A Framework for Transformation of Java Programs

(Stratego/XT for Java)

Martin Bravenboer

Karl Trygve Kalleberg

Rob Vermaas

Eelco Visser

Department of Information & Computing Sciences
Utrecht University, The Netherlands

www.stratego-language.org

IFIP WG 2.11, Dagstuhl, January 27, 2006

Stratego/XT Project Mission Statement

Create a high-level, language parametric, rule-based program transformation system, which supports a wide range of transformations, admitting efficient implementations that scale to large programs.

Outline

Introduction

- Program transformation
- Language independent transformation infrastructure
- Infrastructure for Java transformation

Techniques

- Rewrite rules
- Concrete syntax
- Rewriting strategies
- Annotations
- Dynamic rules
- Dryad API

Examples

- Conditional lifting
- Expression blocks
- Type annotation
- Interface conformance
- JavaJava assimilation
- (Constant propagation)

Transformation Example: Conditional Lifting

```
public String getArg(String[] args)
{
  return args.length > 0 ? args[0] : "";
}
```

lift conditionals to statement level

```
public String getArg(String[] args)
{
  String expr_0;
  if(args.length > 0)
    expr_0 = args[0];
  else
    expr_0 = "";
  return expr_0;
}
```

Transformation Example: Interface Conformance

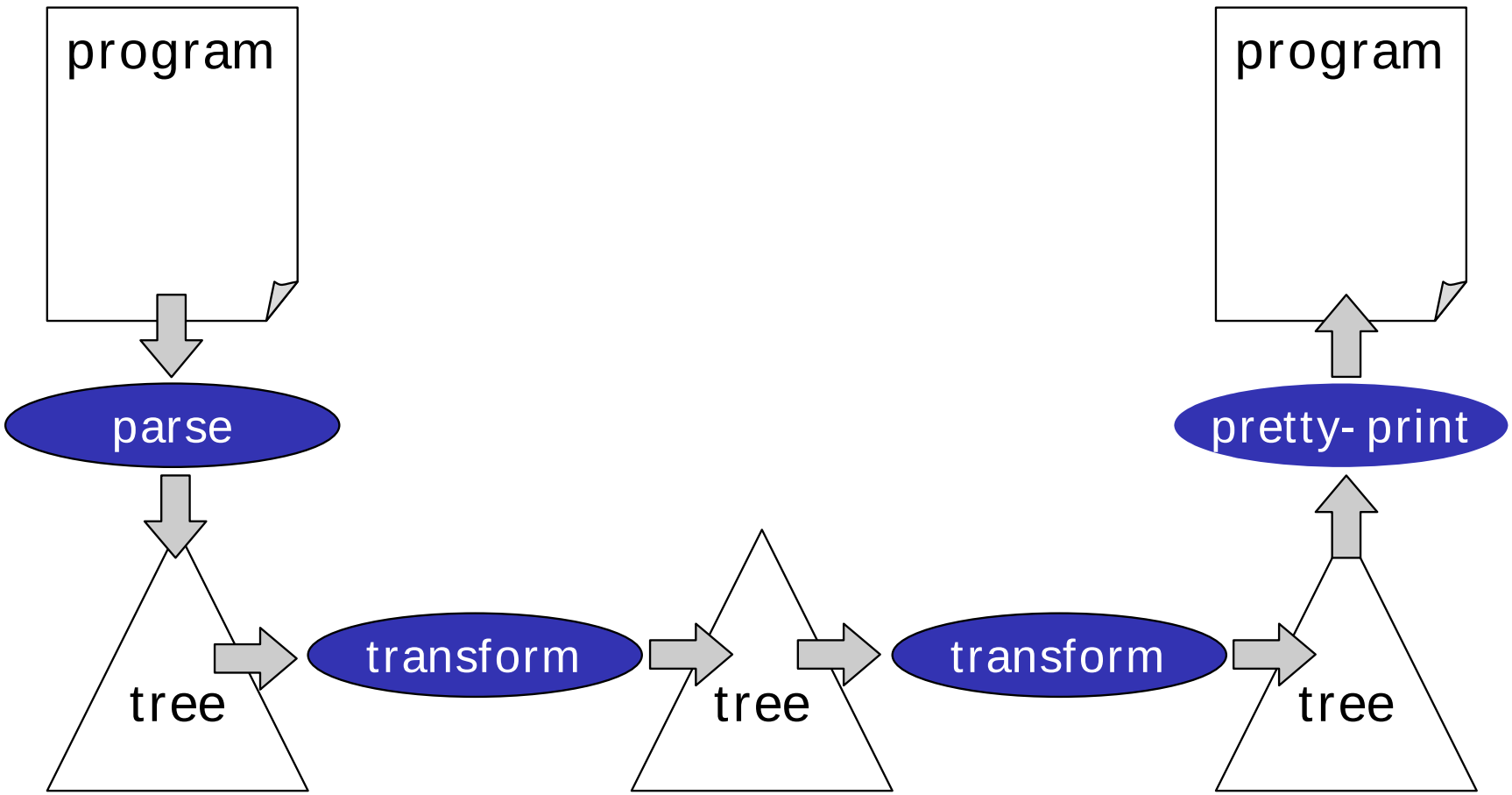
```
@Implements ({"java.awt.event.MouseListener"})
public class SomeListener {
  public void mousePressed (MouseEvent event) { ... }
  public void mouseDragged (MouseEvent event) { ... }
}
```

