving through Import Edge

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

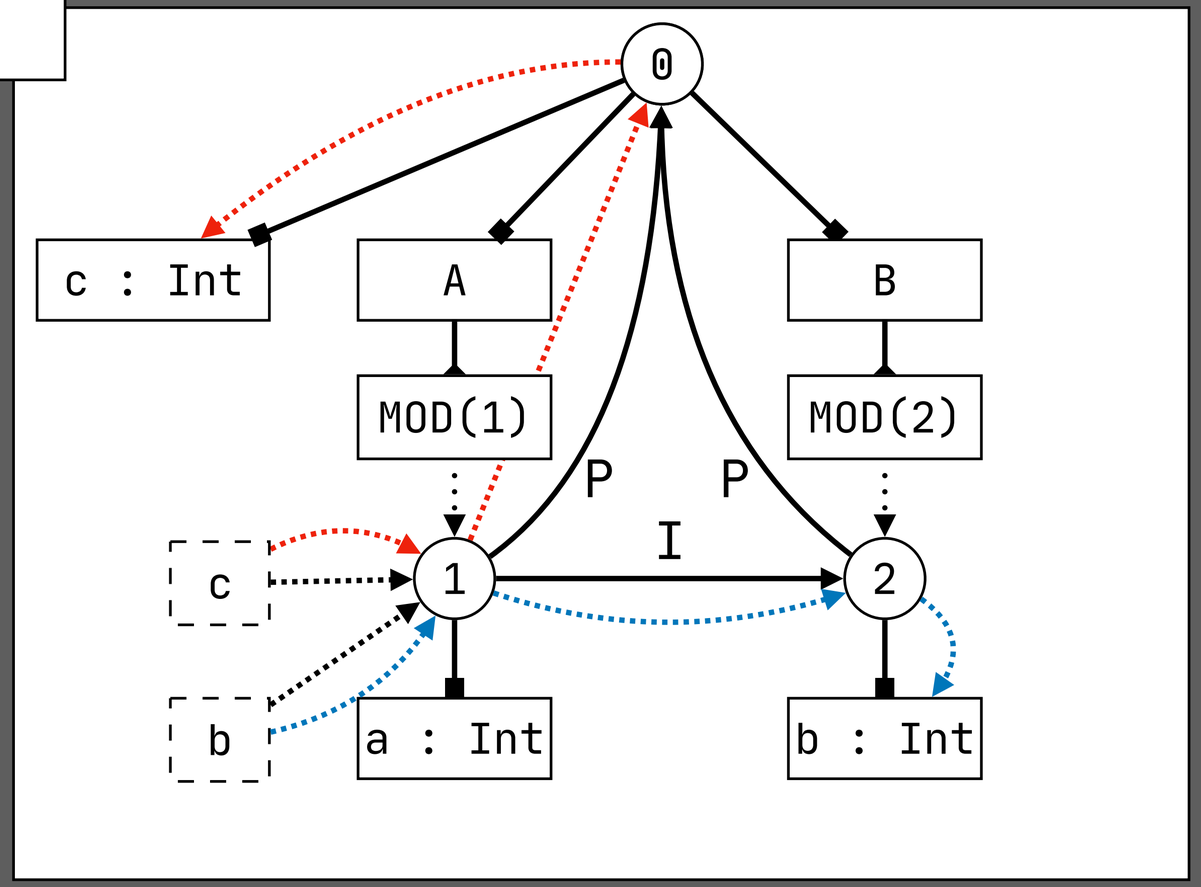
scope as type

resolve import

import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges



Import vs Parent

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
def b = 0
module A {
  import B
  def a = b
}
module B {
  def b = 2
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

