MetaBorg in Action

Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT

Martin Bravenboer, René de Groot, Eelco Visser

Department of Information & Computing Sciences
Utrecht University, The Netherlands

July 5, 2005

MetaBorg: DSL Embedding and Assimilation

Generative programming methods and program transformation techniques can be used to overcome the lack of abstraction in general-purpose languages.

MetaBorg Method

Introduce domain abstractions in a general-purpose language.

- domain notation (syntax)
- code organization and structure
- domain specific analysis

1. *Embedding* of domain-specific language
2. *Assimilation* of embedded domain code

Example: Swul in Java

```
JMenuBar menubar = new JMenuBar();
JMenu filemenu = new JMenu("File");
JMenuItem newfile = new JMenuItem("New");
JMenuItem savefile = new JMenuItem("Save");
newfile.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N, 2));
savefile.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S, 2));
filemenu.add(newfile);
filemenu.add(savefile);
menubar.add(filemenu);

JPanel buttons = new JPanel(new GridLayout(1, 2));
JPanel south = new JPanel(new BorderLayout());
buttons.add(new JButton("Ok"));
buttons.add(new JButton("Cancel"));
south.add(BorderLayout.EAST, buttons);

JPanel panel = new JPanel(new BorderLayout());
panel.add(BorderLayout.CENTER, new JScrollPane(new JTextArea(20, 40)));
panel.add(BorderLayout.SOUTH, south);
```

Example: Swul in Java

```
menubar = {
  menu {
    text = "File"
    items = {
      menu item { text = "New"   accelerator = ctrl-N }
      menu item { text = "Save"  accelerator = ctrl-S }
    }
  }
}

content = panel of border layout {
  center = scrollpane of textarea { rows = 20  columns = 40 }

  south = panel of border layout {
    east = panel of grid layout {
      row = {
        button of "Accept"
        button of "Cancel"
      }
    }
  }
}
```

Example: Regular Expressions in Java

```
Pattern ipline = Pattern.compile(
  "( ( [0-1]?\\d{1,2} \\. ) | ( 2[0-4]\\d \\. ) | ( 25[0-5] \\. ) ){3}
   ( ( [0-1]?\\d{1,2}     ) | ( 2[0-4]\\d     ) | ( 25[0-5]     ) )");

if(ipline.matcher(input).matches()) {
  System.out.println("Input is an ip-number.");
} else {
  System.out.println("Input is NOT an ip-number.");
}
```

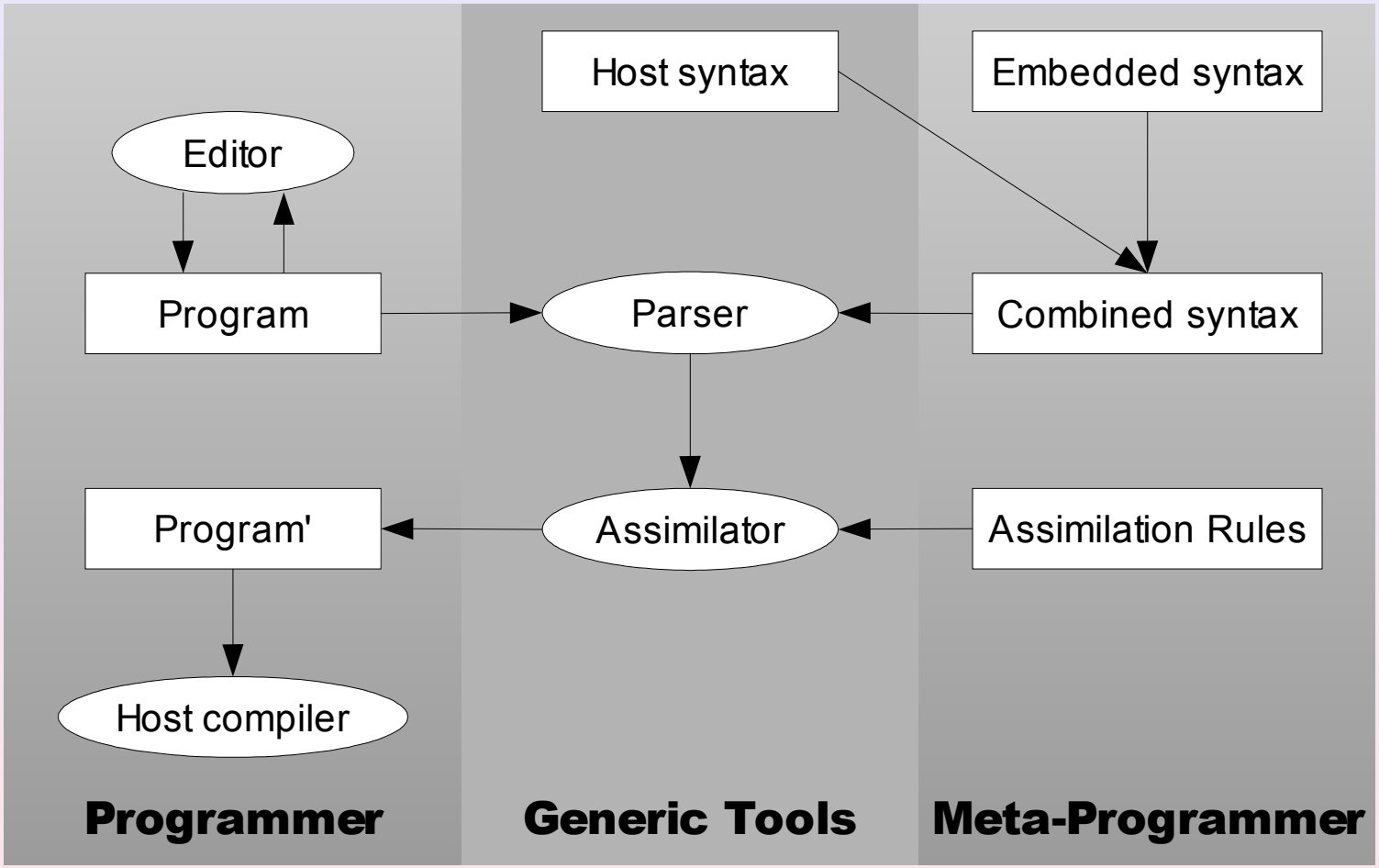
```
regex ipline = [/
    ( ( [0-1]?\d{1,2} \. ) | ( 2[0-4]\d \. ) | ( 25[0-5] \. ) ){3}
    ( ( [0-1]?\d{1,2}    ) | ( 2[0-4]\d    ) | ( 25[0-5]    ) )
  /];

if( input ~? ipline ) {
  System.out.println("Input is an ip-number.");
} else {
  System.out.println("Input is NOT an ip-number.");
}
```

Example: Generating Java in Java

```
String vName = "propertyChangeListeners";
jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");
jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
```

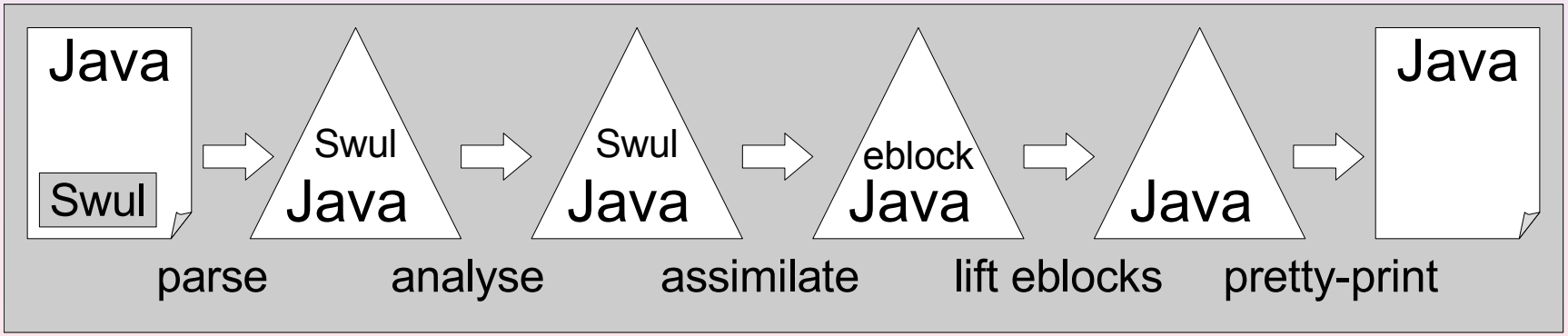
```
String x = "y";
CompilationUnit dec = |[ public class Foo {
                           public int bar() {
                             return #[x] * x;
                           }
                         } ]|;
```

MetaBorg: Architecture



Swul Implementation: Overview

1. Swul syntax definition
2. Embed Swul syntax in Java
3. Extend Java type checker with Swul typing rules
4. Assimilate embedded Swul to Swing Java code
5. Profit!



Swul: Syntax

```
context-free syntax
  ComponentType Props?     -> Component {cons("Component")}
  "{" Prop* "}"            -> Props     {cons("Props")}
  PropType "=" PropValues -> Prop       {cons("Prop")}

  "{" Component* "}" -> PropValues {cons("PropMultiValue")}
  Component          -> PropValues {cons("PropSingleValue")}

context-free syntax
  "panel"      -> ComponentType {cons("JPanel")}
  "tabbedpane" -> ComponentType {cons("JTabbedPane")}
  "tab"        -> ComponentType {cons("Tab")}

  "border" "layout" -> ComponentType {cons("BorderLayout")}
  "grid" "layout"   -> ComponentType {cons("GridLayout")}
  "box" "layout"    -> ComponentType {cons("BoxLayout")}

  "content" -> PropType {cons("Content")}
  "layout"  -> PropType {cons("Layout")}
  "tabs"    -> PropType {cons("Tabs")}
  "title"   -> PropType {cons("Title")}
```

Swul: Syntax

Syntax definition for modifiers:

```
context-free syntax
  (Modifier "-")* KeyEvent -> Prop {cons("Accelerator")}
  "ctrl"  -> Modifier {cons("CtrlModifier")}
  "alt"   -> Modifier {cons("AltModifier")}
  "shift" -> Modifier {cons("ShiftModifier")}
  "meta"  -> Modifier {cons("MetaModifier")}
```

Embed Swul in Java:

```
context-free syntax
  SwulComponent -> JavaExpr      {avoid, cons("ToExpr")}
  JavaExpr      -> SwulComponent {avoid, cons("FromExpr")}
```

Reject cyclic derivations:

```
context-free priorities
  JavaExpr -> SwulComponent > SwulComponent -> JavaExpr
```

Swul: Assimilation Rules

```
SwulAs-JButton :
  |[ button { ps* } ]|{x} -> |[ {| x = new JButton(); bstm* | x |} ]|
  where <map(SwulAs-JButtonProp(|x))> ps* => bstm*

SwulAs-AbstractButtonProp(|x) :
  |[ mnemonic = k ]| -> |[ x.setMnemonic(e); ]|
  where <SwulAs-KeyEvent> k => e

SwulAs-JPanel :
  |[ panel of c ]|{x} -> |[ {| x = new JPanel(); x.setLayout(e); | x |} ]|
  where <SwulAs-LayoutManager(|x)> c => e

SwulAs-GridLayout(|x) :
  |[ grid layout {ps*} ]|{y} -> |[ {| y=new GridLayout(i,j); |y| bstm*|} ]|
  where <nr-of-rows> ps*    => i
      ; <nr-of-columns> ps* => j
      ; <map(SwulAs-LayoutProp(|x,y))> ps* => bstm*

SwulAs-LayoutProp(|x,y) :
  |[ horizontal gap = c ]| -> |[ y.setHgap(e) ; ]|
  where <SwulAs-Component> c => e
```

Swul: Traversal Strategy

Traversal strategy for Swul Assimilation

```
swul-assimilate =
      class-declaration
  <+ class-initializer
  <+ class-method
  <+ swul-expression
  <+ all(swul-assimilate)

class-initializer :
  |[ static { bstm1* } ]| -> |[ static { bstm2* } ]|
  where {| FieldModifier :
           rules(FieldModifier :+ _ -> |[ static ]|)
           ; <swul-assimilate> bstm1* => bstm2*
         |}

swul-expression =
  ?ToExpr(<SwulAs-Component>)
```

Regular Expressions: Analysis

```
dryad-type-of :
  ToBooleanExpr(x,y) -> Boolean()
  where <type-attr> x => TypeString()
      ; <type-attr> y => Regex()

dryad-type-of :
  Assign(x, y) -> Regex()
  where <type-attr> x => Regex()
      ; <type-attr> y => Regex()
```

```
public void bar(java.lang.String input)
{
  java.lang.String ipline = "...";
  if(input ~? ipline)
  {
    java.lang.System.out.println("Input is an ip-number.");
  }
  else
  {
    java.lang.System.out.println("Input is NOT an ip-number.");
  }
}
```

JavaJava: Syntax

```
context-free syntax
  "|[" CompUnit  "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" TypeDec   "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" BlockStm  "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" BlockStm* "]|" -> MetaExpr {cons("ToMetaExpr")}

context-free syntax
  "#[" MetaExpr "]"-> ID   {cons("FromMetaExpr")}
  "#[" MetaExpr "]"-> Expr {cons("FromMetaExpr")}

variables
  MetaVarID -> ID
  MetaVarID -> Expr
  MetaVarID -> {Expr ","}+

lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> MetaVarID
```

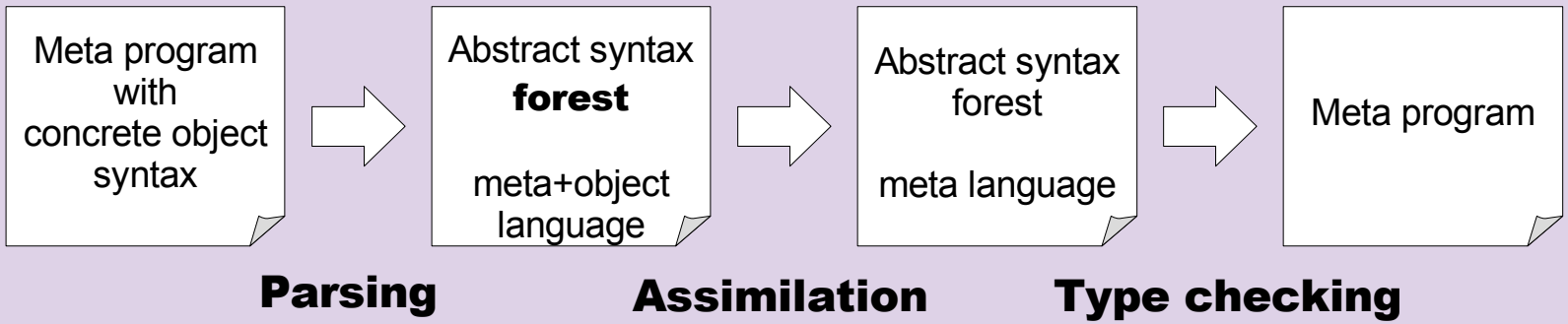
JavaJava: Assimilation to Eclipse JDT Core

```
Assimilate(rec) :
  |[ return; ]| -> |[ _ast.newReturnStatement() ]|

Assimilate(r) :
  type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

Assimilate(r) :
  |[ e; ]| -> |[ ast.newExpressionStatement(~e: <r> e) ]|

Assimilate(rec) :
  |[ e.y(e*) ]| -> |[
    {| MethodInvocation x = _ast.newMethodInvocation();
       x.setName(~e:<AssimilateId(rec)> y);
       x.setExpression(~e:<rec> e);
       bstm* | x |} ]|
  where <newname> "inv" => x
      ; <AssimilateArgs(rec | x)> e* => bstm*

Assimilate(r) :
  e |[ i ]| -> |[
      {| NumberLiteral x = ast.newNumberLiteral();
         x.setToken("~i");
       | x |} ]|
    where <newname> "int" => x
```

Embedding of Java in Java is ambiguous

```
String x = "y";
CompilationUnit dec = |[ public class Foo {
                           public int bar() {
                             return #[x] * x;
                           }
                         } ]|;
```