add default (empty) implementations for missing methods

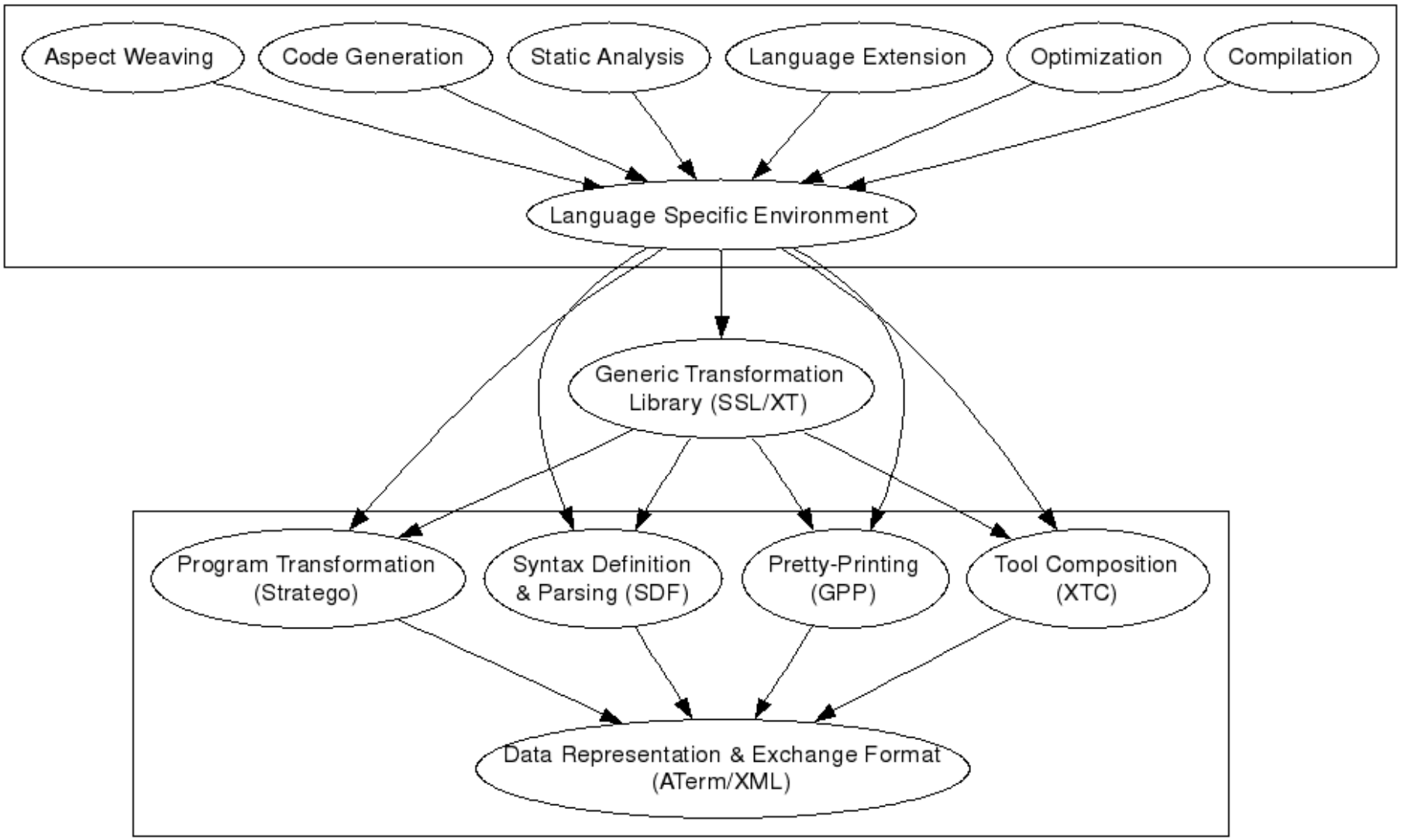
```
@Implements ({"java.awt.event.MouseListener"})
public class SomeListener implements MouseListener {
  public void mousePressed  (MouseEvent event) { ... }
  public void mouseDragged  (MouseEvent event) { ... }
  public void mouseClicked  (MouseEvent e) {}
  public void mouseReleased (MouseEvent e) {}
  public void mouseEntered  (MouseEvent e) {}
  public void mouseExited   (MouseEvent e) {}
}
```

(Example due to Yannis Smaragdakis)