resolve import

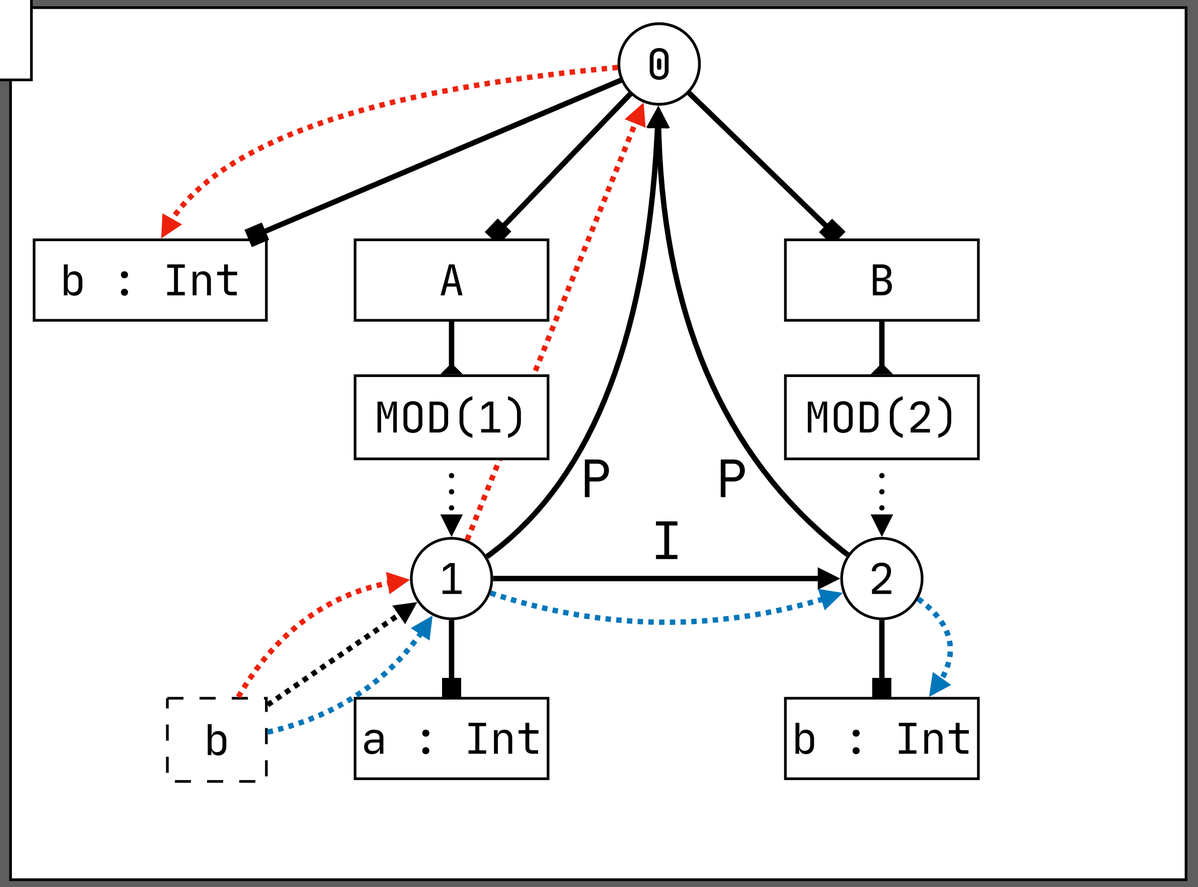
import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

prefer import

prefer blue path over red path



Mutual Imports

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

scope as type

resolve import

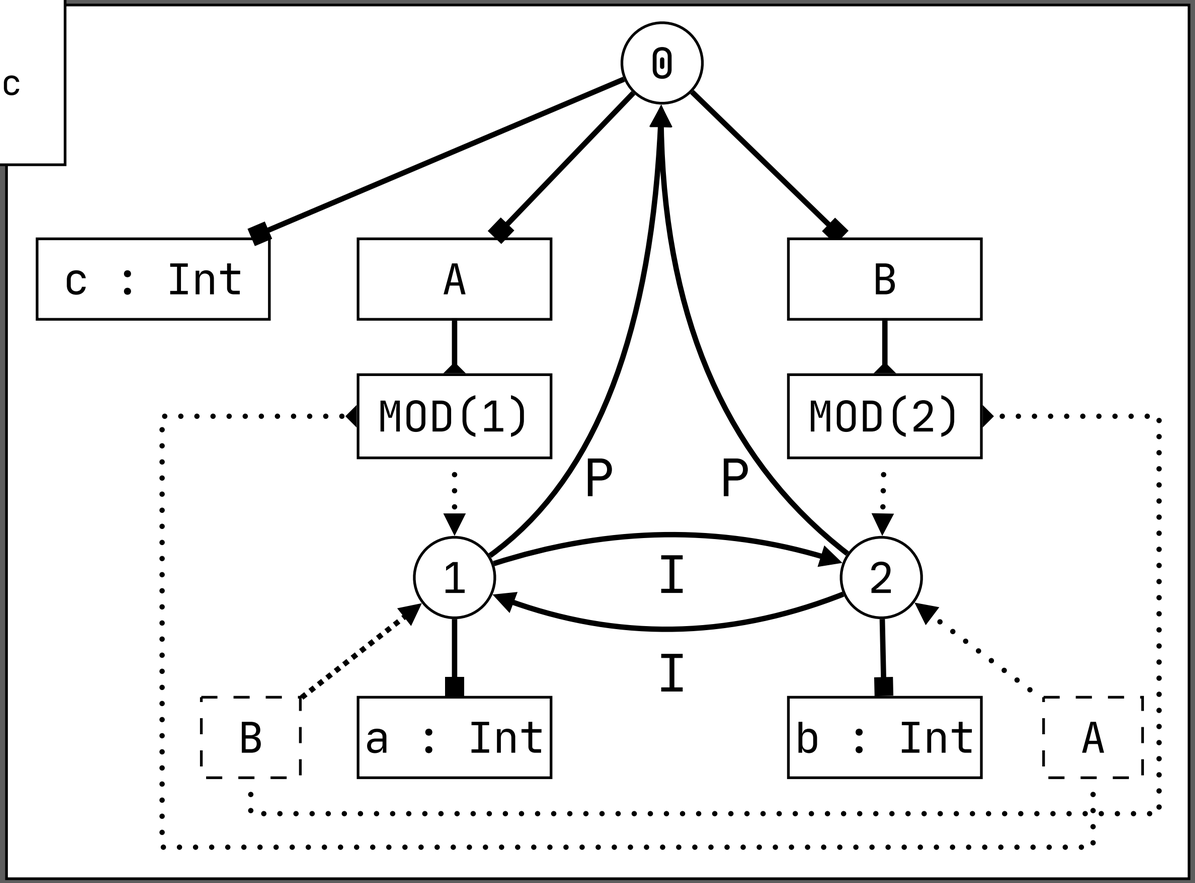
import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