Pipeline for assimilation and disambiguation



JavaJava: Disambiguation

```
Assign(ExprName(Id("dec")),
       1> ToMetaExpr(  CompUnit(... ClassDec(... Id("Foo")...) ...)  )
       2> ToMetaExpr(  ClassDec(... Id("Foo") ...)   )
       3> ToMetaExpr([ ClassDec(... Id("Foo") ...) ] ) )

1> {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
      TypeDeclaration class_0 = _ast.newTypeDeclaration();
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 |}
2> {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
      class_1.setName(_ast.newSimpleName("Foo")); ... |class_1 |}
3> {| List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();
      decs_0.add( ... ); ... | decs_0 |}
```

Ambiguity Lifting

```
dec = 1> CompUnit 2> TypeDec 3> List<BodyDec>

1> dec = CompUnit 2> dec = TypeDec 3> dec = List<BodyDec>

 f(1> CompUnit 2> TypeDec 3> List<BodyDec>)

 1> f(CompUnit) 2> f(TypeDec) 3> f(List<BodyDec>)
```

Discussion: Syntax Definition

Essential ingredients:

- Modular syntax definition
- Full and automatic parser generation

SDF and SGLR

- Modular syntax definition
- Defines lexical and context-free syntax
- Declarative disambiguation
- Context-sensitive lexical analysis
  - Parsing 'combined' with lexical analysis
  - Keywords not automatically reserved
- Allows ambiguities

Implementation: SGLR (Scannerless Generalized LR parsing)

Discussion: Program Transformation

Stratego

- **Rewrite rules**
  - Basic translation of DSL
- **Strategies**
  - Control application of the rewrite rules
  - Rules are not applicable everywhere
- **Generic Term Traversals**
  - Avoid handcoding traversals over a large language
  - Handle only the important cases
- **Scoped dynamic rewrite rules**
  - Context-sensitive translation
- **Concrete Object Syntax**
  - Write rewrite rules using syntax of DSL and GPL

Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

Example: Swul Events

```
class MenuEvent {
  static void newFileEvent() { ... }

  static void main(String[] ps) { ...
    menubar = {
      menu {
        text = "File"
        items = {
          menu item {
            text = "New"
            action event = { newFileEvent(); }
          }
          menu item {
            text = "Exit"
            action event = { System.exit(0); }
          }
          ...
```

Example: Swul Events

```
class MenuEvent {
  private static ClassHandler_0 classHandler_0 = new ClassHandler_0();
  public static void newFileEvent() { ... }

  public static void main(String[] ps) { ...
    JMenuItem_0 = new JMenuItem();
    JMenuItem_0.setText("New");
    JMenuItem_0.addActionListener(
      EventHandler.create(..., ClassHandler_0, "ActionListener_0", ""));
    ... }

  public static class ClassHandler_0 {
    public void ActionListener_0(ActionEvent event) { newFileEvent(); }
    public void ActionListener_1(ActionEvent event) { System.exit(0); }
  }
}
```

Example: Regular Expression Rewrites

```
String input = ...

regex body = [/ <body[^>]*?> .* </body> /]

regex amp = [/ & /] -> [/ &amp; /];
regex lt  = [/ < /] -> [/ &lt;  /];
regex gt  = [/ > /] -> [/ &gt;  /];
regex escape = amp <+ lt <+ gt;

regex noattach  =
  [/ <a[^>]*?> \s* Attach \s* </a> /]
    -> [/ <strike> Attach </strike> /];

regex edittopic =
  [/ %EDITTOPIC% /]
    -> [/ <a href="%EDITURL%"><b>${editAction}</b></a> /];

input ~= one(body <~> all(edittopic <+ noattach <+ escape))
```

Example: Regular Expression Rewrites

```
regex amp = [/ & /] -> [/ &amp; /];
regex lt  = [/ < /] -> [/ &lt;  /];
regex gt  = [/ > /] -> [/ &gt;  /];
regex escape = amp <+ lt <+ gt;

input ~= all(escape);
```

```
Pattern amp = Pattern.compile("&", Pattern.MULTILINE);
Pattern lt = Pattern.compile("<", Pattern.MULTILINE);
Pattern gt = Pattern.compile(">", Pattern.MULTILINE);

StringBuffer buffer_0 = new StringBuffer();

Matcher matcher_2 = amp.matcher(input);
Matcher matcher_1 = lt.matcher(input);
Matcher matcher_0 = gt.matcher(input);
```

Example: Regular Expression Rewrites

```
int matchedto_0;
boolean matched_0 = false;
label_0 : for(int i_0 = 0; i_0 < input.length(); i_0++) {
  matcher_2.region(i_0, input.length());
  matcher_1.region(i_0, input.length());
  matcher_0.region(i_0, input.length());

  if(matcher_2.lookingAt()) {
    matchedto_0 = matcher_2.end();
    buffer_0.append("&amp;");
    i_0 = matchedto_0 - 1;
    matched_0 = true;
    continue label_0;
  }
  if(matcher_1.lookingAt()) { ... buffer_0.append("&lt;") ... }
  if(matcher_0.lookingAt()) { ... buffer_0.append("&gt;") ... }
  buffer_0.append(input.charAt(i_0));
}
if(matched_0) { } else {
  buffer_0.delete(buffer_0.length() - input.length(), buffer_0.length());
  buffer_0.append(input);
}
input = buffer_0.toString();
```

Discussion: Intrastructure for Program Transformation

- Generic tools and methods
- Still much work to implement basic infrastructure
- Language-specific support

Stratego/XT Extensions Provide

Excellent syntactic support for Java

- Java syntax definition (5.0)
- Java pretty-printer (5.0)

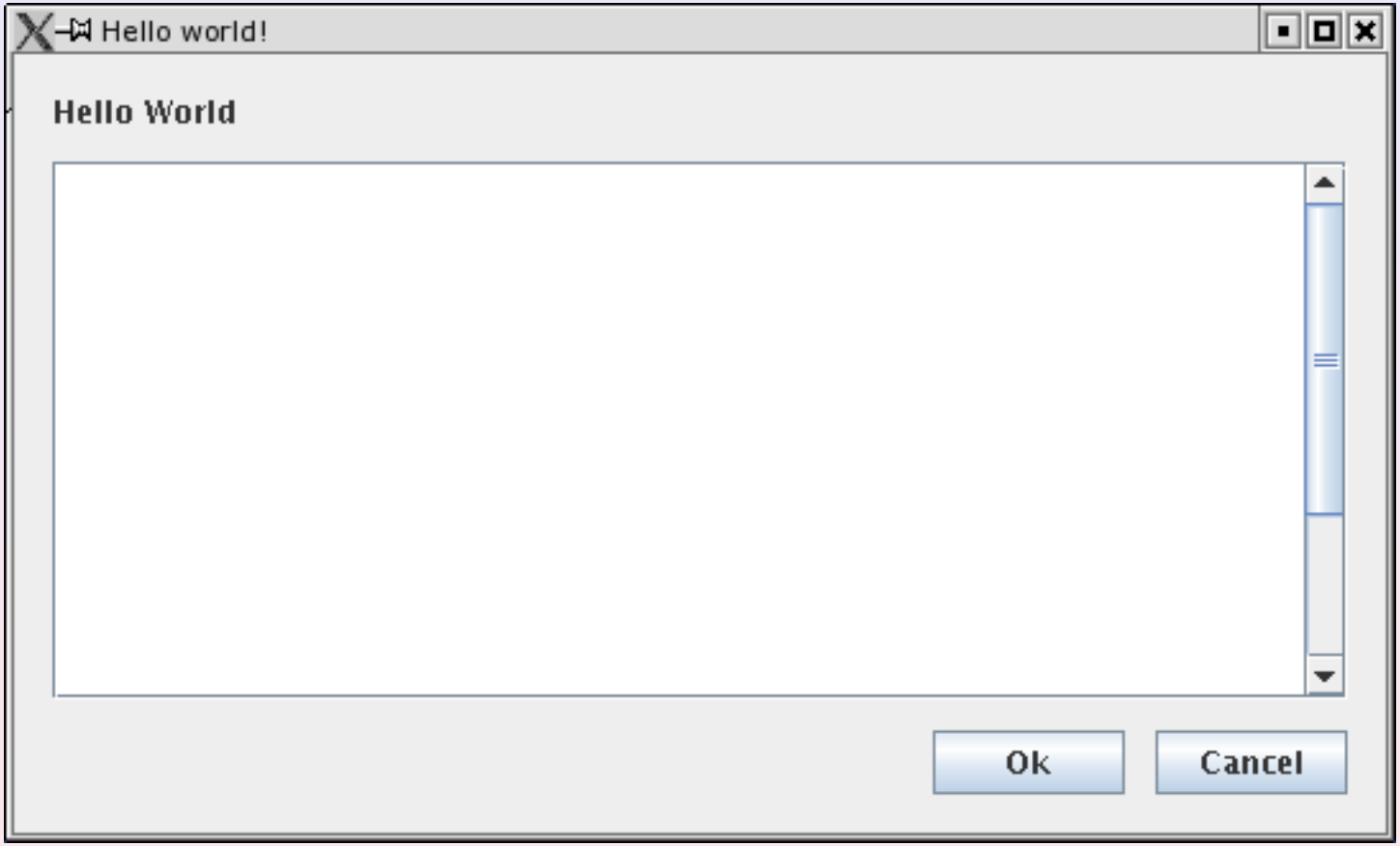
Quickly improving semantic support for Java

- Java disambiguation
- Java type information
- API for reflecting over Java code

Part II

MetaBorg

Example 1: Implement a GUI



Example 1: Implement a GUI using Java/Swing

```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons);
    panel.add(BorderLayout.SOUTH, south);

    ...
```

Example 1: Implement a GUI using Java/Swing

```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons);
    panel.add(BorderLayout.SOUTH, south);

    ...
```

Does not correspond to hierarchical structure of the user-interface.

Example 1: Implement a GUI using Java/Swing

```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons);
    panel.add(BorderLayout.SOUTH, south);

    ...
```

Does not correspond to hierarchical structure of the user-interface.

Example 1: Implement a GUI using Java/Swing

```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons);
    panel.add(BorderLayout.SOUTH, south);

    ...
```

Does not correspond to hierarchical structure of the user-interface.

Analysis of user-interface structure is impossible or difficult.

Concrete Syntax for Objects

Domain abstraction in general-purpose languages

- Semantic domain abstraction
  - Designed for extensibility and reuse
- No syntactic domain abstraction
  - Only generic syntax of method invocations
  - No domain-specific notation and composition

Concrete Syntax for Objects

Domain abstraction in general-purpose languages

- Semantic domain abstraction
  - Designed for extensibility and reuse
- No syntactic domain abstraction
  - Only generic syntax of method invocations
  - No domain-specific notation and composition

Concrete syntax for domain abstractions

- Semantic domain abstraction
- Syntactic domain abstraction

The MetaBorg Method:

1. *Embedding* of domain-specific language
2. *Assimilation* of embedded domain code

Example 1: Implement a GUI using Concrete Syntax

```
public class HelloWorld {
  public static void main(String[] ps) {

    JPanel panel = panel of border layout {
      north = label "Hello World"

      center = scrollpane of textarea {
        rows    = 20
        columns = 40
      }

      south = panel of border layout {
        east = panel of grid layout {
          row = {
            button "Ok"
            button "Cancel"
          }
        }
      }
    };

    ...
```

Example 1: Implement a GUI using Concrete Syntax

Syntax reflects the hierarchical structure of the user-interface.

```
public class HelloWorld {
  public static void main(String[] ps) {

    JPanel panel = panel of border layout {
      north = label "Hello World"

      center = scrollpane of textarea {
        rows    = 20
        columns = 40
      }

      south = panel of border layout {
        east = panel of grid layout {
          row = {
            button "Ok"
            button "Cancel"
          }
        }
      }
    };

    ...
```

Example 1: Implement a GUI using Concrete Syntax

```
public class HelloWorld {
  public static void main(String[] ps) {

    JPanel panel = panel of border layout {
      north = label "Hello World"

      center = scrollpane of textarea {
        rows    = 20
        columns = 40
      }

      south = panel of border layout {
        east = panel of grid layout {
          row = {
            button "Ok"
            button "Cancel"
          }
        }
      }
    };

    ...
```

Syntax reflects the hierarchical structure of the user-interface.

The interaction between the domain-specific and general-purpose code is seamless.

Example 2: Code Generation

Suppose we want to generate:

```
if(propertyChangeListeners == null)
  return;

PropertyChangeEvent event =
  new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
  ((PropertyChangeListener)
      propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

Parameterized by the name of the listeners variable.

(Fragment generated by Castor)

Example 2: Code Generation using Strings

```
String vName = "propertyChangeListeners";

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Example 2: Code Generation using Strings

```
String vName = "propertyChangeListeners";

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Uses the Java syntax: the syntax of the domain.

Example 2: Code Generation using Strings

```
String vName = "propertyChangeListeners";

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, new

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Uses the Java syntax: the syntax of the domain.

No syntactic checks of the generated code.

Escaping to the meta language is difficult.

Code generator tries to do some pretty printing.

Further processing of the code is impossible.

Example 2: Code Generation using Abstract Syntax Trees

```
VariableDeclarationFragment fragment =
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

Example 2: Code Generation using Abstract Syntax Trees

Extremely verbose and unclear: 90 lines of code!

```
VariableDecl
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"
ClassInstanceCreation newi = _ast.newClassI
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

Does not correspond to the structure of the code to be generated.

Example 2: Code Generation using Abstract Syntax Trees

Extremely verbose and unclear: 90 lines of code!

```
VariableDecl
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"
ClassInstanceCreation newi = _ast.newClassI
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

Does not correspond to the structure of the code to be generated.

Code is syntactically checked by host language compiler and further processing is possible.

Don't worry about the layout.

Example 2: Code Generation using Concrete Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(x == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < x.size(); c++) {
    ((PropertyChangeListener)
      x.elementAt(c)).propertyChange(event);
  }
]|;
```

Example 2: Code Generation using Concrete Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(x == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < x.size(); c++) {
    ((PropertyChangeListener)
      x.elementAt(c)).propertyChange(event);
  }
]|;
```

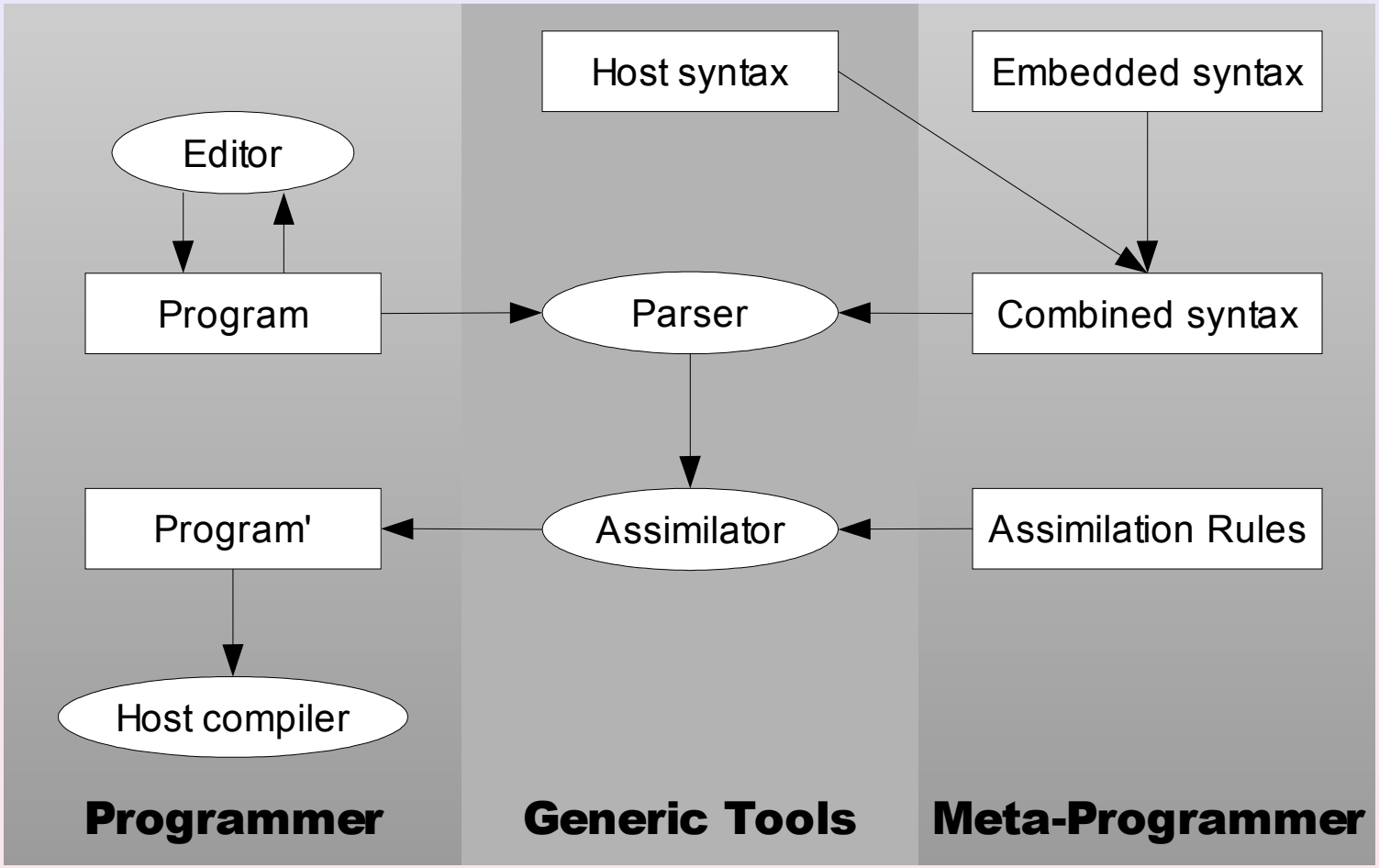
Uses the syntax of the domain: Java.

Syntax of the generated code is checked and further processing is possible.

Separate pretty-printer: don't worry about the layout.

Support for interaction between the generated code and the meta language.

Architecture of the MetaBorg Method



Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

Characteristics of the MetaBorg Method

1. **Syntactic**
   - Syntax of embedded code checked at compile-time
2. **No restrictions on syntax**
   - Arbitrary context-free languages
   - Embed languages with different lexical syntax
3. **Not specific to single host language**
   - Embed domain syntax in any host language
4. **Interaction with host language**
   - Weave embedded code in host language
5. **Combination of extensions**
   - Embed multiple languages
6. **No restrictions on assimilation**
   - Context-sensitive, global
   - Optimization, semantic checks

MetaBorg Applied: Java/Pairs

Syntactic domain abstraction