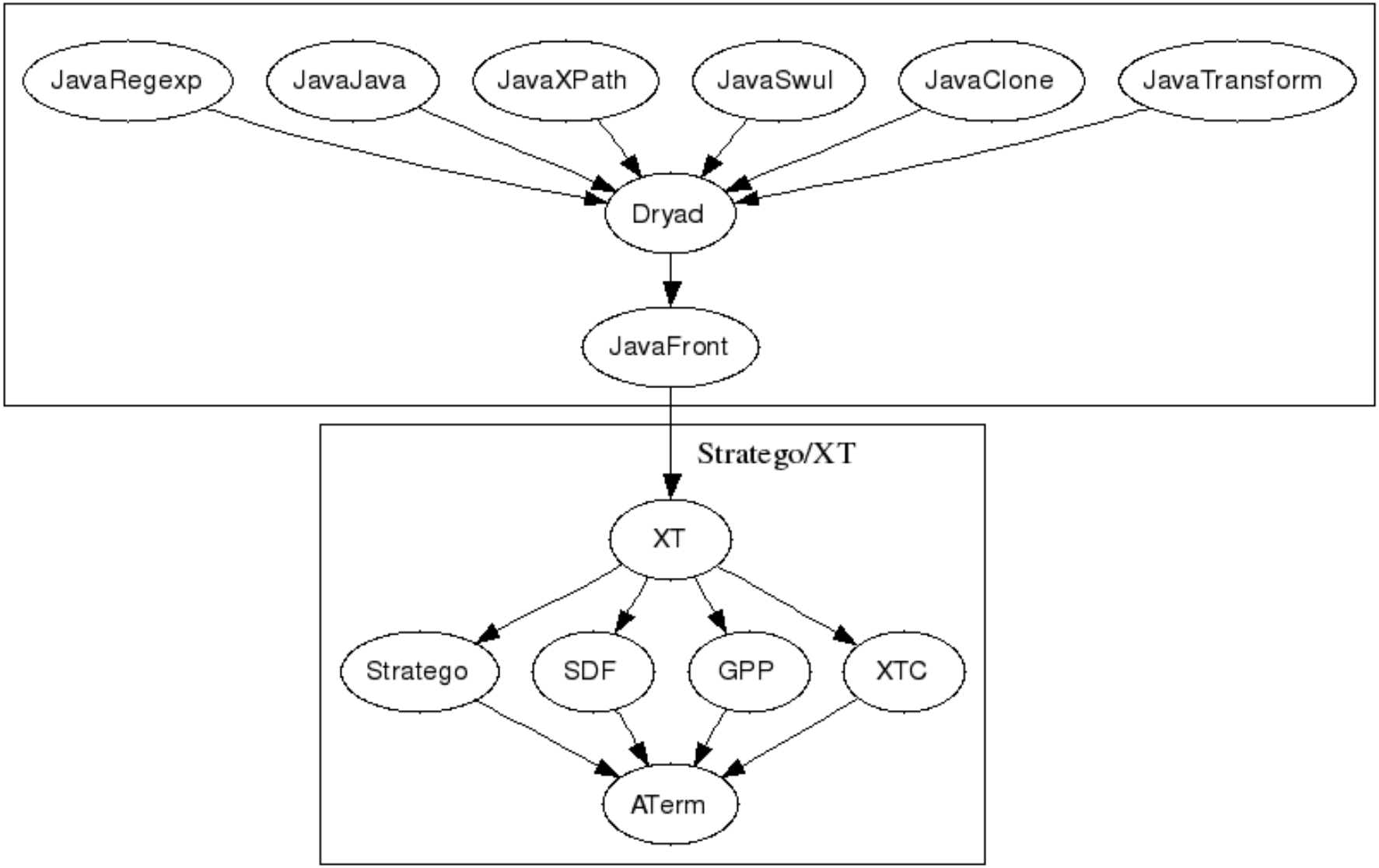
Transformation Pipelines



From Generic to Specific Components



Transformation Infrastructure for Java



JavaFront

Syntax Definition: SDF grammar for Java 5

(i.e. generics, enums, annotations, . . .)

- Modular, structure of Java Specification, 3^{rd} Edition
- Declarative disambiguation
  (i.e single expression non-terminal)
- Integrated lexical and context-free syntax
  Important for language extension (AspectJ)

Pretty-Printer

- Modular, rewrite rules, extensible
- Preserves priorities (generated)
- Heavy testing: roundtrip