import after parent

prefer import

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```



Mutual Imports

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

scope as type

resolve import

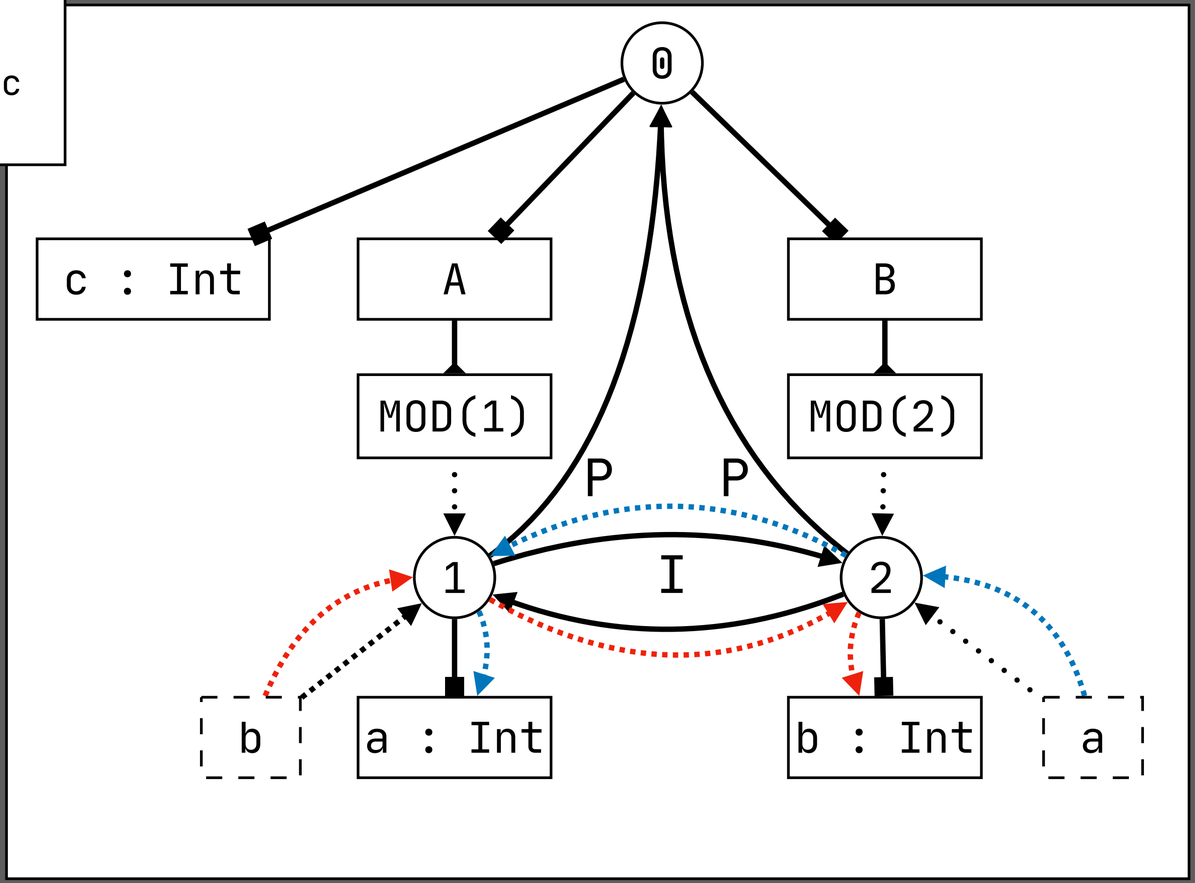
import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