```
(Integer, String) t = (1, "Hello world!");
```

API: Semantic domain abstraction

```
public class Pair<F, S> {
  public Pair(F first, S second) ...
  public static <F1, S1> Pair<F1, S1> construct(F1 f, S1 s)

  public F getFirst()  ...
  public void setFirst(F value)  ...
}
```

After assimilation

```
Pair<Integer, String> t = Pair.construct(1 , "Hello world!");
```

Realizing Java/Pairs

Embed syntax for Pairs in Java

```
module Java-Pair imports Java-15
exports
  context-free syntax
    "(" Expr "," Expr ")" -> Expr {cons("NewPair")}
    "(" Type "," Type ")" -> Type {cons("PairType")}
```

Assimilate Pairs to Pair API

```
module Java-Pair-Assimilate imports Java-Pair
rules
  AssimilatePair :
    expr |[ (e1, e2) ]| -> expr |[ Pair.construct(e1, e2) ]|

  AssimilatePair :
    type |[ (t1, t2) ]| -> type |[ Pair<t1, t2> ]|
```

Realizing Java/Pairs

Embed syntax for Pairs in Java

```
module Java-Pair imports Java-15
exports
  context-free syntax
    "(" Expr "," Expr ")" -> Expr {cons("NewPair")}
    "(" Type "," Type ")" -> Type {cons("PairType")}
```

Assimilate Pairs to Pair API

```
module Java-Pair-Assimilate imports Jav
rules
  AssimilatePair :
    expr |[ (e1, e2) ]| -> expr |[ Pair.construct(e1, e2) ]|

  AssimilatePair :
    type |[ (t1, t2) ]| -> type |[ Pair<t1, t2> ]|
```

Assimilation rules use concrete syntax for Java and Pairs as well!

MetaBorg Applied: Java/Swul

Syntactical domain abstraction

```
"panel" "of" Layout -> Component {cons("Panel")}
"button" String     -> Component {cons("ButtonText")}
```

Embedding of domain specific language

```
Component -> Expr      {cons("ToExpr")}
Expr      -> Component {cons("FromExpr")}
```

Assimilation rules

```
Swulc-Component :
  swul |[ button e ]| -> expr |[ new JButton(e) ]|

Swulc-Layout :
  swul |[ grid layout {ps*} ]| -> expr |[ new GridLayout(i,j) ]|
  where <nr-of-rows> ps* => i
      ; <nr-of-columns> ps* => j
```

MetaBorg Applied: JavaJava

Embed Java syntax in Java

```
"e" [0-9]*      -> Expr          {prefer}
"e" [0-9]* "*" -> {Expr ","}*   {prefer}
"type" "|[" Type "]|" -> MetaExpr {cons("ToMetaExpr")}
```

Assimilation rules for Eclipse JDT Core API

```
Assimilate(r) :
  type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

Assimilate(r) :
  |[ e; ]| -> |[ ast.newExpressionStatement(~e: <r> e) ]|

Assimilate(r) :
  |[ y(e*) ]| -> |[
    {| MethodInvocation x = ast.newMethodInvocation();
       x.setName(ast.newSimpleName("~y"));
       bstm* | x |}
  ]|
  where <newname> "inv" => x
      ; <ExplodeArgs(r | x)> e* => bstm*
```

MetaBorg Applied: JavaJava

Embed Java syntax in Java

```
"e" [0-9]*      -> Expr          {prefer}
"e" [0-9]* "*" -> {Expr ","}*   {prefer}
"type" "|[" Type "]|" -> MetaExpr {cons("ToMetaExpr")}
```

Assimilation rules for Eclipse JDT Core API

```
Assimilate(r) :
  type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

Assimilate(r) :
  |[ e; ]| -> |[ ast.newExpressionStatement(~e: <r> e) ]|

Assimilate(r) :
  |[ y(e*) ]| -> |[
    {| MethodInvocation x = ast.newMeth
       x.setName(ast.newSimpleName("~y"
       bstm* | x |}
  ]|
  where <newname> "inv" => x
      ; <ExplodeArgs(r | x)> e* => bstm
```

To make the implementation of assimilation rules easier, declarations and statements are allowed in expressions.

MetaBorg Foundations

MetaBorg embeddings are relatively easy to implement. Why?

- SDF
  - Modular syntax definition
  - Defines lexical and context-free syntax
  - Declarative disambiguation
  - Allows ambiguities
- SGLR
  - Scannerless Generalized LR parsing
  - Lexical analysis is context-sensitive
- Stratego
  - Strategies and rewrite rules
  - Meta-programming with concrete syntax

All available and proven technology!

Scope of MetaBorg

- Meta programming
  - *Code generation* (run-time)
  - Annotation processing
- *Graphical user interfaces*
- Embedded query languages
  - XPath, XQuery, SQL, JDOQL
- Language processing
  - Context-free grammars
  - Regular expressions
- *XML processing*
- Concurrency abstractions
- ...

Available as prototype in JavaBorg

Conclusion

MetaBorg: Concrete Syntax for Domain Abstractions

- *Embedding* of domain-specific language
- *Assimilation* of embedded domain code

Embedded domain-specific languages . . .

- make code more readable.
- encourage a better style of programming.
- future work: integration in compilers, debuggers, refactoring tools, documentation generators, etc.

`http://www.metaborg.org`

Part III

Infrastructure for Java Transformation Systems

Planet Stratego/XT

- **Stratego** – Generic language for program transformation
- **XT** – Generic infrastructure for transformation systems
- **XT Orbit** – Language specific tools

Satellite Java

- **Java Front**: Syntax related infrastructure for Java
- **Dryad**: Components for Java transformation systems
- **AspectJ Front**: extension of Java Front

Goals

- Complete tool set for Java meta-programming systems
- Compatible and up to date
- Validation: apply Stratego/XT to real-life language
- World domination

language specific

end-programmer applications

compilation

code generation

optimization

static analysis

reflection

language extension

annotations

code generation

refactoring

aspect weaving

declarative transformation

model

types

binary

disambiguation

semantics

pretty printing

embedding

ast

syntax definition

syntax

generic

rewrite rules

traversals

? !

concrete syntax

library

dynamic rules

stratego

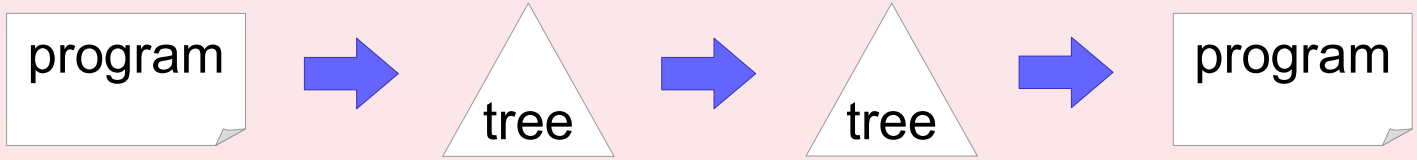
sdf

aterm

rtg

gpp

xt



Java Front

- **Syntax Definition**
- **Parser**
- **Pretty-Printer**
- **Embedding of Java**

Syntax Definition

SDF grammar for Java 2 version 1.5
(i.e. generics, enums, annotations, . . .)

- Modular
- Structure of Java Language Specification, 3^{rd} Edition
- Declarative disambiguation
  - Single expression non-terminal
  - Context-free priorities
  - Lexical restrictions
  - Rejections
- Integrated lexical and context-free syntax
  - Important for language extension (AspectJ)

Syntax Definition: Ambiguities

```
lexical restrictions
  "+" -/- [\+]
  ID  -/- [a-zA-Z0-9\_\$]
  FloatNumeral -/- [fFdD]

lexical syntax
  Keyword -> ID {reject}

context-free priorities ...
  > {left: Expr "*"   Expr -> Expr
           Expr "/"   Expr -> Expr }
  > {left: Expr "+"   Expr -> Expr
           Expr "-"   Expr -> Expr }
  > {left: Expr "<<"  Expr -> Expr
           Expr ">>"  Expr -> Expr }
  > {left: Expr "instanceof" RefType -> Expr
           Expr "<"   Expr -> Expr
           Expr ">"   Expr -> Expr }
  > {left: Expr "=="  Expr -> Expr
           Expr "!="  Expr -> Expr }
  >        Expr "&"   Expr -> Expr
  >        Expr "^"   Expr -> Expr
  >        Expr "|"   Expr -> Expr ...
```

Syntax Definition: Ambiguities: Cast Expressions

JLS: Cast Expressions

(ReferenceType) UnaryExpressionNotPlusMinus
(PrimitiveType) UnaryExpression

JLS: Different Priorities

```
$ echo "(Integer) - 2" | parse-java -s Expr | pp-aterm
Minus(ExprName(Id("Integer")), Lit(Deci("2")))

$ echo "(Integer) (- 2)" | parse-java -s Expr | pp-aterm
CastRef(
  ClassOrInterfaceType(TypeName(Id("Integer")), None)
, Minus(Lit(Deci("2")))
)

$ echo "(int) - 2" | parse-java -s Expr | pp-aterm
CastPrim(Int, Minus(Lit(Deci("2"))))
```

Compare to C: syntactical ambiguity (see Transformers)

Syntax Definition: Ambiguities: Cast Expressions

Java Front: Cast Expressions

```
"(" PrimType ")" Expr -> Expr {cons("CastPrim")}
"(" RefType  ")" Expr -> Expr {cons("CastRef")}
PrimType -> Type
RefType  -> Type
```

Java Front: Cast Priorities

```
context-free priorities
  "(" RefType ")" Expr -> Expr
> { "++"  Expr -> Expr
    "--"  Expr -> Expr
    "+"   Expr -> Expr
    "-"   Expr -> Expr }

context-free priorities
  "(" PrimType ")" Expr -> Expr
> {left:
    Expr "*" Expr -> Expr
    Expr "/" Expr -> Expr
    Expr "%" Expr -> Expr }
```

Syntax Definition: Context-Dependent Ambiguities

Java is an ambiguous language

- `import java.util.ArrayList`

Package, typename

```
TypeImportDec(
  TypeName(
    PackageOrTypeName(PackageOrTypeName(Id("java")), Id("util"))
  , Id("ArrayList")
  )
)
```

- `System.out.println("Hello world")`

Package, typename, field, local variable

```
MethodName(
  AmbName(AmbName(Id("System")), Id("out"))
, Id("println")
)
```

Syntax Definition: How to Handle Ambiguities?

- Preserve ambiguities: parse forest (GLR)
- Generalize syntactic sorts: `PackageOrTypeName`, `AmbiguousName`, `ClassOrInterfaceType`

JLS: Mixture

```
ReferenceType:
    ClassOrInterfaceType
    TypeVariable

ClassOrInterfaceType:
    ClassType
    InterfaceType

PackageOrTypeName
    Identifier
    PackageOrTypeName . Identifier

ExpressionName:
    Identifier
    AmbiguousName . Identifier
```

Syntax Definition: Context-Dependent Ambiguities

Java Front

- Non-ambiguous (ambiguities encoded in grammar)
- e.g. `PackageOrTypeName`, `AmbName`, `ClassOrInterfaceType`

Alternative: Preserve

- Use Generalized LR and parse forest (a la Transformers)
- Declarative syntax definition
- Performance? Embedding?
- Would like to experiment with this: already have disambiguating type checker.

Alternative parsing technology resulted in various fixes in the JLS itself!

Parser and Pretty-Printer

`parse-java` and `pp-java`

- Supports comment preservation (e.g. Javadoc)

`parse-java`

- Supports preserving position information

`pp-java`

- Hand-crafted in Stratego/Box
  - Good-case and worst-case formatting
  - Full Stratego pattern-matching
- Preserves priorities
  - Inserts parentheses where necessary
  - generated from SDF syntax definition

Functional tests: roundtrip

- GNU Classpath and J2SDK
- `ast = parse; ast' = parse | pp | parse; compare`
- Found various bugs in syntax definition and pretty-printer

Unit tests: parse unit

```
test always take longest match for --
  "1--2" fails

test multiplication has higher priority than addition
  "1 + 2 * 3" -> Plus(_, Mul(_, _))

test Cast operators 1
  "(int) -1" -> CastPrim(_, Minus(_))

test Cast operators 8
  "(Integer) -1 " -> Minus(ExprName(Id("Integer")),Lit(Deci("1")))
```

Embedding of Java

Concrete syntax for Java in meta language

- Stratego-Java-15

```
Explode(r) :
  bstm |[ for(lvdec; e1; e*) stm ]| -> |[
       ForStatement x1 = _ast.newForStatement();
       x1.setBody(e2);
       x1.setExpression(e3);
       x1.initializers().add(~e: <r> lvdec);
       java.util.List x2 = x1.updaters();
       bstm2*
    ]| where ...

insert-admin(|x_matcher, x_admin) :
  bstm |[ if( x_matcher.lookingAt() ) {
             bstm_inner*
           } else { bstm* }
       ]| ->
  bstm |[ if( x_matcher.lookingAt() ) {
             x_admin = x_matcher.end();
             bstm_inner*
           } else { bstm* }
       ]|
```

Embedding of Java

- Generic embedding: parameterized with expression sort of meta language
- Explicit disambiguation

```
module Embedded-Java-15[E]
imports Java-15-Prefixed
exports
  variables
    [ij]  [0-9]*  -> JavaDeciLiteral {prefer}
    [xyz] [0-9]*  -> JavaID          {prefer}
    "e"   [0-9]*  -> JavaExpr        {prefer}

  context-free syntax
            "|[" JavaBlockStm  "]|" -> E {cons("ToMetaExpr")}
    "bstm"  "|[" JavaBlockStm  "]|" -> E {cons("ToMetaExpr")}
    "bstm*" "|[" JavaBlockStm* "]|" -> E {cons("ToMetaListExpr")}