Dryad, The Tree Nymph

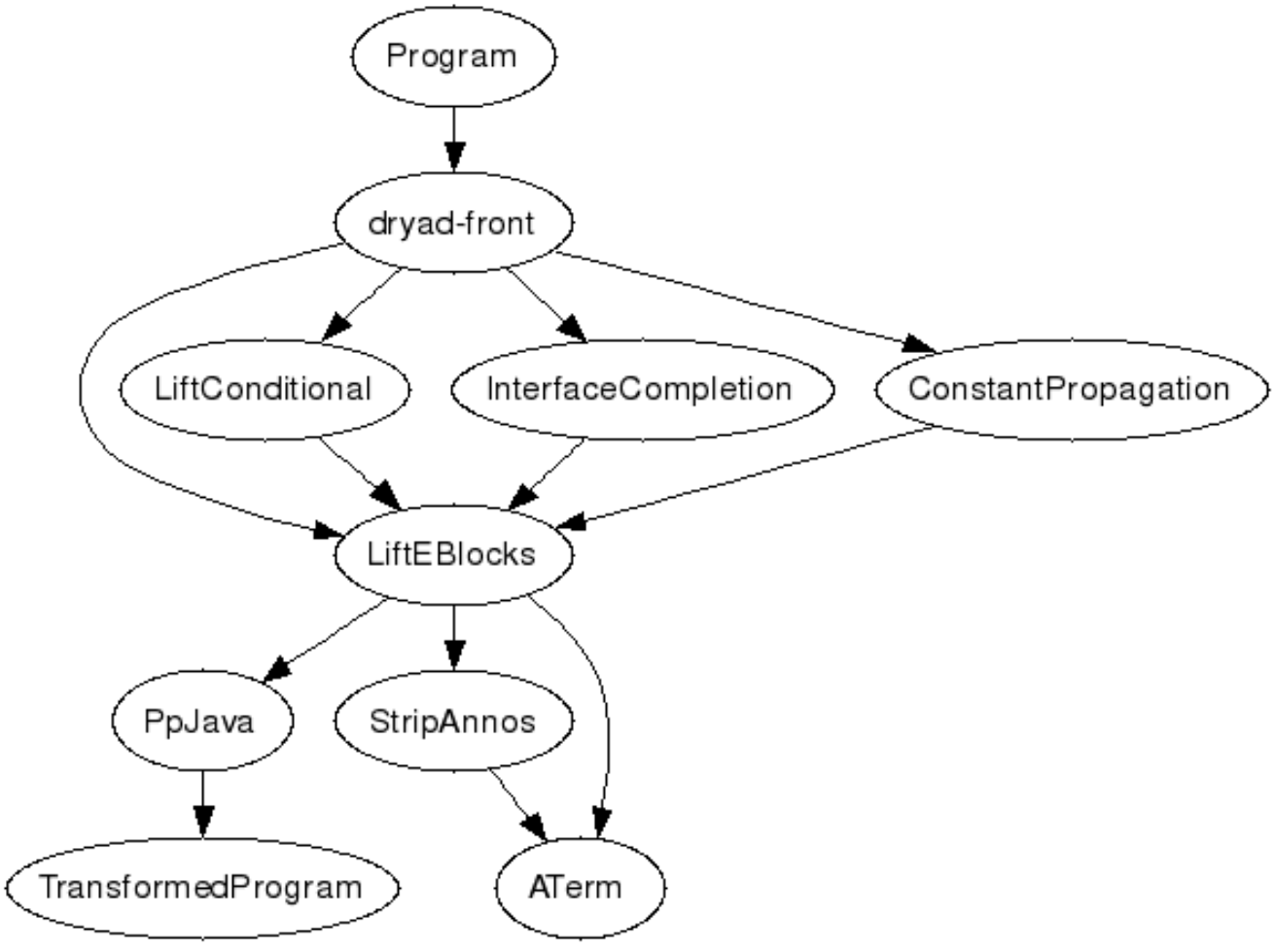
Beyond support for Java syntax

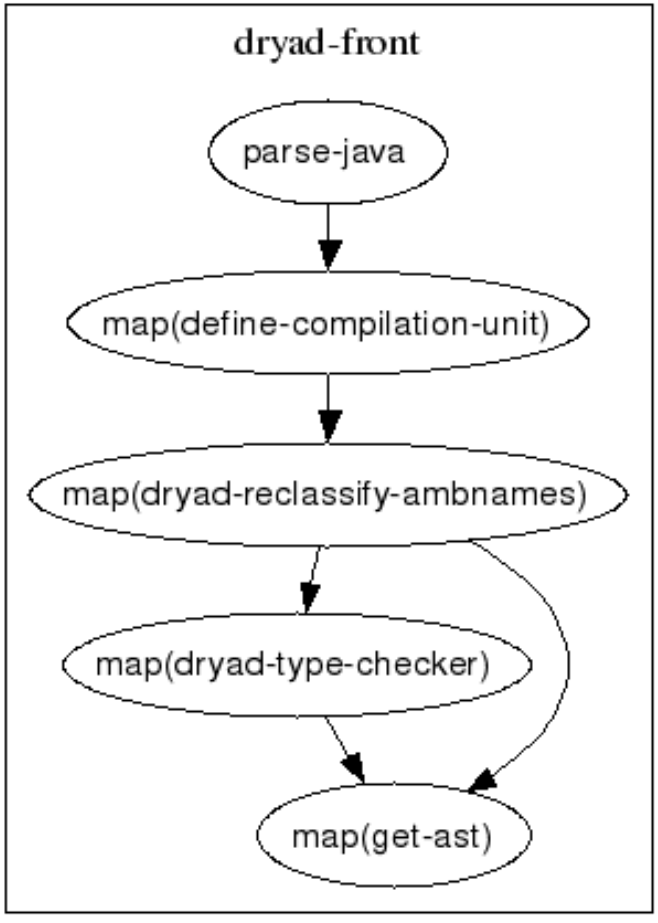
Java is quite ambiguous: semantic analysis is required for any non-trivial Java transformation system.