prefer import

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```



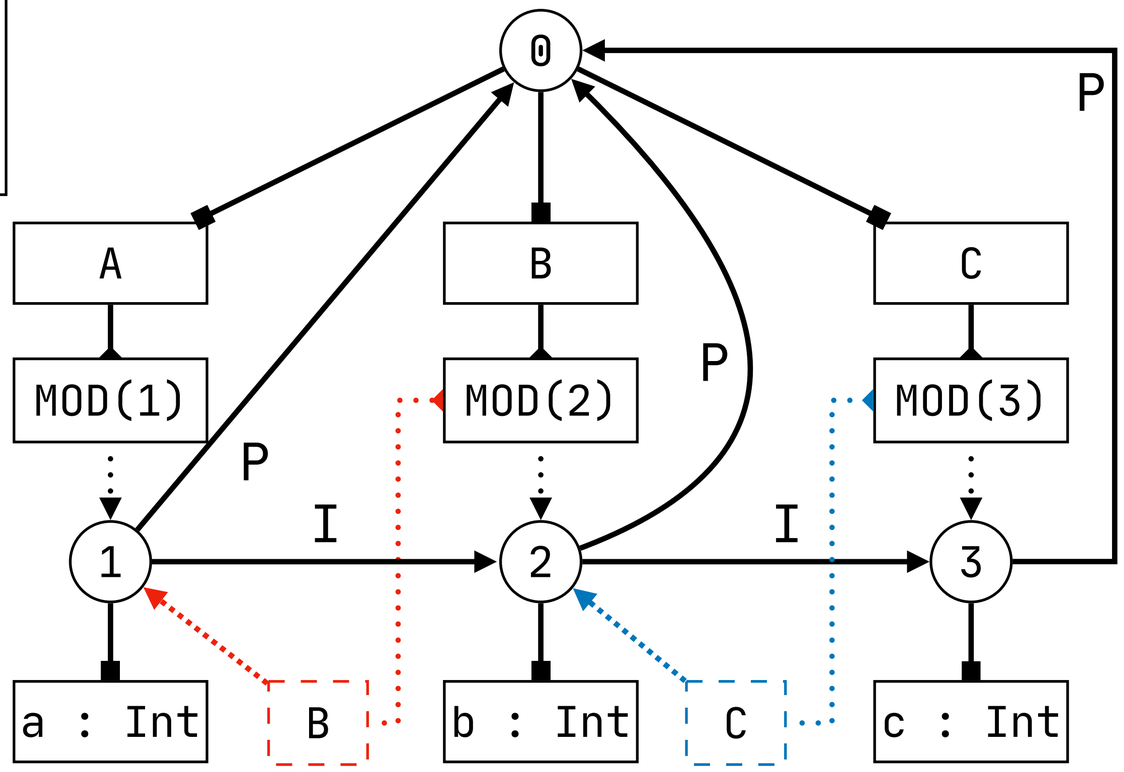
Transitive Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

```
module A {
  import B
  def a = b + c
}
module B {
  import C
  def b = c + 2
}
module C {
  def c = 1
}
```