    "~"  E ->  JavaExpr         {cons("FromMetaExpr")}
    "~*" E -> {JavaExpr ","}*   {cons("FromMetaExpr")}
```

Dryad, The Tree Nymph

Beyond support for Java syntax

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

Invaluable for the implementation of a Java transformation system!

Java Bytecode ↔ ATerm Bridge

- Access to bytecode is important for semantic analysis
  - Disambiguation and type-checking (used extensively)
- Bytecode represented in ATerm
  - Follows structure of the JVM Specification
  - Signature: `dryad/bytecode/signature`
- Disassembler: `class2aterm`
  - Generics Signatures
  - Local variables tables, line numbers, etc.
  - Code optional (`-c`)
- Assembler: `aterm2class`
- Implemented in Java, based on Apache's BCEL and Java ATerm Library

Java Bytecode ↔ ATerm Bridge

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Foo implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    String msg = "Don't push me!";
    System.out.println(msg);
  }
}
```

Java Bytecode ↔ ATerm Bridge

```
ClassFile(
  MinorVersion(0), MajorVersion(49)
, AccessFlags([Super, Public])
, ThisClass("Foo")
, SuperClass(Some("java.lang.Object"))
, Interfaces(["java.awt.event.ActionListener"])
, Fields([])
, Methods([
    Method(
      AccessFlags([Public])
    , Name("<init>")
    , MethodDescriptor([], Void)
    , Attributes([])
    )
  , Method(
      AccessFlags([Public])
    , Name("actionPerformed")
    , MethodDescriptor([ObjectType("java.awt.event.ActionEvent")], Void)
    , Attributes([])
    )
  ])
, Attributes([SourceFile("Foo.java")])
)
```

Java Bytecode ↔ ATerm Bridge

```
Code(
  MaxStack(Some(2))
, MaxLocals(Some(3))
, Instructions([
     LDC(String("Don't push me!"))
   , ASTORE(2)
   , GETSTATIC(
       FieldRef(Class("java.lang.System"), Name("out")
       , FieldDescriptor(ObjectType("java.io.PrintStream")))
     )
  , ALOAD(2)
  , INVOKEVIRTUAL(
      MethodRef(
        Class("java.io.PrintStream"), Name("println")
      , MethodDescriptor([ObjectType("java.lang.String")], Void)
      )
    )
  , RETURN
  ])
, ExceptionTable([])
, Attributes([])
)
```

Dryad Library

Dryad Model

- `dryad/model/-`: representation of source and bytecode
- Object oriented
- Global structure linked: `get-superclass`, etc.
- `get-methods`, `get-fields`, `get-formal-parameter-types`, ...
- Abstraction over source code or bytecode

JLS definitions

- Based on model
- Conversions and types
- For example: `is-assignment-convertable(|`*t*`)`, `supertypes`, `is-subtype(|`*type*`)`
- `<proper-supertypes> Int() => [Long, Float, Double]`
- Many of these definitions are non-trivial

Dryad: Reclassification and Qualification

`dryad-front (dryad-reclassify-ambnames)`

- Reclassification (names, types)
  - Contextually dependendent names
- Qualification (types)
  - Unqualified names are hard too handle
  - Use fully qualified names in transformations

Complex

- Imports, on demand imports, static imports
- Inner classes, non-trivial rules for visibility and shadowing
- Complex scoping rules
  - Even bugs in Sun's Java compiler
  - ...and the JLS
- Transformation should not be bothered with this

Implementation

- Strategies and scoped dynamic rules

Dryad R&Q: TypeName versus PackageName

Java Source

```
import java.util.ArrayList;
```

Parse

```
TypeImportDec(
  TypeName(
    PackageOrTypeName(
      PackageOrTypeName(Id("java")), Id("util")
    )
  , Id("ArrayList")
  ))
```

Reclassify

```
TypeImportDec(
  TypeName(
    PackageName([Id("java"), Id("util")])
  , Id("ArrayList")
  ))
```

Dryad R&Q: AmbName

Java Source

```
System.out.println("Hello World!");
```

Parse

```
MethodName(
  AmbName(AmbName(Id("System")), Id("out"))
, Id("println"))
```

Reclassify

```
MethodName(
  ExprName(
    TypeName(PackageName([Id("java"), Id("lang")])
    , Id("System"))
  , Id("out")
  )
, Id("println"))
```

Dryad R&Q: ClassType, InterfaceType and Qualification

Java Source

```
import java.util.List;

public class Foo {
  List getFoo() {};
}
```

Parse

```
MethodDecHead(...,
  ClassOrInterfaceType(TypeName(Id("List")), None)
  ... )
```

Reclassify

```
MethodDecHead(...,
  InterfaceType(
    TypeName(PackageName([Id("java"), Id("util")])
    , Id("List")), None)
  ...)
```

Dryad Type Checker

`dryad-front --tc on (dryad-type-checker)`

- Annotates expressions with their types . . . or not
- Type-aware transformations
  - e.g. extract method, inner class lifting
- Also available as a library
  - `dryad/type-check/-`
- Implementation: rewrite-rules and scoped dynamic rules

Status

- Basic operators and method resolution works
- But, not yet complete: no inner classes, no access modifiers
- No error reporting: only annotation
- `dryad-vis-tc-jtree`: show untyped expressions

Dryad Type Checker

System.out.println("Hello World!")

```
Invoke(
  Method(
    MethodName(
      ExprName(
        TypeName(PackageName([Id("java"), Id("lang")]) , Id("System"))
      , Id("out")
      ){ ClassType(
           TypeName(PackageName([Id("java"), Id("io")])
           , Id("PrintStream"))
         , None
         )
       }
    , Id("println")
    )
  )
, [ Lit(String([Chars("Hello World!")])){
      ClassType(TypeName(PackageName([Id("java"), Id("lang")]),
        Id("String")), None)}
  ]
){Void}
```

Application: JavaJava

Embed Java syntax in Java

```
context-free syntax
  "type" "|[" Type "]|" -> MetaExpr {cons("ToMetaExpr")}

variables
  "e" [0-9]*      -> Expr          {prefer}
  "e" [0-9]* "*" -> {Expr ","}*   {prefer}
```

Assimilation rules for Eclipse JDT Core API

```
  Assimilate(r) :
    type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

  Assimilate(r) :
    |[ y(e*) ]| -> |[
      {| MethodInvocation x = ast.newMethodInvocation();
         x.setName(ast.newSimpleName("~y"));
         bstm* | x |}
    ]|
    where <newname> "inv" => x
        ; <ExplodeArgs(r | x)> e* => bstm*
```

JavaJava: Ambiguities

Problem: explicit disambiguation in JavaJava

- Indicate syntactic sort in (anti-)quotations
- Naming convention for variables

```
String x = "Foo";

CompilationUnit dec = compilation-unit |[
  public class x {
    public static void main(String[] args) {
      System.out.println("Hello world");
    }
  } ]|;
```

Solution: type-based disambiguation

- No explicit disambiguation: parse forest
- Ambiguities eliminated in extension of Dryad type checker

JavaJava: Ambiguous Embedding

```
context-free syntax
  "|[" CompUnit  "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" TypeDec   "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" BlockStm  "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" BlockStm* "]|" -> MetaExpr {cons("ToMetaExpr")}

context-free syntax
  "#[" MetaExpr "]"-> ID   {cons("FromMetaExpr")}
  "#[" MetaExpr "]"-> Expr {cons("FromMetaExpr")}

variables
  MetaVarID -> ID
  MetaVarID -> Expr
  MetaVarID -> {Expr ","}+

lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> MetaVarID
```

JavaJava: Disambiguation

```
Assign(ExprName(Id("dec")),
       1> ToMetaExpr(  CompUnit(... ClassDec(... Id("Foo")...) ...)  )
       2> ToMetaExpr(  ClassDec(... Id("Foo") ...)   )
       3> ToMetaExpr([ ClassDec(... Id("Foo") ...) ] ) )

1> {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
      TypeDeclaration class_0 = _ast.newTypeDeclaration();
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 |}
2> {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
      class_1.setName(_ast.newSimpleName("Foo")); ... |class_1 |}
3> {| List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();
      decs_0.add( ... ); ... | decs_0 |}
```

Ambiguity Lifting

```
dec = 1> CompUnit 2> TypeDec 3> List<BodyDec>

1> dec = CompUnit 2> dec = TypeDec 3> dec = List<BodyDec>

 f(1> CompUnit 2> TypeDec 3> List<BodyDec>)

 1> f(CompUnit) 2> f(TypeDec) 3> f(List<BodyDec>)
```

Conclusion

- Working solid support for implementing Java transformations systems
- Some components are finished, some are work in progress

What's next?

- Full type checker
- Extensibility of components
- High-level transformation for Java
- JVM Bytecode back-end

See Also

- `http://dryad.stratego.org` – Dryad website
- `http://planet.stratego.org` – Weblog Karl, Rob & Martin

Planet Stratego/XT

Stratego

- Language for program transformation
- Suitable for implementing complete programs

XT

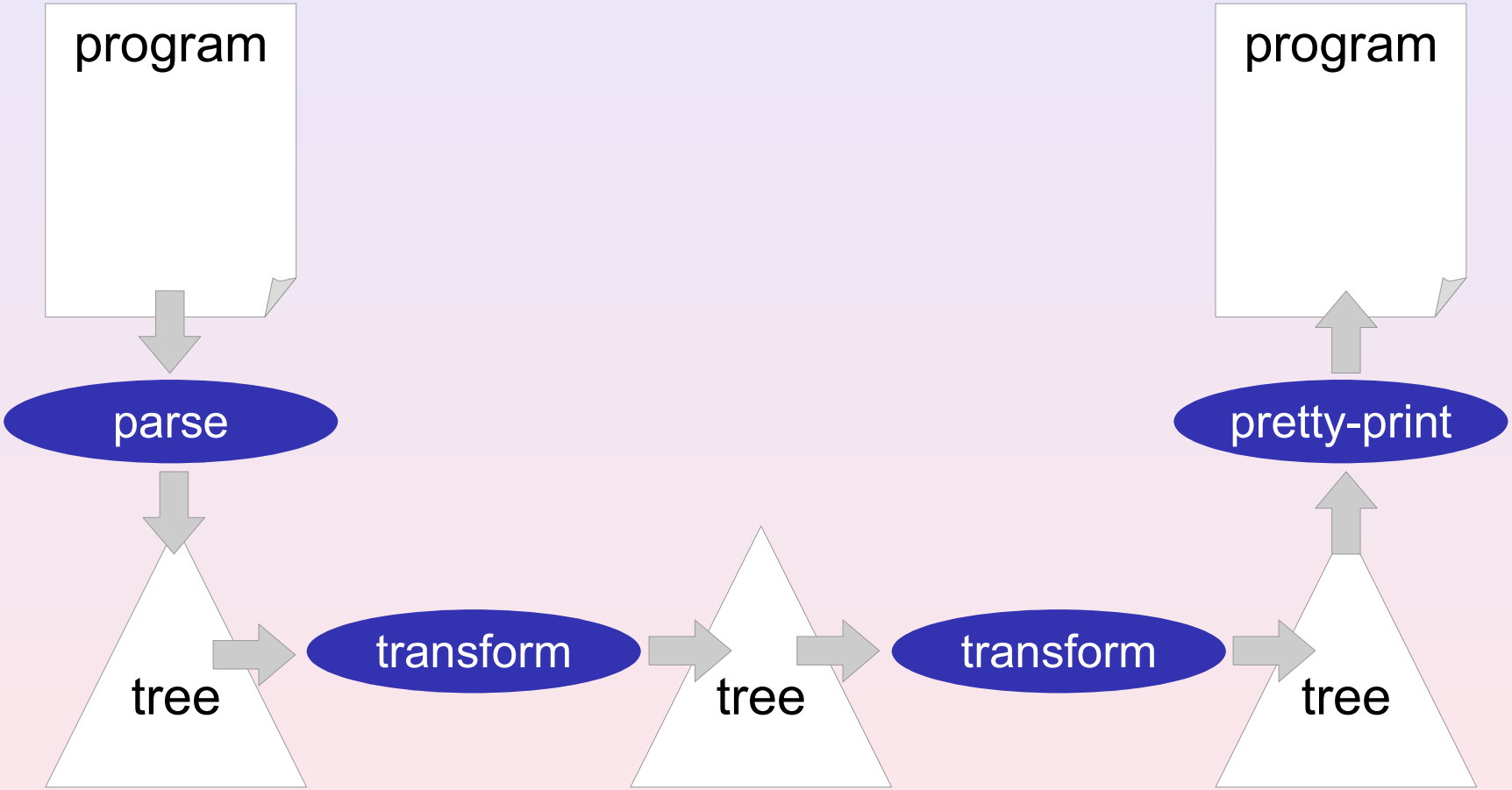
- Collection of Transformation (X) Tools
- Infrastructure for implementing transformation systems
- Parsing, pretty-printing, interoperability

XT Orbit

- Language specific tools
- Java, C, C++, Octave, ...