Features

- Java Bytecode $\leftrightarrow$ ATerm bridge
- Dryad Library
  - Model for Java source and bytecode
  - Implementation of JLS definitions
- Reclassification (disambiguation) and qualification
- Type checker (types, conversions, method resolution)

Demo Application: Java-Transform





Transformation Example: Conditional Lifting

```
public String getArg(String[] args)
{
  return args.length > 0 ? args[0] : "";
}
```

lift conditionals to statement level

```
public String getArg(String[] args)
{
  String expr_0;
  if(args.length > 0)
    expr_0 = args[0];
  else
    expr_0 = "";
  return expr_0;
}
```

Term Rewrite Rules

```
LiftConditionalFromReturn :
  Return(Some(Cond(e1, e2, e3))) ->
  If(e1, Return(Some(e2)), Return(Some(e3)))
```

```
return args.length > 0 ?  args[0] :  "";
```

```
Return(Some(
  Cond(Gt(Field(ExprName(Id("args")), Id("length")), Lit(Deci("0")))
      , ArrayAccess(ExprName(Id("args")), Lit(Deci("0")))
      , Lit(String([])))))
```

```
if(args.length > 0) return args[0]; else return "";
```

```
If(Gt(Field(ExprName(Id("args")), Id("length")), Lit(Deci("0")))
  , Return(Some(ArrayAccess(ExprName(Id("args")), Lit(Deci("0")))))
  , Return(Some(Lit(String([])))))
```

Concrete Syntax

```
LiftConditionalFromReturn :
  Return(Some(Cond(e1, e2, e3))) ->
  If(e1, Return(Some(e2)), Return(Some(e3)))
```

```
LiftConditionalFromReturn :
  |[ return e1 ? e2 : e3; ]| ->
  |[ if(e1) return e2; else return e3; ]|
```

Use concrete syntax of source language in specification of term patterns

- Embedding of source language in Stratego
- Quotation symbols |[and]| delimit embedded code fragments
- *Meta-variables* indicate holes in pattern

Rules for Lifting Conditionals

```
LiftConditionalFromReturn :
  |[ return e1 ? e2 : e3; ]| ->
  |[ if(e1) return e2; else return e3; ]|

LiftConditionalFromAssign :
  |[ e0 = e1 ? e2 : e3; ]| ->
  |[ if(e1) e0 = e2; else e0 = e3; ]|

LiftConditionalFromPlus :
  |[ e0 + (e1 ? e2 : e3) ]| ->
  |[ e1 ? (e0 + e2) : (e0 + e3) ]|

LiftConditionalFromPlus :
  |[ (e1 ? e2 : e3) + e0 ]| ->
  |[ e1 ? (e2 + e0) : (e3 + e0) ]|
```

Rewriting Strategies

Rewriting strategies control application of rules

- Select the rules to apply
- Select the strategy to apply them with

Example strategies

- `innermost(s)`: exhaustive application of rules
- `topdown(s)`: pre-order
- `bottomup(s)`: post-order
- `try(s)`: try to apply a rule

```
lift-conditionals =
  bottomup(try(
    LiftConditionalFromReturn
    <+ LiftConditionalFromAssign
    <+ ...
  ))
```

Programmable Rewriting Strategies

Strategies are programmable using basic strategy combinators

- `s1 <+ s2`: deterministic choice
- `s1 ; s2`: sequential composition
- `all(s)`: generic one-level traversal
- ...

```
try(s) = s <+ id

topdown(s) = s; all(topdown(s))

bottomup(s) = all(bottomup(s)); s

innermost(s) = bottomup(try(s; innermost(s)))
```

Conditional Lifting Revisited

Completing conditional lifting requires rules of the form

```
LiftConditionalFromExpression :
  |[ E[(e1 ? e2 : e3)] ]| -> |[ e1 ? E[e2] : E[e3] ]|
```

for each expression construct `E[.]`, and rules of the form

```
LiftConditionalFromStatement :
  |[ S[(e1 ? e2 : e3)] ]| -> |[ if(e1) S[e2] else S[e3] ]|
```

for each statement construct `S[.]`

Expression Blocks

Lifting expressions to statement level is a frequently occurring transformation, which can be factored out using *expression blocks*.

- syntactic extension of Java: `{| stat* | e |}`
- is expression
- statements in `stat*` can have side-effects

```
LiftConditionalToEBlock :
  |[ e1 ? e2 : e3 ]| ->
  |[ {| if(e1) x = e2; else x = e3; | x |} ]|
  where <newname> "expr" => x
```

```
z = z + (args.length > 0 ? args[0] : "");
```

e-block lifing

```
z = z + {| if(args.length > 0)
             expr_1 = args[0];
           else
             expr_1 = "";
         | expr_1 |};
```

```
if(args.length > 0)
  expr_1 = args[0];
else
  expr_1 = "";
z = z + expr_1;
```

Generating Variable Declarations

`LiftConditionalToEBlock` does not add a declaration for the new variable; that requires type information

```
LiftConditionalToEBlock :
  |[ e1 ? e2 : e3 ]| ->
  |[ {| ty x; if(e1) x = e2; else x = e3; | x |} ]|
  where type-attr => ty
      ; <newname> "expr" => x
```

```
z = z + (args.length > 0 ? args[0] : "");
```

e-block lifing

```
z = z + {| java.lang.String expr_1;
           if(args.length > 0)
             expr_1 = args[0];
           else
             expr_1 = "";
         | expr_1 |};
```

```
java.lang.String expr_1;
if(args.length > 0)
  expr_1 = args[0];
else
  expr_1 = "";
z = z + expr_1;
```

Term Annotations

Term annotations store additional (semantic) information

term representation of `args[0]`

```
ArrayAccess(
  ExprName(Id("args"))
  , Lit(Deci("0"))
)
```

with type annotations

```
ArrayAccess(
  ExprName(Id("args")){Type(ArrayType(t1))}
  , Lit(Deci("0")){Type(Int)}
){Type(t1)}

t1 = ClassType(t2, None)

t2 = TypeName(PackageName([Id("java"),Id("lang")]),Id("String"))
```

Adding Type Annotations (Simplified!)