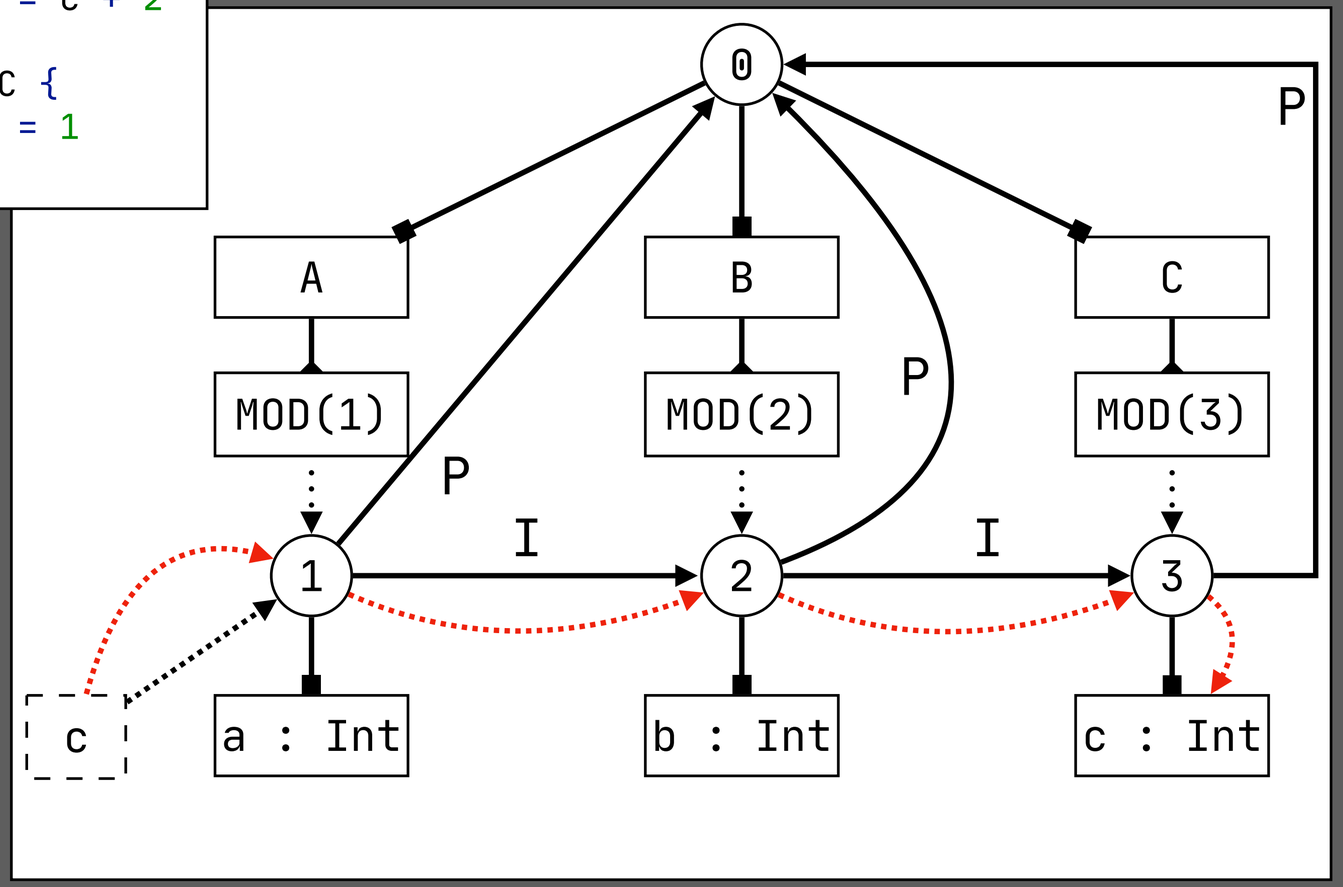
Transitive Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

```
module A {
  import B
  def a = b + c
}
module B {
  import C
  def b = c + 2
}
module C {
  def c = 1
}
```



Statix Interpretations

Statix Interpretations (In Progress)

Declarative Semantics [OOPSLA'18]

- `G ⊨ programOk(s, p)`
- Does program `p` satisfy the `programOk` predicate in scope `s`, given scope graph `G`?

Type Checking

- Given a program term `p`, what is valid scope graph `G`?
- Operational semantics is safe wrt declarative semantics [OOPSLA'20]
- Type check programs concurrently and/or incrementally

Code Completion [ECOOP'19]

- Given a hole (placeholder) in an incomplete program, what are valid completions?

Renaming

- Given a name x in a program, can it be renamed to y, without being captured?

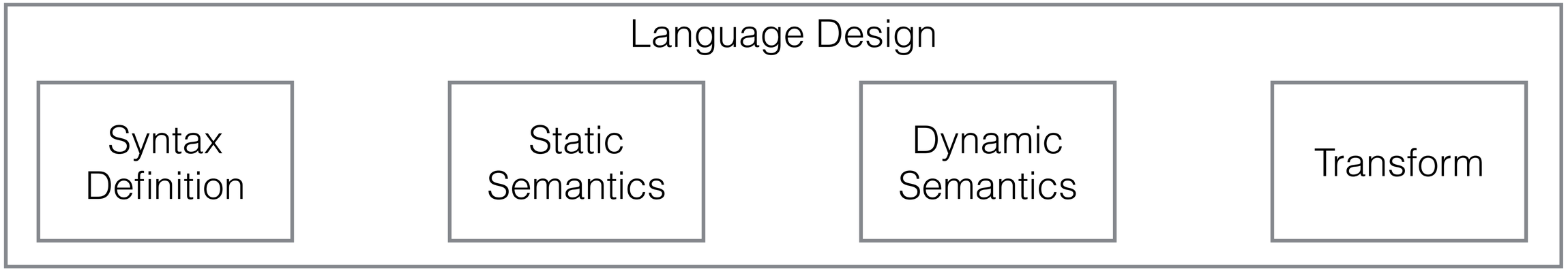
Quick Fixes

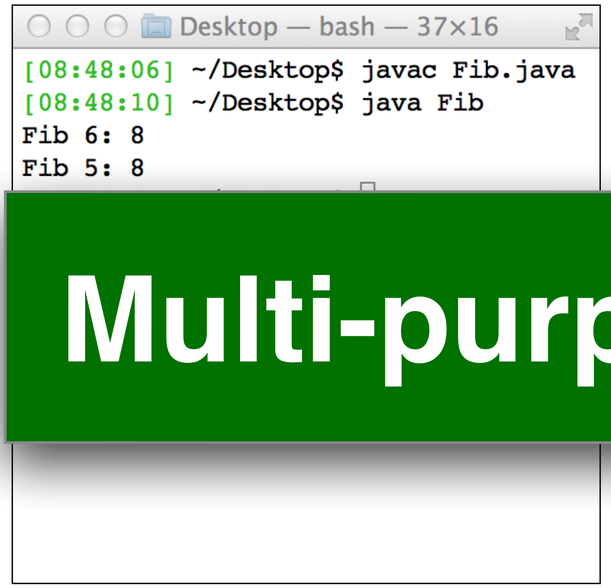
- Given a name/type error in a program, what is repair that would solve the error?