This lecture: the XT of Stratego/XT

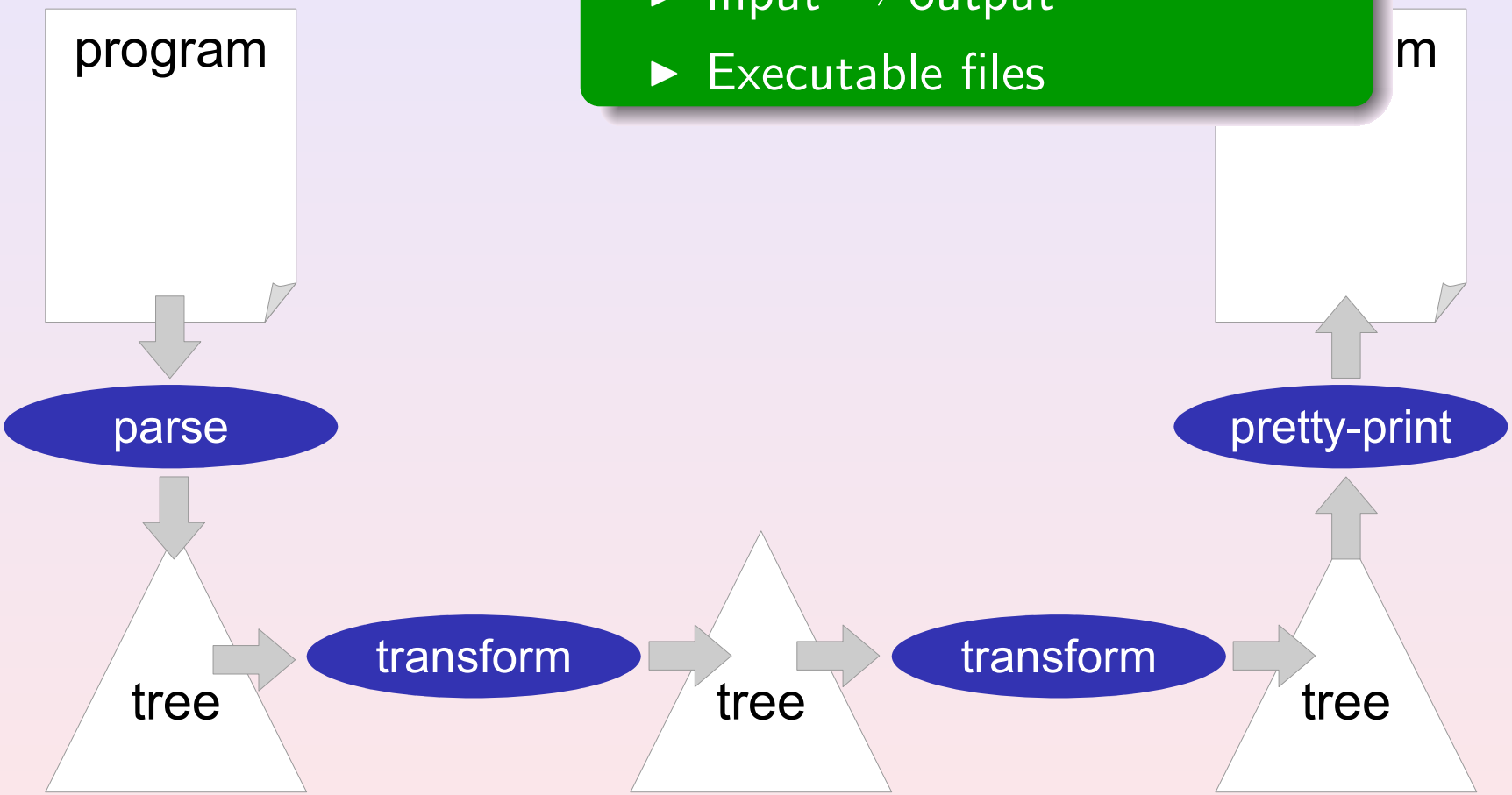
Program Transformation Pipeline



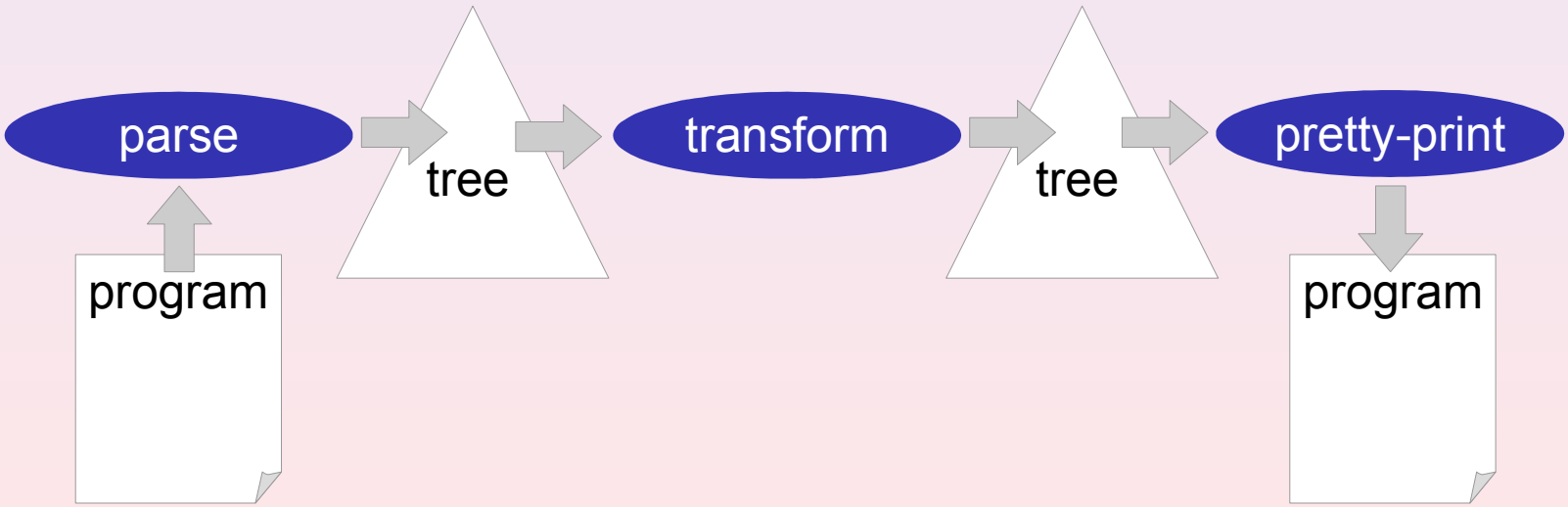
Program Transformation Pipeline

Transformation tools

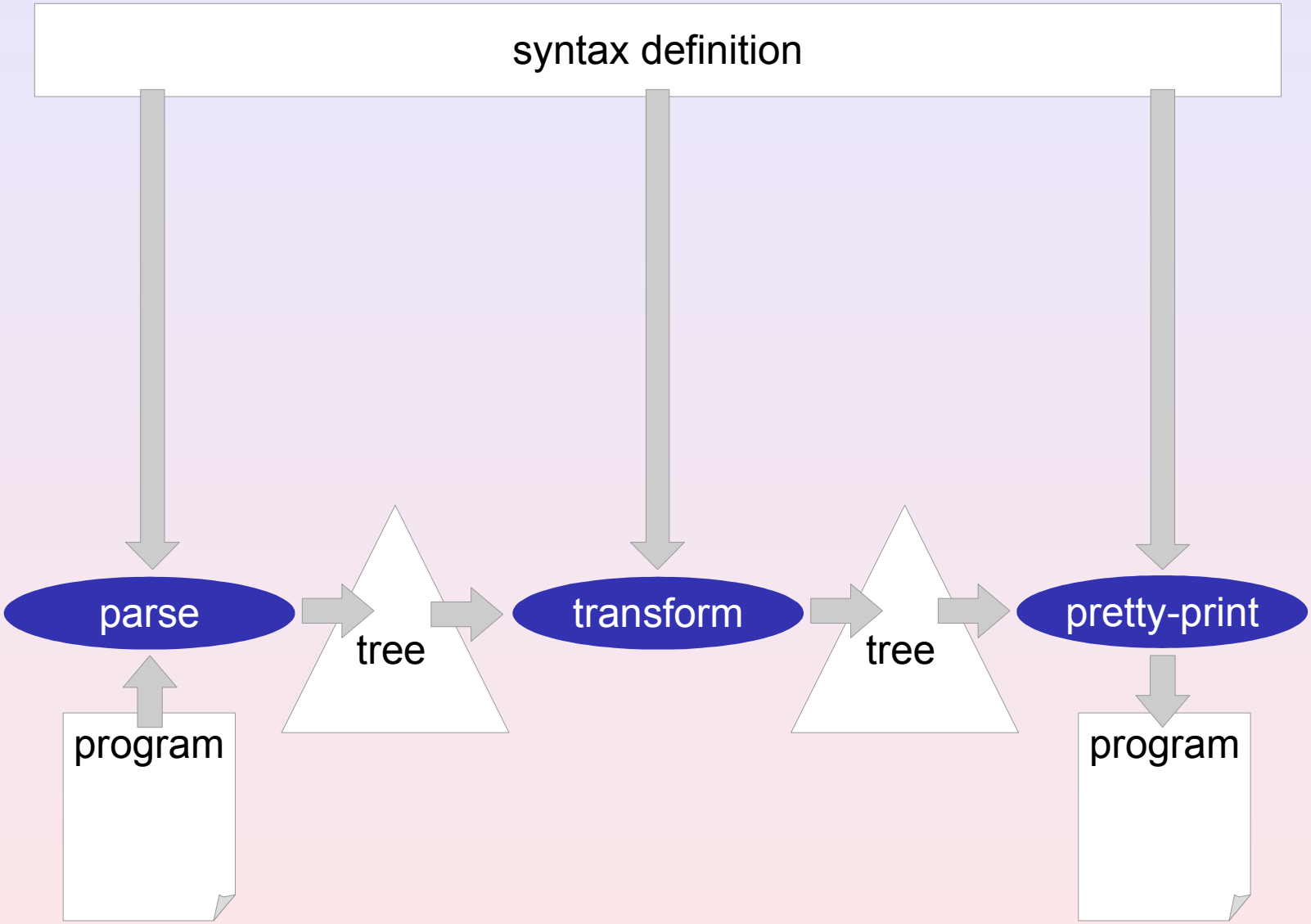
- Input → output
- Executable files



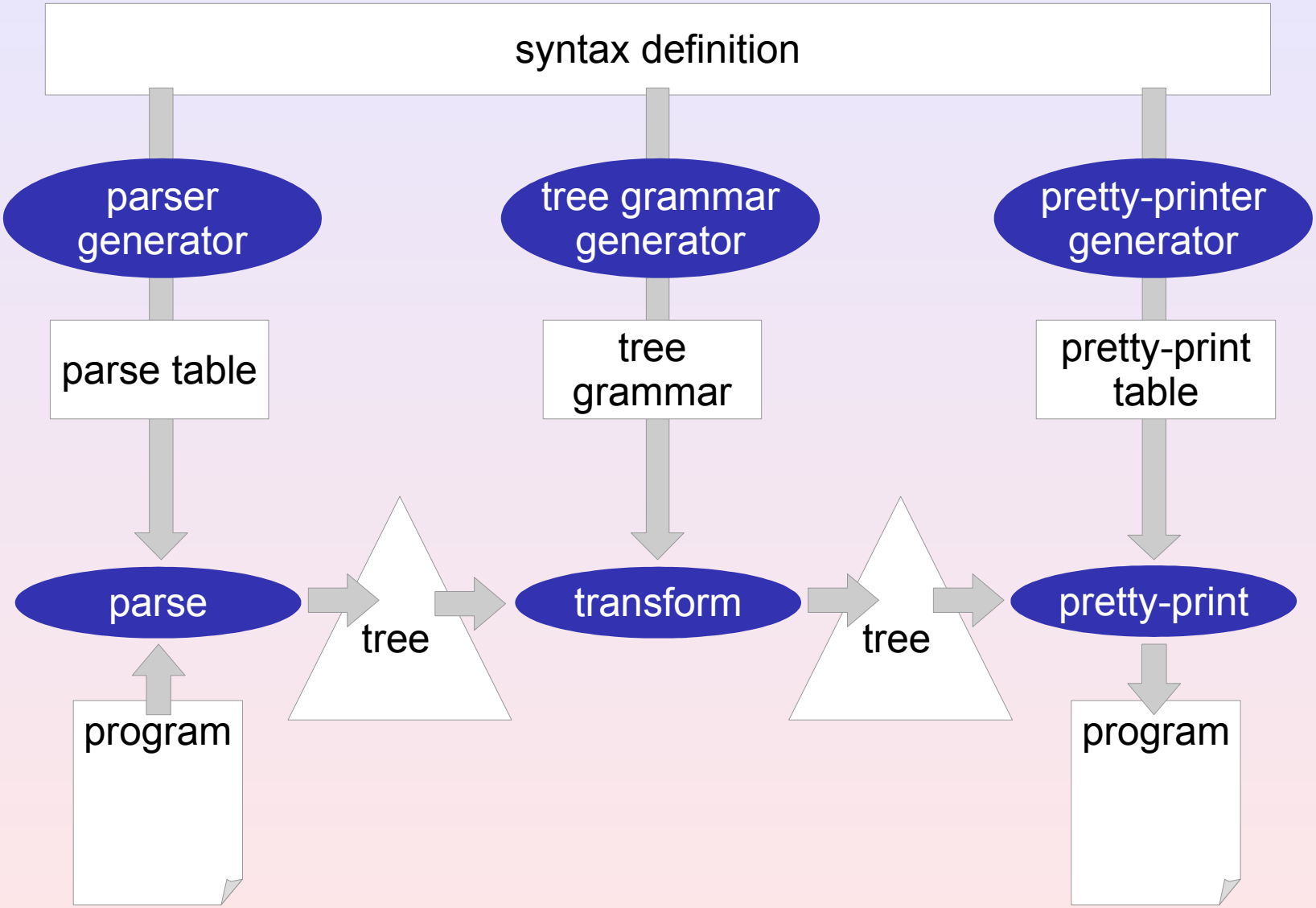
Architecture of Stratego/XT



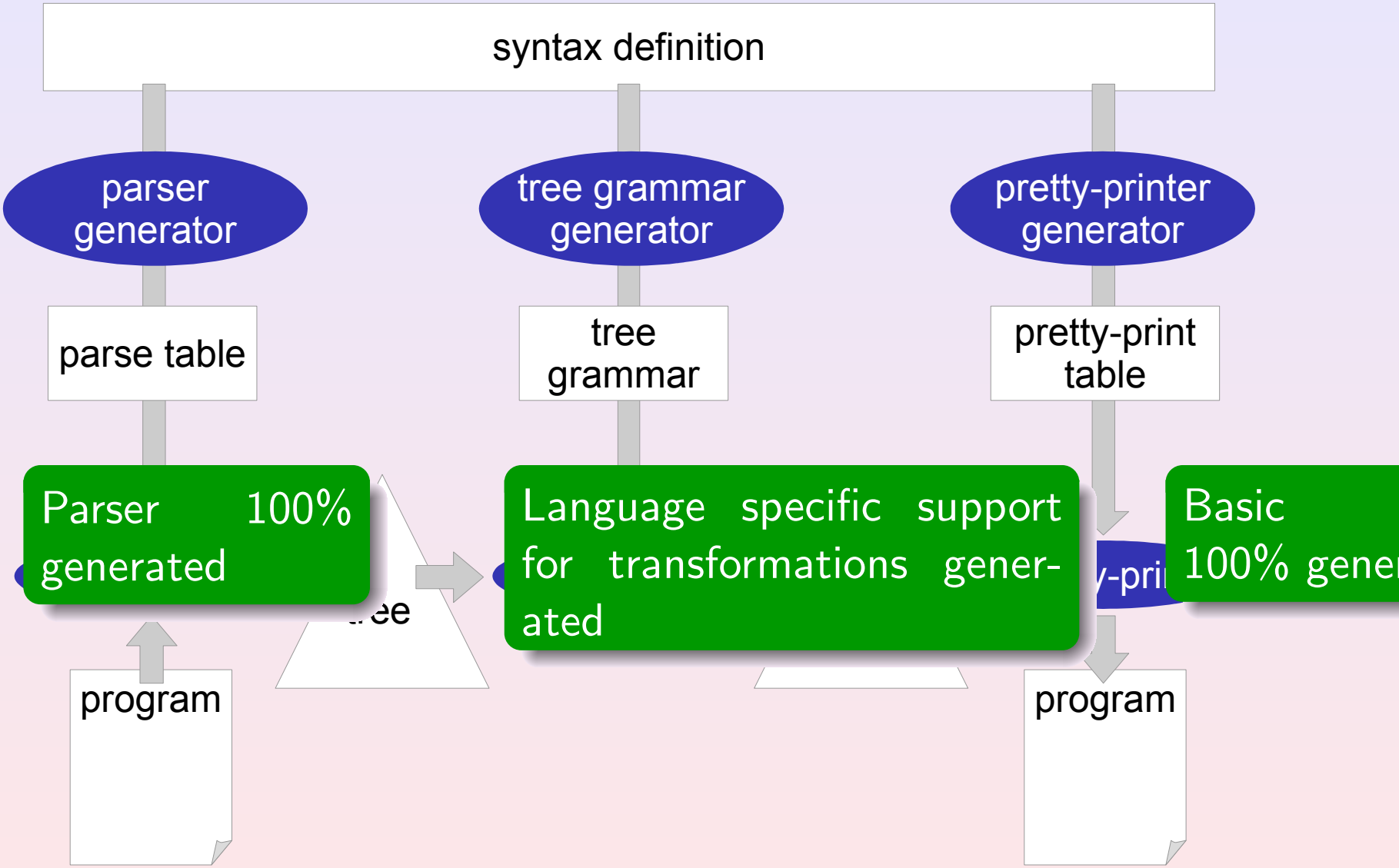
Architecture of Stratego/XT



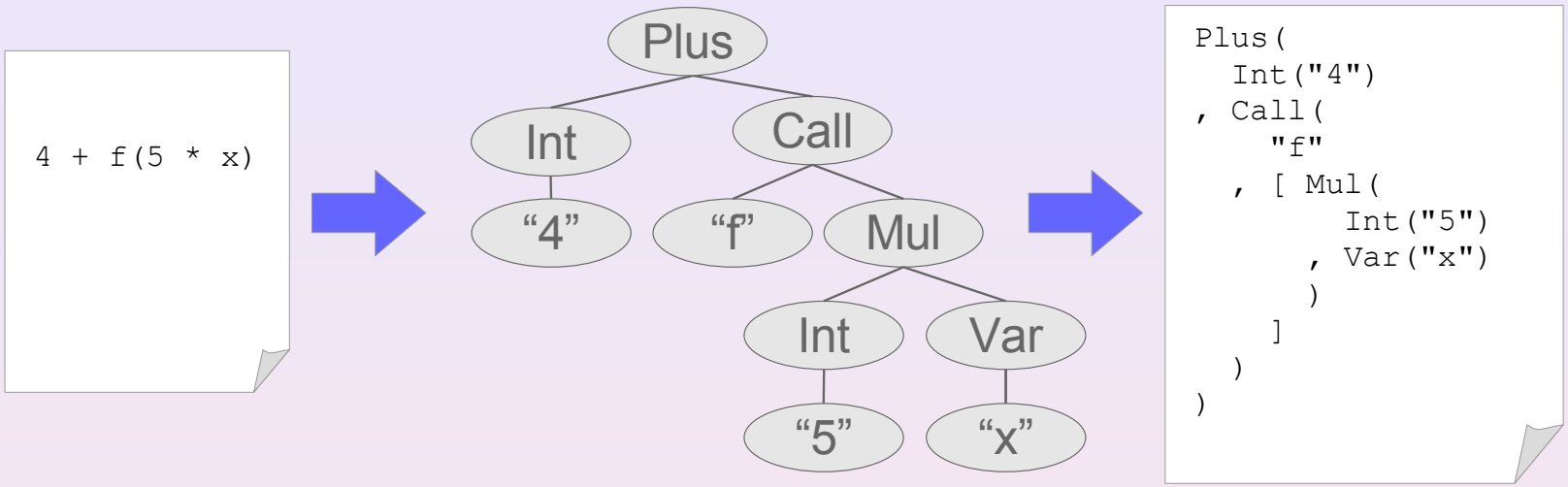
Architecture of Stratego/XT



Architecture of Stratego/XT



Tree Representation



Trees are represented as terms in the ATerm format

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

ATerm Format

Application	`Void()`, `Call(`t`, `t`)`
List	`[]`, `[`t`, `t`, `t`]`
Tuple	`(`t`, `t`)`, `(`t`, `t`, `t`)`
Integer	`25`
Real	`38.87`
String	`"Hello world"`
Annotated term	t{t, t, t}