Type of expressions

```
dryad-type-of :
  |[ e1 + e2 ]| -> t3
  where <type-attr> e1 => t1
      ; <type-attr> e2 => t2
      ; <is-convertible-to-numeric-type> t1
      ; <is-convertible-to-numeric-type> t2
      ; <binary-numeric-promotion-of-types> (t1, t2) => t3
```

Composing a typechecker (first attempt)

```
dryad-type-checker =
  bottomup(try(add-type-annotation))

add-type-annotation :
  e -> e{Type(ty)}
  where <type-of> e => ty
```

Annotating Variables?!

But how do we add type annotations to expressions such as variables and method invocations?

Their annotations depend on declarations in the context.

And rewrite rules only provide local information from matching.

Annotating Variables?!

But how do we add type annotations to expressions such as variables and method invocations?

Their annotations depend on declarations in the context.

And rewrite rules only provide local information from matching.

The solution: dynamic rewrite rules

Type Annotation with Dynamic Rules (Still Simplified!)

Type of variables

```
dryad-tc-declare-local-variables(rec) =
  ?[ |[ ty x; ]| | _ ]
  ; {| TypeOf :
       rules(TypeOf : Id(x) -> ty)
       ; [id | rec]
     |}

dryad-type-of :
  ExprName(Id(x)) -> ty
  where <TypeOf> Id(x) => ty

dryad-type-checker =
  dryad-tc-declare-local-variables(dryad-type-checker)
  <+ all(dryad-type-checker)
     ; try(add-type-annotation)
```

Transformation Example: Interface Conformance

```
@Implements ({"java.awt.event.MouseListener"})
public class SomeListener {
  public void mousePressed (MouseEvent event) { ... }
  public void mouseDragged (MouseEvent event) { ... }
}
```

add default (empty) implementations for missing methods

```
@Implements ({"java.awt.event.MouseListener"})
public class SomeListener implements MouseListener {
  public void mousePressed  (MouseEvent event) { ... }
  public void mouseDragged  (MouseEvent event) { ... }
  public void mouseClicked  (MouseEvent e) {}
  public void mouseReleased (MouseEvent e) {}
  public void mouseEntered  (MouseEvent e) {}
  public void mouseExited   (MouseEvent e) {}
}
```

(Example due to Yannis Smaragdakis)

Interface Conformance (1/4)

complete *all* class declarations (also nested) in a source file

```
module complete-interface
imports
  liblib                              // Stratego/XT library
  libdryad                            // Java static analysis library

strategies

  main-complete-interface =
    io-wrap(observables-wrap(complete-interface))
                                      // observe referenced classes
  complete-interface =
    topdown(try(CompleteInterface))
                                     // apply everywhere
```

Interface Conformance (2/4)

add missing methods to class annotated with Implements

```
CompleteInterface :
  type-dec|[ mod* class x { ~*decs1 } ]| ->
  type-dec|[
    mod* class x implements tname {
      ~*<conc> (decs1, decs2)
    }
  ]|
  where <get-implements-annotations> mod* => [tname]
      ; <get-method-names> decs1 => methods
      ; <create-missing-methods> (tname, methods) => decs2
```

(should be generalized a bit)

Interface Conformance (3/4)

get typename from Implements annotation in class modifiers

```
get-implements-annotations =
  fetch(?cmod|[ @pkgtname.Implements(e) ]|)
  ; <collect(?Chars(<typename-from-string>))> e

typename-from-string =
  string-tokenize(|['.'])
  ; !TypeName(PackageName(<init; map(!Id(<id>))>),Id(<last>))
```

```
@Implements({"java.awt.event.MouseListener"}) public
```

```
TypeName(
  PackageName([Id("java"),Id("awt"),Id("event")]),
  Id("MouseListener")
)
```

Interface Conformance (4/4)

```
get-method-names =
  collect(?MethodDecHead(_,_,_,Id(<id>),_,_))

create-missing-methods :
  (tname, methods) -> decs
  where <lookup-class> tname => cl
      ; <get-declared-methods> cl
      ; filter(where(<not(member)>(<get-simple-name>, methods)))
      ; map(make-default-method(|cl)) => decs

make-default-method(|cl) :
  mo -> class-body-dec|[ public tname x(param*) {} ]|
  where <get-simple-name> mo => x
      ; <get-return-type-in-class(|cl)> mo => tname
      ; <get-formal-parameter-types(|cl)
        ; map(!Param([],<id>,Id(<new>)))> mo => param*
```

(methods are compared by name; should be signature)

Interface Conformance: Discussion

- 42 LOC
  - including module header and whitespace, no comments
  - using java-front/dryad library as is
- Should be more compact
- Needed improvements to dryad library
  - clone signature of existing (abstract) method
  - get methods from current source class
  - (anti)quotation/meta-variables not complete
  - invoke parser instead of tokenize hack
  - (may already be supported, easy to realize otherwise)
- Needed improvements to Stratego
  - List matching would make matching more elegant