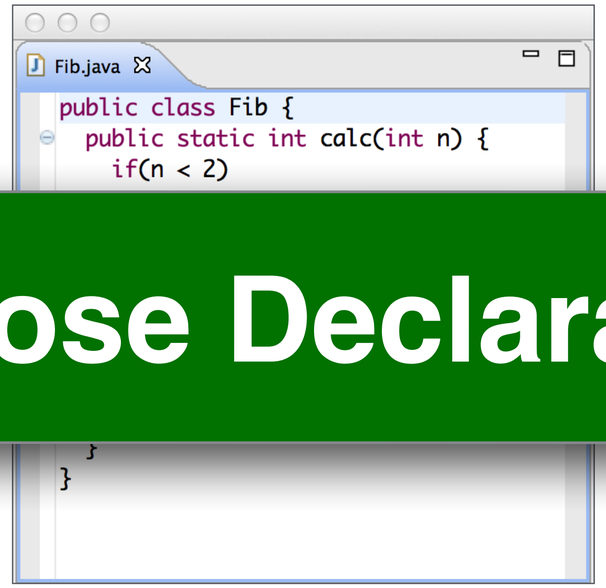
Random Term Generation

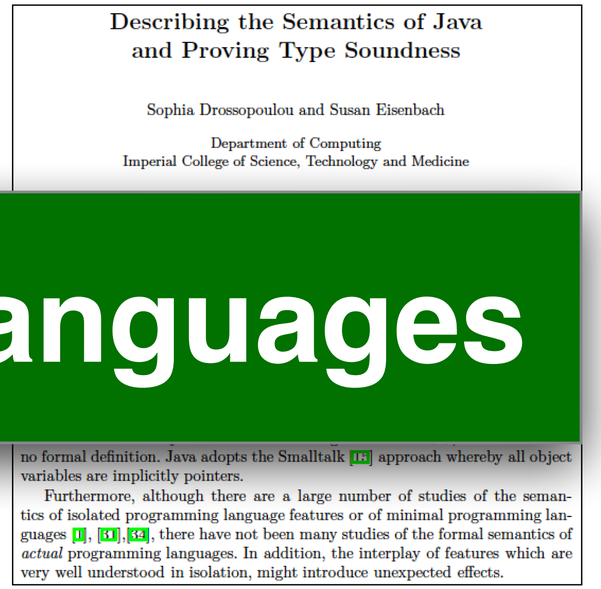
- Given a placeholder (and type), randomly generate a program that is syntactically, binding, and type correct