- Exchange of structured data
- Efficiency through maximal sharing
- Binary encoding

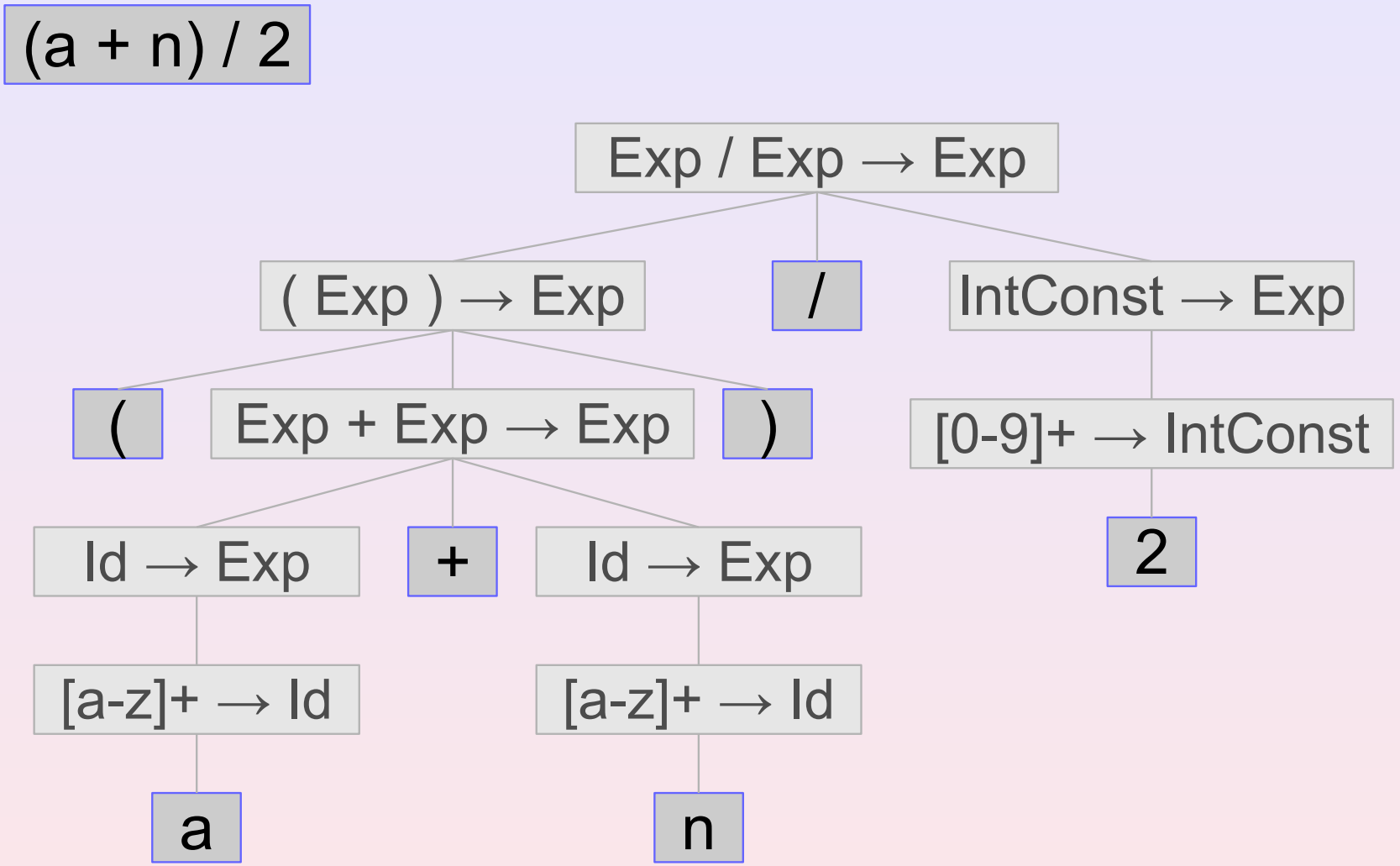
Structured Data: comparable to XML
Stratego: internal is external representation

Context-Free Grammars

Simple Expression Language

Id	$\longrightarrow$	[a-z]+
IntConst	$\longrightarrow$	[0-9]+
Exp	$\longrightarrow$	*Id*
		IntConst
		Exp + *Exp*
		Exp - *Exp*
		Exp * *Exp*
		Exp / *Exp*
		(*Exp*)

Parse Trees


(a + n) / 2
Exp / Exp → Exp
(Exp) → Exp
/
IntConst → Exp
(
Exp + Exp → Exp
)
[0-9]+ → IntConst
Id → Exp
+
Id → Exp
2
[a-z]+ → Id
[a-z]+ → Id
a
n

Parse Trees and Abstract Syntax Trees

Text → Parse Tree → Abstract Syntax Tree

AsFix: ATerm language for parse trees

- Describes Applications of productions
- All characters of the input
  - Even whitespace and comments!
- Yield parse tree to text

```
$ asfix-yield -i exp.asfix
(a + n) / 2
```

Abstract Syntax Trees

- Remove literals, whitespace, comments

```
$ implode-asfix -i exp.asfix
Div(Plus(Var("a"), Var("n")), Int("2")
```

Generic Tools for ATerms

Pretty-print an ATerm in a nice layout

```
$ pp-aterm -i foo.aterm
```

Usually applied at the end of a pipeline:

```
$ echo "foo([0], bar(1, 2), fred(3,4))" | ... | pp-aterm
foo(
  [0]
, bar(1, 2)
, fred(3, 4)
)
```

Ambiguity in Context-Free Grammars

- **e1 + e2 * e3**
  - `(e1 + e2) * e3`
  - `e1 + (e2 * e3)`
- **e1 + e2 + e3**
  - `(e1 + e2) + e3`
  - `e1 + (e2 + e3)`
- **++a**
  - `+(+a)`
  - `++ a?`
- **null**
  - Keyword or identifier?
- **if e1 then if e2 then e3 else e4**
  - `if e1 then (if e2 then e3) else e4`
  - `if e1 then (if e2 then e3 else e4)`

SDF – Syntax Definition Formalism

1. **Declarative**
   - Important for code generation
   - Completely define the syntax of a language
2. **Modular**
   - Syntax definitions can be composed!
3. **Context-free and lexical syntax**
   - No separate specification of tokens for scanner
4. **Declarative disambiguation**
   - Priorities, associativity, follow restrictions
5. **All context-free grammars**
   - Beyond LALR, LR, LL

SDF: Modular

```
module Lexical
exports
  lexical syntax
    ...
```

```
module Expressions
imports Lexical
exports
  context-free syntax
    ...
```

```
module Main
imports Expressions
exports
  context-free start-symbols Exp
```

SDF: Parser Generation

Modules and Definitions

- SDF Module (`.sdf`)
- SDF Definition (`.def`)

Generating a parser

- Collect SDF modules into a single syntax definition

```
$ pack-sdf -i Example.sdf -o Example.def
```

- Generate a parse-table

```
$ sdf2table -i Example.def -o Example.tbl -m Main
```

- Parse an input file

```
$ sglri -i foo.exp -p Example.tbl
```

- Parse an input file (alternative)

```
$ sglr -2 -i foo.exp -p Example.tbl | implode-asfix
```

SDF: Lexical Syntax

Lexical syntax is defined with ordinary productions.

```
module Lexical
exports
  sorts Id IntConst BoolConst
  lexical syntax
    [A-Za-z][A-Za-z0-9]* -> Id

    [0-9]+  -> IntConst
    "true"  -> BoolConst
    "false" -> BoolConst

    [\r\n\t\ ] -> LAYOUT
    "//" ~[\n]* [\n] -> LAYOUT
```

- Even *context-free* lexical syntax is possible
- Avoid complex regular expressions

SDF: Disambiguation of Lexical Syntax

Declaring reserved keywords: reject certain productions

```
lexical syntax
  "true"  -> Id {reject}
  "false" -> Id {reject}
```

Longest match: follow restriction

```
lexical restrictions
  Id       -/- [A-Za-z0-9]
  IntConst -/- [0-9]
```

Require layout after a keyword

```
lexical restrictions
  "if" -/- [A-Za-z0-9]
```

SDF: Disambiguation of Lexical Sy

```
amb([Bool("true"),Var("true")]
```

Declaring reserved keywords: reject certain productions

```
lexical syntax
  "true"  -> Id {reject}
  "false" -> Id {reject}
```

Longest match: follow restriction

```
lexical restrictions
  Id       -/- [A-Za-z0-9]
  IntConst -/- [0-9]
```

Require layout after a keyword

```
lexical restrictions
  "if" -/- [A-Za-z0-9]
```

SDF: Disambiguation of Lexical Syntax

Declaring reserved keywords: reject certain productions

```
lexical syntax
  "true"  -> Id {reject}
  "false" -> Id {reject}
```

Rejects unintended split of iden

```
$ echo "xinstanceof Foo" | sgl
InstanceOf(Var("x"),"Foo")
```

Longest match: follow restriction

```
lexical restrictions
  Id       -/- [A-Za-z0-9]
  IntConst -/- [0-9]
```

Require layout after a keyword

```
lexical restrictions
  "if" -/- [A-Za-z0-9]
```

SDF: Disambiguation of Lexical Syntax

Declaring reserved keywords: reject certain productions

```
lexical syntax
  "true"  -> Id {reject}
  "false" -> Id {reject}
```

Longest match: follow restriction

```
lexical restrictions
  Id       -/- [A-Za-z0-9]
  IntConst -/- [0-9]
```

Require layout after a keyword

```
lexical restrictions
  "if" -/- [A-Za-z0-9]
```

Rejects unintended split of keyw

```
$ echo "ifx then y" | sglri
IfThen(Var("x"),Var("y"))
```

SDF: Context-free Syntax

```
context-free syntax
  Id              -> Exp {cons("Var")}
  IntConst        -> Exp {cons("Int")}
  BoolConst       -> Exp {cons("Bool")}

  "(" Exp ")"     -> Exp {bracket}

  Exp "+" Exp     -> Exp {cons("Plus")}
  Exp "-" Exp     -> Exp {cons("Min")}
  Exp "*" Exp     -> Exp {cons("Mul")}
  Exp "/" Exp     -> Exp {cons("Div")}

  Exp "&" Exp     -> Exp {cons("And")}
  Exp "|" Exp     -> Exp {cons("Or")}
  "!" Exp         -> Exp {cons("Not")}

  Id "(" {Exp ","}* ")" -> Exp {cons("Call")}
```

SDF: Associativity of Operators

```
$ echo "1 + 2 + 3" | sglri -p Example.tbl
amb([
  Plus(Plus(Int("1"), Int("2")), Int("3"))
, Plus(Int("1"), Plus(Int("2"), Int("3")))
])
```

Declare associativity in attribute:

```
    Exp "+" Exp -> Exp {left, cons("Plus")}
    Exp ">" Exp -> Exp {non-assoc, cons("Gt")}
```

- left
- right
- assoc
- non-assoc

SDF: Priority of Operators

```
$ echo "1 + 2 * 3" | sglri -p Example.tbl
amb([
  Mul(Plus(Int("1"), Int("2")), Int("3"))
, Plus(Int("1"), Mul(Int("2"), Int("3")))
])
```

```
context-free priorities
      "!"  Exp -> Exp
  > {
      Exp "*" Exp -> Exp
      Exp "/" Exp -> Exp
    }
  > {
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp
    }
  >   Exp "&" Exp -> Exp
  >   Exp "|" Exp -> Exp
```

SDF: Associativity of Operators in Group

```
$ echo "1 + 2 - 3" | sglri -p Example.tbl
amb([
  Min(Plus(Int("1"),Int("2")),Int("3"))
, Plus(Int("1"),Min(Int("2"),Int("3")))
])
```

```
context-free priorities
      "!"  Exp -> Exp
  > {left:
      Exp "*" Exp -> Exp
      Exp "/" Exp -> Exp
    }
  > {left:
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp
    }
  >   Exp "&" Exp -> Exp
  >   Exp "|" Exp -> Exp
```

Parse-unit: Testing SDF Syntax Definitions

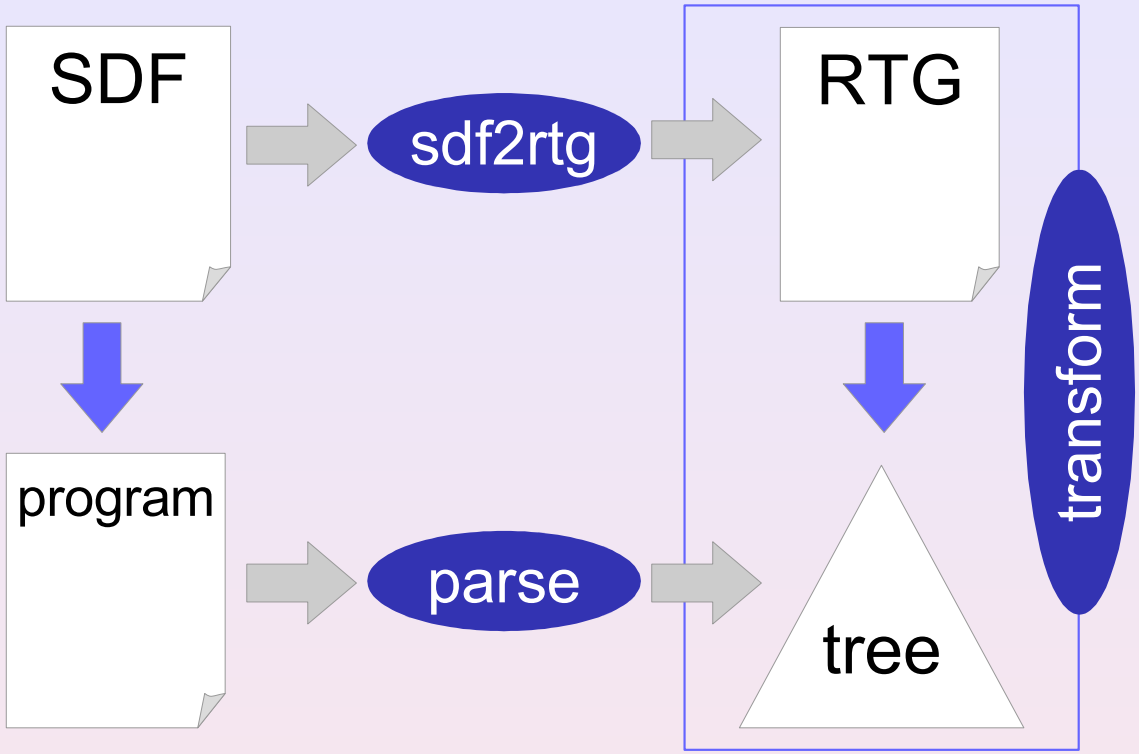
```
testsuite Expressions
topsort Exp

test simple addition
  "2 + 3" -> Plus(Int("2"), Int("3"))

test addition is left associative
  "1 + 2 + 3" -> Plus(Plus(_, _), _)

test > is not associative
  "1 > 2 > 3" fails

test
  file foo.exp succeeds
```

```
$ parse-unit -i exp.testsuite -p Example.tbl
...
```

SDF: Parsing Technology

- SDF requires an extraordinary general parsing algorithm.
- SDF relies on SGLR parsing
- Scannerless: no separate lexical analysis
  - Every character is a token
  - Context-dependent lexical syntax
- Generalized LR: allows ambiguities
  - All derivations
  - Produces a parse forest
  - Technique: fork LR parsers
- Advantage: declarative syntax definition
  - Excellent for code generation