JavaJava: Embedding Java in Java

Supporting Java code generation with Java

- Java program that generates Java
- Representation is AST represented by objects
- Generation methods use concrete syntax

$\Rightarrow$ An extension of Java

- Assimilation reduces embedded fragments to Java code

```
public CompilationUnit run(ClassDescriptor cd)
{
  CompilationUnit result =
    |[
      import java.util.*;
      public class #[ cd.getName() ] {
        #[ attributes(cd) ]
      }
    ]|;
  return result;
}
```

Plans

An extensible compiler for Java

- Extend syntax definition
- Extend transformation pipeline
- Extend existing transformations (typechecker)

Applications

- Language extensions
- Domain-specific language embeddings and assimilations
- Active libraries and domain-specific optimizations
- Meta-programming facilities
  (e.g., alternative aspect languages)
- Software composition
- Static analysis (e.g., architecture conformance)
- ...

Issues

- Open extensibility with concurrent variants
- Composition of extensions
- Orthogonality of assimilations
- Automatic extension of transformations based on ‘semantic description’ of extension
- Trees vs graphs
- Language-specific disambiguation and desugaring of rules

Availability

Stratego/XT is open source (LGPL)

- www.stratego-language.org
- Download: source and binaries (RPM, MacOS X, Cygwin)

We want to get lots of users

- Manual: tutorial, examples, reference material
- Mailinglists
- IRC channel: freenode.net/#stratego

We also care about software engineering

- `svn.cs.uu.nl -> StrategoXT`: subversion repository
- `bugs.cs.uu.nl`: issue tracking
- `nix.cs.uu.nl`: continuous builds and releases

Context-Sensitive Transformations

There are lots of such context-sensitive transformations

- Constant propagation
- Copy propagation
- Common-subexpression elimination
- Partial evaluation
- Function inlining
- Dead code elimination

Dynamic rules can be used for those as well

Constant Folding

```
y := x * (3 + 4) ⇒ y := x * 7
```

```
EvalBinOp :
  expr|[ i1 + i2 ]| -> expr|[ i3 ]|
  where <addS> (i1, i2) => i3

EvalRelOp :
  expr|[ i1 > i2 ]| -> <eval-rel-op(<gtS> (i1, i2))>

eval-rel-op(s) =
  if s then !expr|[ true ]| else !expr|[ false ]| end

EvalIf :
  |[ if(false) stm1 else stm2 ]| -> bstm|[ stm2 ]|

EvalIf :
  |[ if(true) stm1 else stm2 ]| -> bstm|[ stm1 ]|
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = b + 3;
b = foo();
a = b + c
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = b + 3;
b = foo();
a = b + c
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = b + 3;
b = foo();
a = b + c
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = b + 3;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = b + 3;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = b + 3;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 1 + 3;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 1 + 3;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + c
```

```
b -> 1
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + c
```

```
b -> 1 & c -> 4
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + c
```

```
b -> 1 & c -> 4
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + c
```

```
b -    & c -> 4
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + c
```

```
b -     & c -> 4
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + 4
```

```
b -    & c -> 4
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

Constant Propagation and Folding in Straight-Line Code

```
b = 1;
c = 4;
b = foo();
a = b + 4
```

```
b -    & c -> 4 & a -
```

```
prop-const =
  PropConst
  <+ prop-const-assign
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x = <prop-const => e> ]|
  ; if <is-value> e then
      rules( PropConst : |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end
```

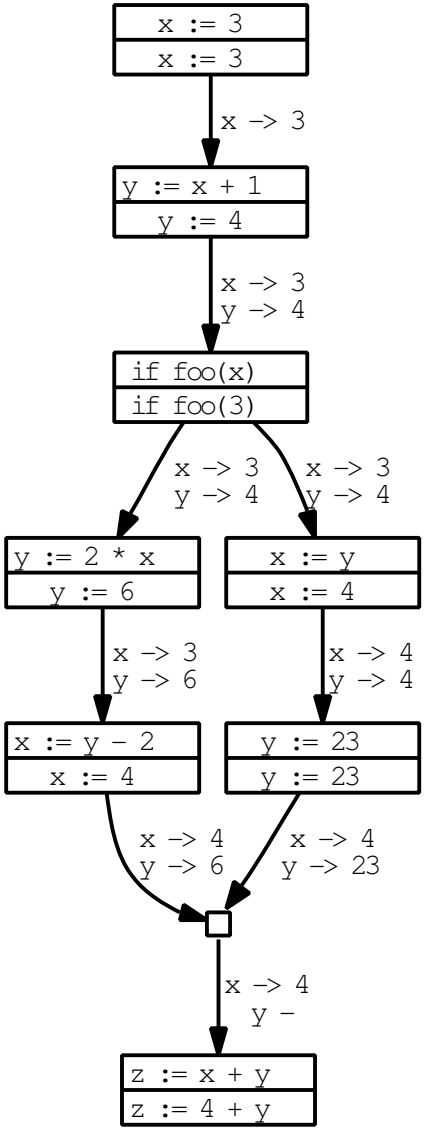
Flow-Sensitive Transformation