Conclusion









Multi-purpose Declarative Meta-Languages

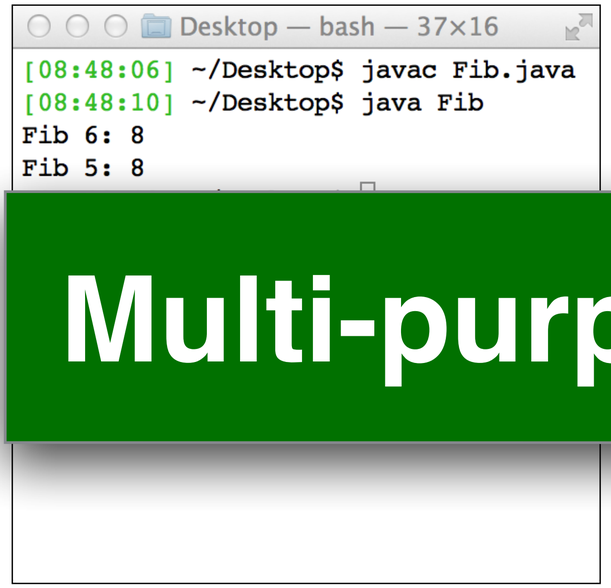
Language Design

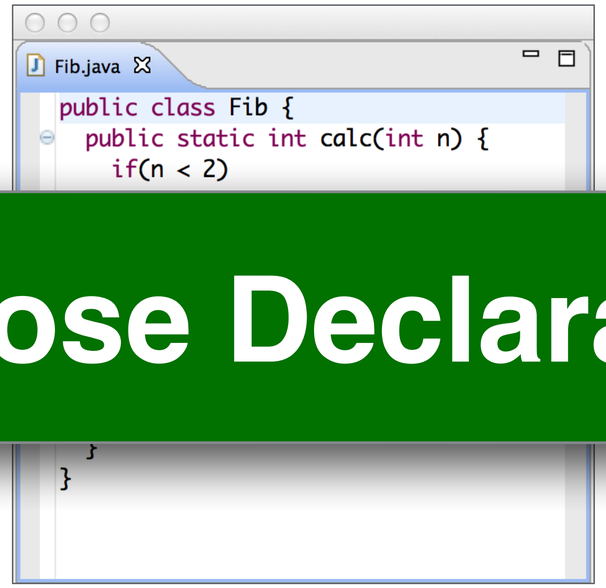
SDF3

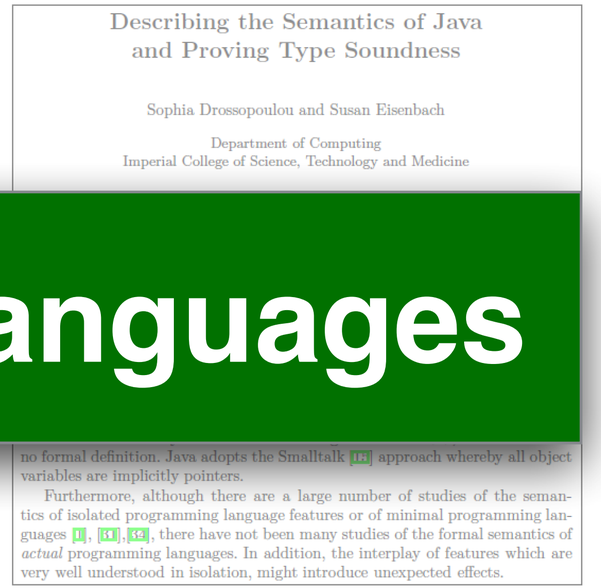
Statix

Dynamic Semantics

Transform







Multi-purpose Declarative Meta-Languages

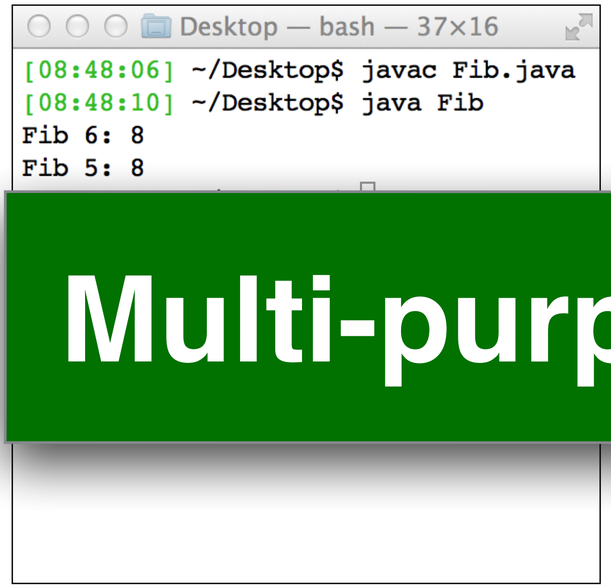
Language Design

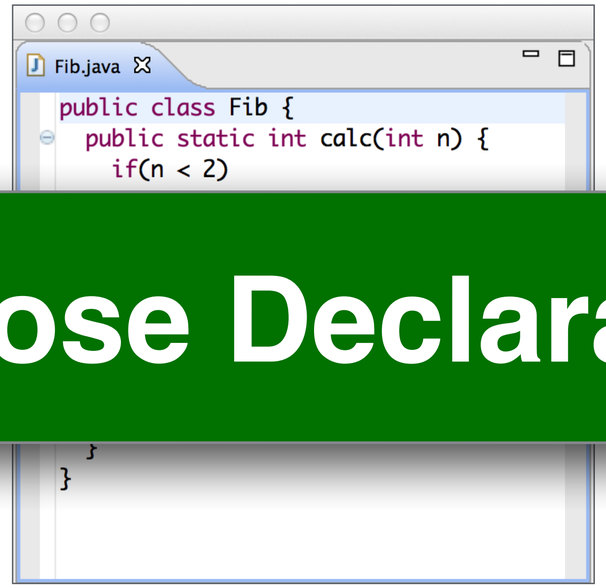
SDF3

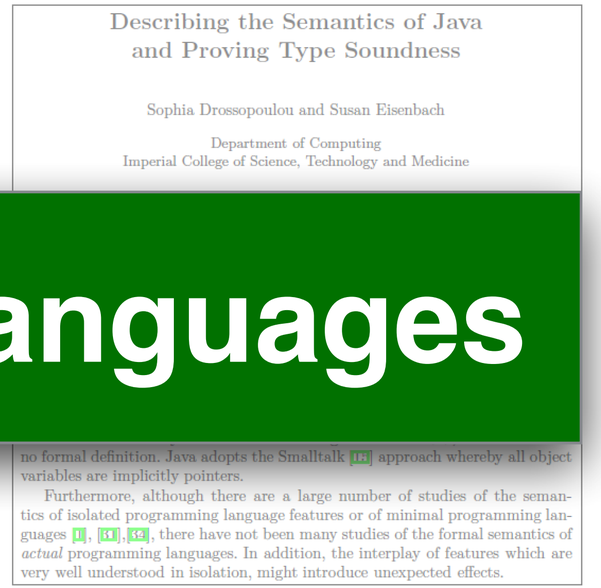
Statix

DynSem
Dynamix

Stratego







Multi-purpose Declarative Meta-Languages

More Information

Eelco Visser About Research Teaching News Blog Contact

Publications by Year

See also: Projects | Talks | Posters | Archives | BibTex | Researchr | DBLP | Google Scholar | ACM DL | Researchgate

2020

Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System
Jeff Smits, Gabriël D. P. Konat, Eelco Visser.
Programming 4(3) 2020 [pdf, doi, bib, researchr]

FlowSpec: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis
Jeff Smits, Guido Wachsmuth, Eelco Visser.
JCL (JVLC) 57 2020 [pdf, doi, bib, researchr]

Multi-Purpose Syntax Definition with SDF3
Luís Eduardo Amorim de Souza, Eelco Visser.
Software Engineering and Formal Methods - 18th International Conference, SEFM 2020 2020 [pdf, bib, researchr, abstract]

Intrinsically-typed definitional interpreters for linear, session-typed languages
Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, Eelco Visser.
CPP 2020 [pdf, doi, bib, researchr]

Safety and Completeness of Disambiguation corresponds to Termination and Confluence of Reordering
Luís Eduardo Amorim de Souza, Eelco Visser.
2020 [pdf, bib, researchr]

2019

Editorial Message
Eelco Visser.
PACMPL 3(OOPSLA) 2019 [pdf, bib, researchr]

Fast and Safe Linguistic Abstraction for the Masses
Eelco Visser.
A Research Agenda for Formal Methods in the Netherlands 2019 [bib, researchr]

From Whole Program Compilation to Incremental Compilation: A Critical Case
Jeff Smits, Gabriël Konat, Eelco Visser.
Second Workshop on Incremental Computing (IC 2019) 2019 [pdf, bib, researchr]

Scopes and Frames Improve Meta-Interpreter Specialization
Vlad A. Vergu, Andrew Tolmach, Eelco Visser.
ECOOP 2019 [pdf, doi, bib, researchr]

Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)
Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Eelco Visser.
ECOOP 2019 [pdf, doi, bib, researchr]