Tree Grammars as Contracts



- Syntax definitions (grammars) define a set of strings
- Transformation tools operate on trees
- Tree grammars define the format of trees
- Compare to DTD, W3C XML Schema, RELAX NG

Regular Tree Grammars

```
regular tree grammar
  start Exp
  productions
    Exp -> Int(IntConst)
         | Bool(BoolConst)
         | Not(Exp)
         | Mul(Exp, Exp)
         | Plus(Exp, Exp)
         | Call(Id, Exps)

    Exps -> <nil>()
          | <cons>(Exp, Exps)

    BoolConst -> <string>
    IntConst  -> <string>
    Id        -> <string>
```

Tools for Regular Tree Grammars

- Derive from SDF syntax definition

```
$ sdf2rtg -i Example.def -m Example -o Example.rtg
```

- Check the format of a tree

```
$ format-check --rtg Example.rtg
```

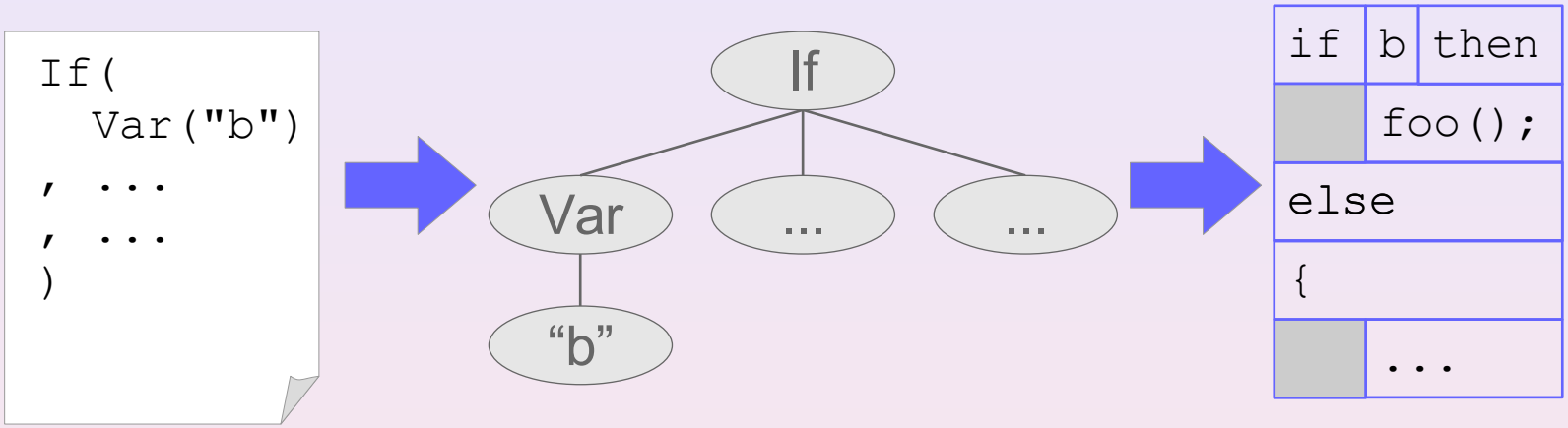
```
martin@logistico:~> format-check --rtg Exp.rtg -i exp3.trm --vis
error: cannot type Int(1)
    inferred types of subterms:
    typed 1 as <int>
error: cannot type Div(1,Var("c"))
    inferred types of subterms:
    typed 1 as <int>
    typed Var("c") as Exp
Plus(
  Mul(Int(1), Var("a"))
, Minus(Var("b"), Div(1, Var("c")))
)
martin@logistico:~>
```

- Generate tools and libraries

```
$ rtg2sig -i Example.rtg -o Example.str
```

Pretty-printing

Code generators and source to source transformation systems need support for pretty-printing.

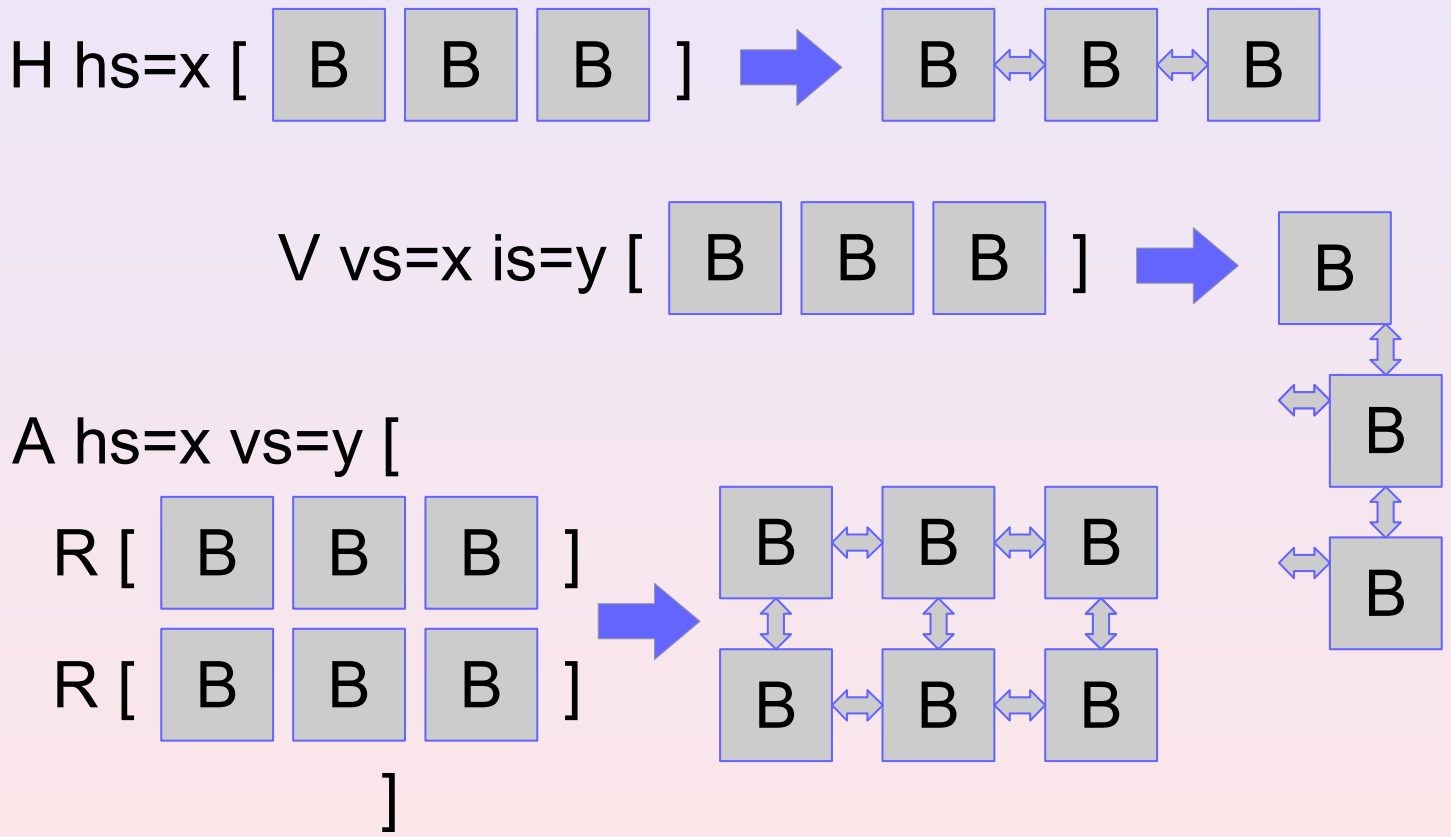


Stratego/XT: GPP (Generic Pretty-Printing)

- Box language
- Pretty-printer generation
- Different back-ends: `abox2text`, `abox2html`, `abox2latex`

Box Language

- Text formatting language
- Options for spacing, indenting
- 'CSS for plain text'



Other boxes: `HV`, `ALT`, `KW`, `VAR`, `NUM`, `C`

Example Box

```
V is=2 [
  H [KW["while"] "a" KW["do"]]
  V [
    V is=2 [
      H hs=1 [KW["if"] "b" KW["then"]]
      H hs=0 ["foo()" ";"]
    ]
    KW["else"]
    V [V is=2 ["{" "..."] "}"]
  ]
]
```

```
while a do
  if b then
    foo();
  else
  {
    ...
  }
```

Pretty-print Tables

- List of pretty-print rules
- Applied by constructor name (`cons` attribute)

Example Pretty-Print Table

```
[
   Var  -- _1,
   Bool -- _1,
   Int  -- _1,
   Mul  -- _1 KW["*"] _2,
   Plus -- _1 KW["+"] _2,
   Min  -- _1 KW["-"] _2,
   Call -- _1 KW["("] _2 KW[")"],
   Call.2:iter-star-sep -- _1 KW[","]
]
```

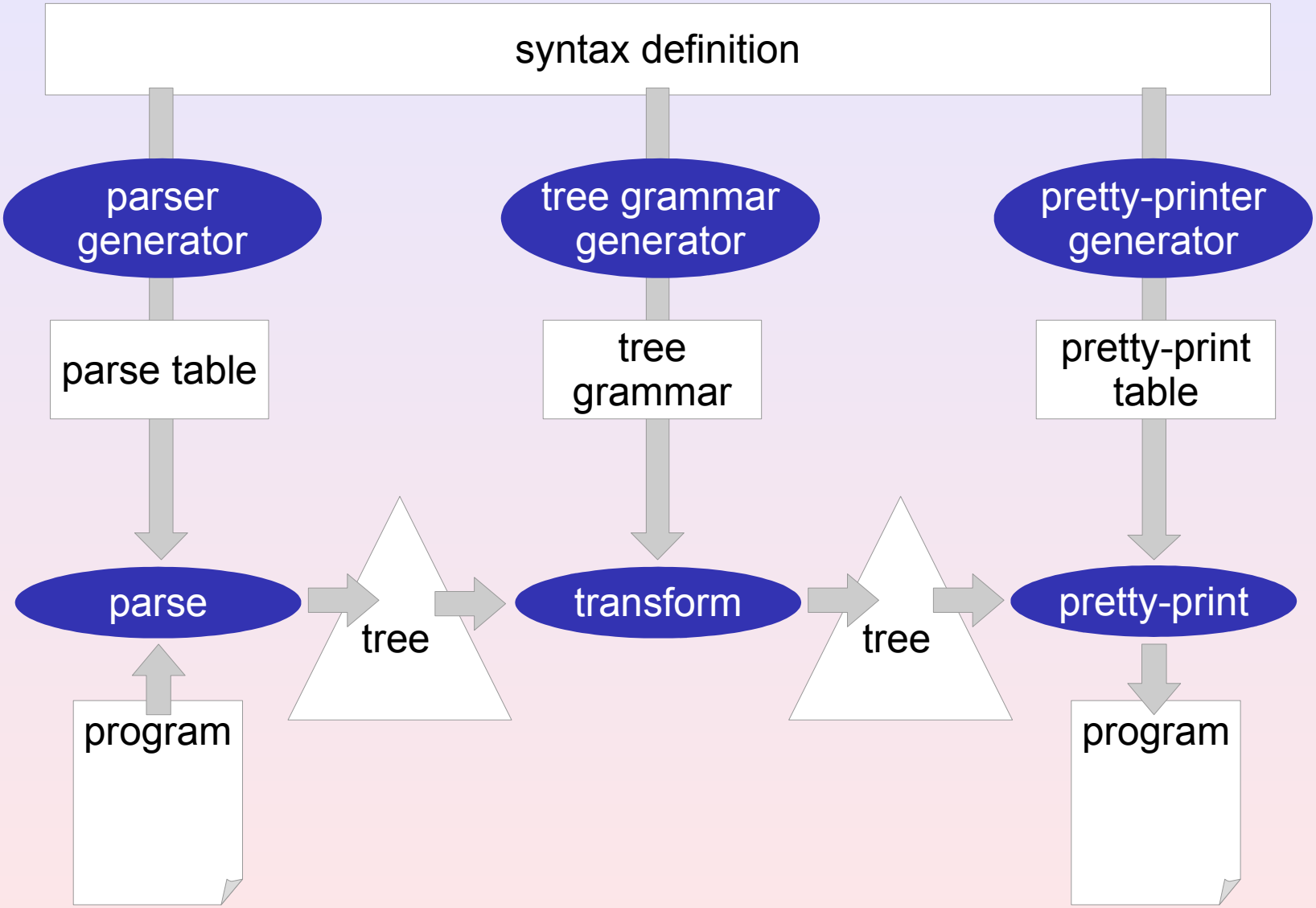
- `ast2abox` accepts sequence of pretty-print tables
- Tables can be combined and reused

```
$ echo "1 + 2" | sglri -p Ex.tbl | ast2abox -p Ex.pp | abox2text
```

Pretty-printer Generation

- Pretty-print table can be generated from SDF syntax definition (`ppgen`)
  - Complete and correct (usually)
  - Minimal formatting
- Customization by hand for pretty result
  - Tools for consistency checking and patching (`pptable-diff`)
- Parentheses problem: parentheses inserter can be generated from SDF syntax definition (`sdf2parenthesize`).

Architecture of Stratego/XT



XT Orbit

- **Java**
  - High-quality syntax definition (1.5)
  - Handcrafted pretty-printer (1.5)
  - Disambiguation
  - Type-checker
- **C** (EPITA, France)
  - Syntax definition (C99)
  - Disambiguation
- **Octave**
  - Parser
  - Type-checker
  - Compiler
- **Prolog**
  - Syntax definition
  - Embedding of object languages
- **BibTeX**
  - Syntax definition
  - Web services

Meta-Programming with Concrete Object Syntax

- *Meta*-programming:
  - Generating, transforming, or analyzing *object* programs
- Meta-programs should use concrete syntax of object language
  - Implications for meta-language implementation
- General architecture for extending meta-languages
  - Modular syntax definition
  - Meta explosion and assimilation
  - By default available in Stratego/XT
- Running Example
  - Meta language: Stratego
  - Object language: Tiger
  - Meta program: instrumentation of Tiger program

Case Study: Instrumenting Programs

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

```
fact entry
   fact entry
      fact entry
         fact entry
            ...
         fact exit
      fact exit
   fact exit
fact exit
```

Meta-program *TraceAll*:

- Instrument a Tiger program to trace function calls
- *Generate* code for `enterfun` and `exitfun`
- *Transform* code for functions

Case Study: Instrumenting Programs

```
let var ind := 0
    function enterfun(name : string) =
      (ind := (ind + 1);
       for n := 2 to ind do print("   ");
       print(name);
       print("  entry"))
    function exitfun(name : string) =
      (for n := 2 to ind do print("   ");
       ind := (ind - 1);
       print(name);
       print("  exit "))
    function fact(n : int) : int =
      (enterfun(" fact");
       let var a_0 : int := nil
       in if (n < 1)
          then a_0 := 1
          else a_0 := n * fact(n - 1);
          exitfun(" fact");
          a_0
       end)
in printint(fact(10))
end
```

Generation with String-Based Concrete Syntax

```
IntroducePrinters :
  e -> "let var ind := 0"
    ++ "    function enterfun(name : string) ="
    ++ "      (ind := +(ind, 1);"
    ++ "       for i := 2 to ind do print(\" \");"
    ++ "       print(name); print(\" entry\"))"
    ++ "    function exitfun(name : string) = "
    ++ "      (for i := 2 to ind do print(\" \");"
    ++ "       ind := -(ind, 1);"
    ++ "       print(name); print(\" exit\"))"
    ++ " in "
    ++ e
    ++ "end"
```

- *Generation* with concrete syntax
- Readable code
- Easy to implement
- No syntactic checks
- *Transformation* requires analysis of program

Transformation of Abstract Syntax Trees

```
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
```

```
regular tree grammar
  productions
    Var  -> Var(Id)
    Exp  -> Call(Var, Exps)
          | Plus(Exp, Exp)
```

- Structured representation
- Works well with small language and small program fragments
- Analysis and generation of program fragments
- Tree structure: matching and building

Transformation of Abstract Syntax Trees

```
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(t), e) ->
    FunDec(f, xs, Tp(t),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(t),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
```

- Does not scale up to large program fragments
- Requires deep knowledge of syntactic structure
- Does not scale up to large languages (complex syntax)

Write Concrete & Transform Abstract!

```
rules
  TraceProcedure :
    |[ function f(x*) = e ]| ->
    |[ function f(x*) = (enterfun(s); e; exitfun(s)) ]|
    where !f => s

  TraceFunction :
    |[ function f(x*) : tid = e ]| ->
    |[ function f(x*) : tid =
         (enterfun(s);
          let var x : tid := e
           in exitfun(s); x end) ]|
    where new => x ; !f => s
```

- Can be used in transformation *and* generation
- Structured representation (transformation on trees)
- Fragments are syntactically correct
- Requires 'no' deep knowledge of syntactic structure
- Scales up to large languages (complex syntax)

Write Concrete & Transform Abstract!

```
rules
  IntroducePrinters :
    e -> |[ let var ind := 0

                function enterfun(name : string) =
                 (ind := +(ind, 1);
                  for i := 2 to ind do print(" ");
                  print(name); print(" entry"))

                function exitfun(name : string) =
                 (for i := 2 to ind do print(" ");
                  ind := -(ind, 1);
                  print(name); print(" exit"))

            in e end ]|
```

- Scales up to large program fragments

Java: Code Generation

Concrete object syntax is not a Stratego specific issue.