```
begin
 x := 3;
 y := x + 1;
 if foo(x) then
   y := 2 * x;
   x := y - 2;
 else
   x := y;
   y := 23;
 end
 z := x + y;
end
```

```
begin
 x := 3;
 y := 4;
 if foo(3) then
   y := 6;
   x := 4;
 else
   x := 4;
   y := 23;
 end
 z := 4 + y;
end
```

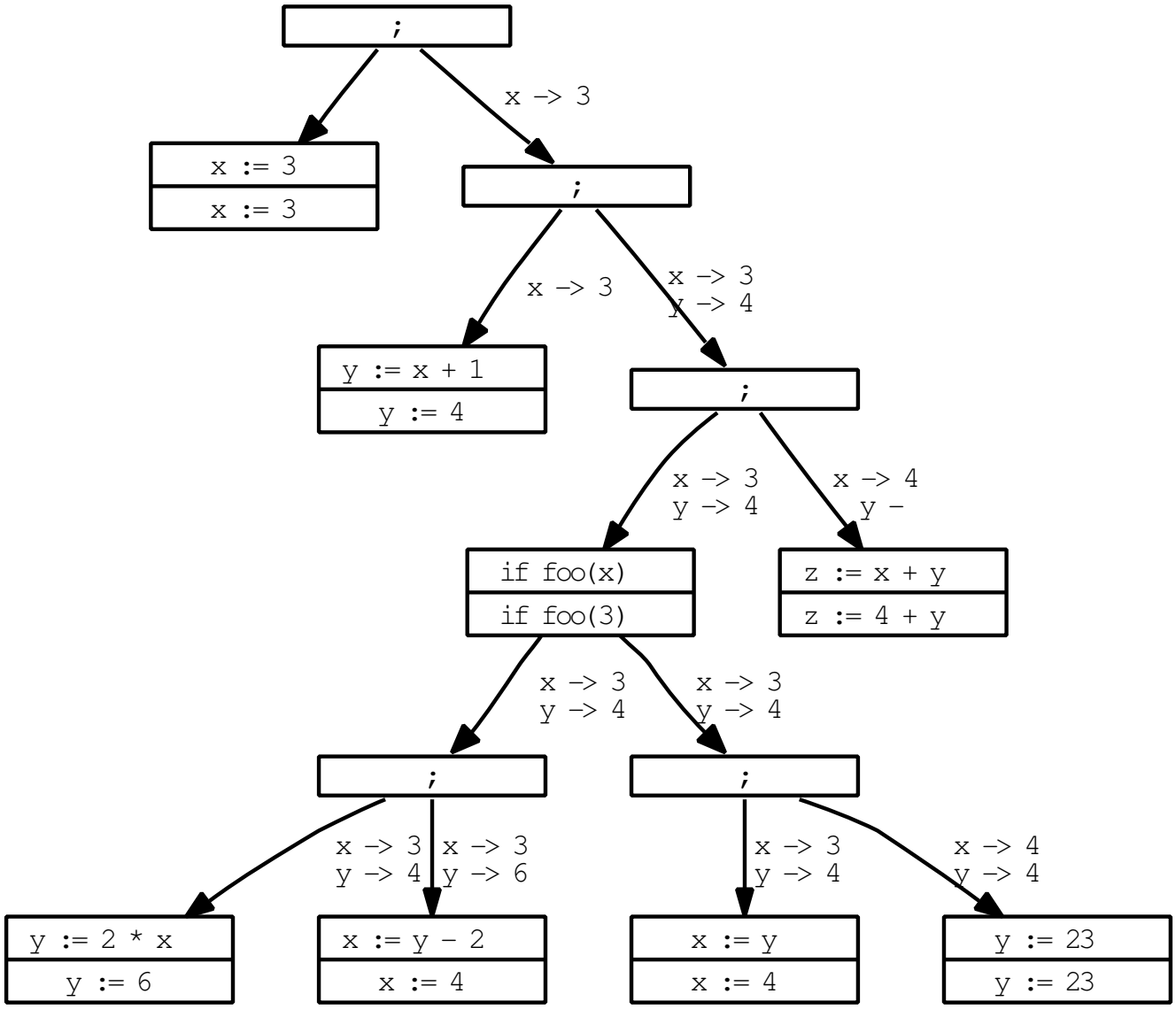
Flow-Sensitive Transformation

```
begin
 x := 3;
 y := x + 1;
 if foo(x) then
   y := 2 * x;
   x := y - 2;
 else
   x := y;
   y := 23;
 end
 z := x + y;
end
```




```
begin
 x := 3;
 y := 4;
 if foo(3) then
   y := 6;
   x := 4;
 else
   x := 4;
   y := 23;
 end
 z := 4 + y;
end
```

Constant propagation in abstract syntax tree



Flow-Sensitive Constant Propagation

```
propconst =
  PropConst
  <+ propconst-assign
  <+ propconst-if
  <+ propconst-while
  <+ all(propconst); try(EvalBinOp <+ EvalRelOp)

propconst-if =
  If(propconst, id, id)
  ; (EvalIf; propconst
     <+ If(id, propconst, id)
          /PropConst\ If(id, id, propconst))

propconst-while =
  ?While(_, _)
  ; /PropConst\* While(propconst, propconst)
```