Towards language-parametric semantic editor services based on declarative type system specifications
Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Eelco Visser.
OOPSLA 2019 [pdf, doi, bib, researchr]

Precise, Efficient, and Expressive Incremental Build Scripts with PIE
Gabriël Konat, Roelof Sol, Sebastian Erdweg, Eelco Visser.
Second Workshop on Incremental Computing (IC 2019) 2019 [pdf, bib, researchr]

http://eelcovisser.org

Spoofax
latest

Search docs

Docs » The Spoofax Language Workbench Edit on GitHub

The Spoofax Language Workbench

Spoofax is a platform for developing textual (domain-specific) programming languages. The platform provides the following ingredients:

- Meta-languages for high-level declarative language definition
- An interactive environment for developing languages using these meta-languages
- Code generators that produces parsers, type checkers, compilers, interpreters, and other tools from language definitions
- Generation of full-featured Eclipse editor plugins from language definitions
- Generation of full-featured IntelliJ editor plugins from language definitions (experimental)
- An API for programmatically combining the components of a language implementation

With Spoofax you can focus on the essence of language definition and ignore irrelevant implementation details.

Developing Software Languages

Spoofax supports the development of *textual* languages, but does not otherwise restrict what kind of language you develop. Spoofax has been used to develop the following kinds of languages:

Programming languages
Languages for programming computers. Implement an existing programming language to create an IDE and other tools for it, or design a new programming language.

Domain-specific languages
Languages that capture the understanding of a domain with linguistic abstractions. Design a DSL for your domain with a compiler that generates code that would be tedious and error prone to produce manually.

Scripting languages
Languages with a special run-time environment and interpreter

Work-flow languages
Languages for scheduling actions such as building the components of a software system

Configuration languages
Languages for configuring software and other systems

Data description languages
Languages for formatting data

Data modeling languages
Languages for describing data schemas

http://metaborg.org