Suppose we want to generate:

```
if(propertyChangeListeners == null)
  return;

PropertyChangeEvent event =
  new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
  ((PropertyChangeListener)
      propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

Parameterized by the name of the listeners variable.

(Fragment generated by Castor)

Java: Code Generation using Strings

```
String vName = "propertyChangeListeners";

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Java: Code Generation using Strings

```
String vName = "propertyChangeListeners";

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Escaping to the meta difficult.

Code generator tries pretty printing.

Further processing of impossible.

Java: Code Generation using Abstract Syntax Trees

```
VariableDeclarationFragment fragment =
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

Java: Code Generation using Abstract Syntax Trees

Does not corr
structure of the
erated.

```
VariableDeclarationFragment fragment =
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"))
args.add(_ast.newSimpleName("oldValue"))
args.add(_ast.newSimpleName("newValue"))
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
                                    ment(fragment);
vard
    _ast.newSimpleName("PropertyChangeEvent")));
```

Code is syntactically checked
host language compiler and furt
processing is possible.

Don't worry about the layout.

Java: Code Generation using Concrete Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(x == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < x.size(); c++) {
    ((PropertyChangeListener)
      x.elementAt(c)).propertyChange(event);
  }
]|;
```

Java: Code Generation using Concrete Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(x == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < x.size(); c++) {
    ((PropertyChangeListener)
      x.elementAt(c)).propertyChange(event);
  }
]|;
```

Syntax of code is ch ther proces ble.

Separate pretty-printer: don't worry about the layout.

Support for interacti between the generat code and the meta la guage.

Implementation: Meta-Programming a la Concrete Syntax

Syntax of Stratego (meta language)
is extended with
syntax of Tiger (object language)

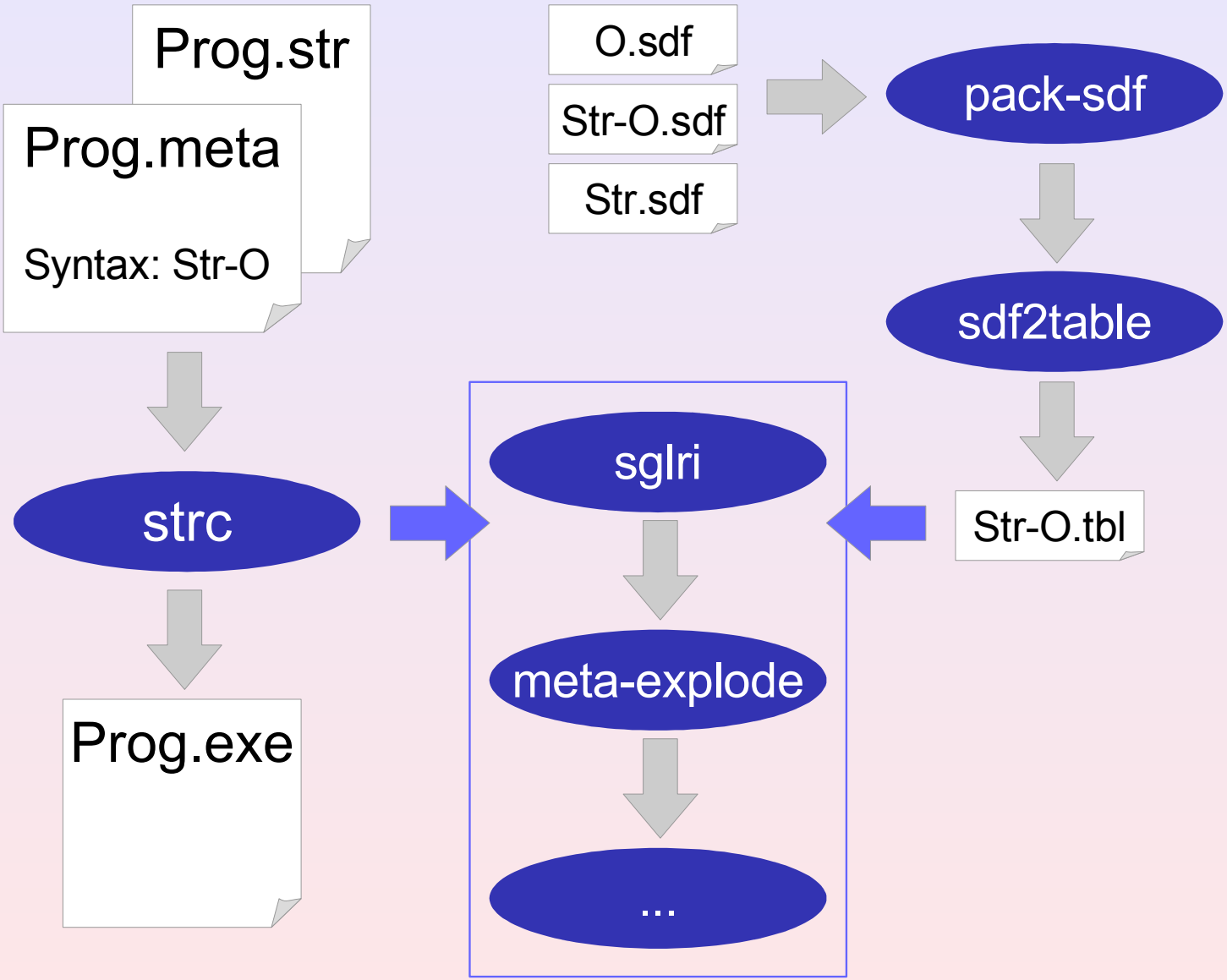
Recipe

1. Combine syntax of meta- and object-languages
2. Parse meta-program with combined syntax
3. Feed parse tree to meta-language compiler

Ingredients

- M-compile: meta-language compiler
- SDF: Modular syntax definition formalism
- SGLR: Scannerless Generalized LR parser

Architecture



How To Use In Stratego (Java)

- Usually, an embedding of `Obj` already exists.
- Use concrete syntax in `Foo.str`

```
module Foo imports Java-15 options
strategies
  main =
    output-wrap(
      !|[ package foo;
          public class Bar {
            public static void main(String[] ps) {
              System.out.println("Hello world!");
            }
          } ]|)
```

- Declare the syntax in `Foo.meta`

```
Meta([ Syntax("Stratego-Java-15") ])
```

- Compile the Stratego program

```
$ strc -i Foo.str -I ...
```

How To Use In Stratego (Tiger)

- Usually, an embedding of `Obj` already exists.
- Use concrete syntax in `Tiger-Profile.str`

```
module Tiger-Profile
imports libstrategolib Tiger

  ...

        InstrumentCall(s) :
          |[ f(a*) ]| -> |[ (z := z + 1; f(a1*)) ]|

  ...
```

- Declare the syntax in `Tiger-Profile.meta`

```
Meta([ Syntax("StrategoTiger") ])
```

- Compile the Stratego program

```
$ strc -i Tiger-Profile.str -I ...
```

Implementation: Syntax of Object Language (Tiger)

SDF Syntax Definition

```
module Tiger
exports
  context-free syntax
    "let" Dec* "in" {Exp ";"}* "end" -> Exp {cons("Let")}
    Id                               -> Var {cons("Var")}
    LValue ":=" Exp                  -> Exp {cons("Assign")}
    "(" {Exp ";"}* ")"               -> Exp {cons("Seq")}
```

Regular Tree Grammar

```
regular tree grammar
  productions
    Exp -> Seq(Exps)
         | Assign(LValue, Exp)
         | Let(Decs, Exps)
    Var -> Var(Id)
```

let *ds* in *x* := (*es*) end $\Rightarrow$ Let(*ds*,[Assign(Var(*x*),Seq(*es*))])

Implementation: Syntax of Meta Language (Stratego)

SDF Syntax Definition of Stratego

```
module Stratego
exports
  context-free syntax
    String                      -> Term {cons("Str")}
    Var                         -> Term {cons("Var")}
    Id "(" {Term ","}* ")"      -> Term {cons("Op")}
    Term "->" Term              -> Rule {cons("Rule")}
```

Stratego Fragment, Concrete Syntax

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

Stratego Fragment, Abstract Syntax

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                  Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
               Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                       Op("Seq",[Var("es")])]),
                          Op("Nil",[])])]))
```

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp           {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+    {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp           {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+    {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp           {cons("FromTerm"
    "~*" StrategoTerm -> {Exp ";"}+    {cons("FromTerm"

  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Anti-quotation: in
pressions into o
sions

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp           {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+    {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Declare meta-va
ject sorts

Meta Variables

```
TraceProcedure :
  |[ function f(x*) = e ]| ->
  |[ function f(x*) = (enterfun(s); e; exitfun(s)) ]|
  where
    !f => s
```

meta-variables

object identifiers

```
variables
  [xyzfgh][0-9]*  -> Id          {prefer}
  [s][0-9]*       -> StrConstr   {prefer}
  [ijkn][0-9]*    -> IntConst    {prefer}
  [e][0-9]*       -> Exp         {prefer}
  "e"[0-9]* "*"   -> {Exp ";"}+  {prefer}
  "a"[0-9]* "*"   -> {Exp ","}+  {prefer}
  "fd"[0-9]* "*"  -> FunDec+     {prefer}
```

Anti-Quotation

```
TraceProcedure :
  |[ function ~f(~*xs) = ~e ]| ->
  |[ function ~f(~*xs) = (enter(~s); ~e; exit(~s)) ]|
  where
    !f => s
```

Distinguish meta-variables by anti-quotation

```
TraceProcedure :
  |[ function ~f(~*xs) = ~e ]| ->
  |[ function ~f(~*xs) =
       (print(~String(<conc-strings>(f," entry")));
        ~e;
        print(~String(<conc-strings>(f," exit")))) ]|
```

Splice code generated by meta-computation into object code

Congruences

```
cong1 = |[ if <s> then <id> else <id> ]|

cong2 = |[ let <*map(s)> in <*id> end ]|

cong3 = |[ let <fd:s> in <*id> end ]|

cong4 = |[ let <fd*:map(s)> in <*id> end ]|
```

Concrete syntax for congruences

```
context-free syntax
  "<"      StrStrategy ">" -> Exp     {cons("FromApp")}
  "<*"     StrStrategy ">" -> Dec*    {cons("FromApp")}
  "<fd:"   StrStrategy ">" -> FunDec  {cons("FromApp")}
  "<fd*:"  StrStrategy ">" -> FunDec+ {cons("FromApp")}
```

Exploding Embedded Abstract Syntax: Example

```
|[ x := let ds in ~* es end ]| -> |[ let ds in x := (~* es) end ]|
```

```
Rule(ToTerm(Assign(Var(meta-var("x")),
                   Let(meta-var("ds"),FromTerm(Var("es"))))),
     ToTerm(Let(meta-var("ds"),
                [Assign(Var(meta-var("x")),
                        Seq(FromTerm(Var("es"))))])))
```

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                  Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
               Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                       Op("Seq",[Var("es")])]),
                          Op("Nil",[])])]))
```

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

SGLR: Scannerless Generalized LR Parsing

Restricted classes of context-free grammars not closed under composition.

- LR/LL parsers do not compose
- Scanners (regular grammars) do not compose

Scannerless Generalized LR Parsing

- Integration of lexical and context-free syntax
- Handle full class of context-free grammars
- Context-based disambiguation
- Unbounded lookahead
- Supports *natural* syntax definition
- Limitation: only *context-free* grammars

Applications

- Java code generation and transformation
  - Java-front
- XML generation in Stratego
  - Available in Stratego/XT
- Handcrafted pretty-printers: Box embedded in Stratego
  - Available in Stratego/XT
- Meta Stratego
  - Stratego Compiler
  - Stratego code generators
- Meta SDF
  - Code generators in Stratego/XT

Application: Tiger/XML

```
function webpage(title : elt, body : elt) : elt =
  <html>
    <head>
      <title> $title </title>
    </head>
    <body>
       $body
    </body>
  </html>
```

Embed XML in a programming language

Application: Stratego/XML

```
ra2mathml(s) =
  !%>
    <?xml version="1.0"?>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <% s %>
    </math>
  <%

ra-to-presentation-mathml(x) :
  Project(ps, r) ->
    %><mrow>
        <msub>
          <mo>&pi;</mo>
          <mrow>
            <mfenced open="" close="">
              <% <map(x)> ps :: * %>
            </mfenced>
          </mrow>
        </msub>
        <mfenced>
          <% <x> r %>
        </mfenced>
      </mrow><%
```

Application: Stratego/Tiger/XML

```
ExpElt :
  |[ <tag> cont* </tag> ]| ->
  |[ let var x := contents[k] of nil
      in ( e* );
         elt{tag = s, text = "", contents = x, size = k}
     end ]|
  where
    new => x; !tag => s
  ; <length> [cont*] => k
  ; <dec; upto> k => is
  ; <zip(
    \(cont, i) -> |[ x[i] := <content> cont </content> ]|\
    )> ([cont*], is) => [e*]
```

Embedding can be nested

Application: Stratego/Stratego

```
Desugar :
  |[ s => t ]| -> |[ s; ?t ]|

Desugar :
  |[ <s> t :S]| -> |[ !t; s ]|

Desugar :
  |[ f(as) : t1 -> t2 where s ]|
    ->
  |[ f(as) = ?t1; where(s); !t2 ]|

Desugar :
  |[ if s1 then s2 else s3 end ]|
    ->
  |[ where(s1) < s2 + s3 ]|
```

Transformation of Stratego programs in Stratego

Application: Stratego/Box

```
expr-to-box :
  Plus(b1, b2) -> H hs=1 [ b1 "+"  b2]

UglyPrint :
  If(b1, b2, b3) ->
    V vs=0 [
      H hs=0 [KW["if"] "(" b1 ")"]
      b2
      KW["else"] b3]

PrettyPrint :
  If(b1, b2, If(b3, b4, b5)) ->
    V vs=0 [
      H hs=0 [KW["if"] "(" b1 ")"]
      b2
      H hs=1 [KW["else"] H hs=0 [KW["if"] "(" b3 ")"]]
      b4
      KW["else"] b5]
```

Application: Stratego/Java

Talk on Tuesday!

Application: Java/Java

Embed Java syntax in Java

```
context-free syntax
  "type" "|[" Type "]|" -> MetaExpr {cons("ToMetaExpr")}

variables
  "e" [0-9]*      -> Expr          {prefer}
  "e" [0-9]* "*" -> {Expr ","}*   {prefer}
```

Assimilation rules for Eclipse JDT Core API

```
  Assimilate(r) :
    type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

  Assimilate(r) :
    |[ y(e*) ]| -> |[
      {| MethodInvocation x = ast.newMethodInvocation();
         x.setName(ast.newSimpleName("~y"));
         bstm* | x |}
    ]|
    where <newname> "inv" => x
        ; <ExplodeArgs(r | x)> e* => bstm*
```

Conclusion

Extend meta language with object language syntax

- combine arbitrary meta and object language
- meta-language does not need to be designed for this purpose
- limitation: language should have context-free syntax

Modular syntax definition is essential

- reuse existing syntax definitions – no changed needed

Scannerless Generalized-LR parsing

- Only context-free grammars closed under composition
- No compositionality: regular grammars, LR, LL