Stratego/XT in a Nutshell

An introduction to the Stratego/XT Transformation System

Karl Trygve Kalleberg

Institutt for Informatikk
Universitetet i Bergen

University of Waterloo
Feb 23, 2006

Outline

1. Motivation
   - Explain motivation and background for Stratego/XT
   - Introduce philosophy and system architecture
2. Transformation Mechanics
   - Introduce *characteristic* system features
   - Show application through an example of language embedding
3. Experience
   - Mention other projects using Stratego/XT
   - Mention other types of transformations with Stratego/XT
4. Conclusion
   - With pointers to more resources

Part I

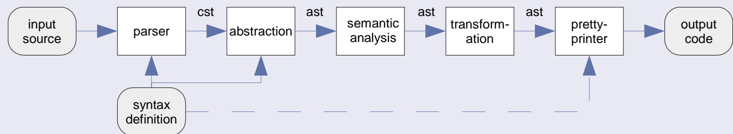
Motivation

The Stratego/XT 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.

The Big Picture

A Simple Pipeline



1. Source code is *parsed* into a *structured* representation, a *concrete syntax tree* (CST).
2. The CST is pruned for non-essential information, to yield an *abstract syntax tree* (AST).
3. Additional information is added, e.g. type information.
4. One or multiple *transformations* are applied to the AST.
5. The result is serialized back to file.

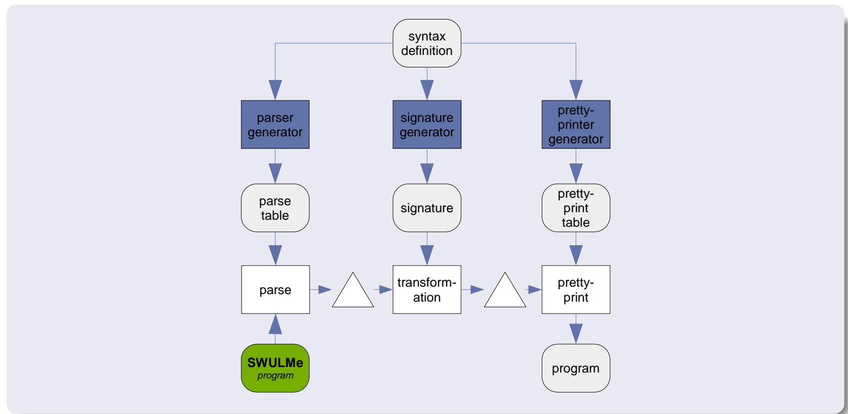
The Small Picture

The *syntax definition* declares the *structure* of the language for the programs we transform and is used in constructing most pipeline stages.

Part II

Transformation Mechanics

The Input



Step 0

A SWUL Example

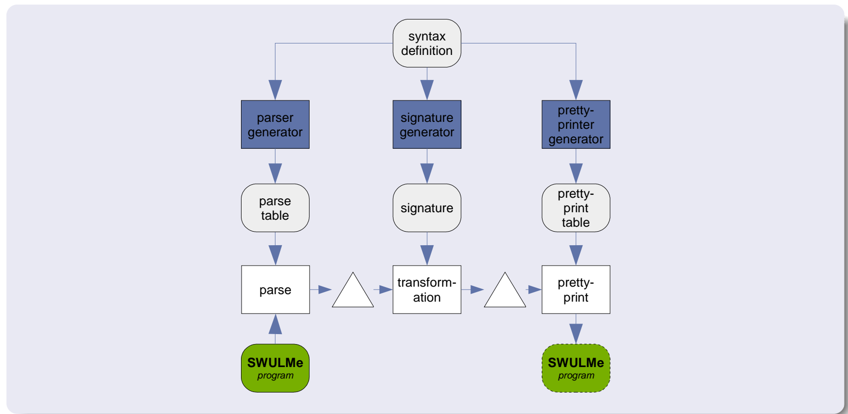
Java with SWUL

```
class SWULMe {
  public static void main(String[] args) {
    JFrame frame = frame {
      title = "Welcome"
      content = panel of border layout {
        center = label { text = "Hello World" }
        south = panel of grid layout {
          row = {
            button { text = "cancel" }
            button { text = "ok" }
          }
        }
      }
    }
  }
}
```

Thanks

The SWUL material is taken from Rene de Groot's Master thesis, *Design and Implementation of Embedded Domain-Specific Languages*.

The Output



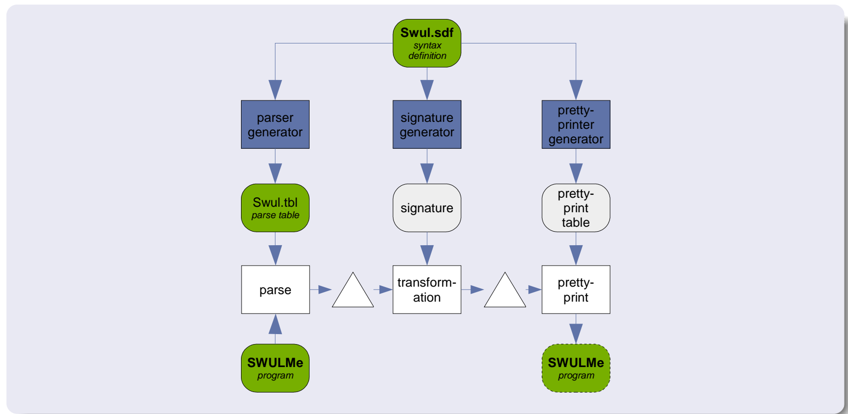
Step 1

Assimilated Result

Pure Java after desugaring

```
class SWULMe {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Welcome");
    JPanel mainPanel = new JPanel();
    BorderLayout borderLayout = new BorderLayout();
    JLabel helloLabel = new JLabel("Hello World");
    JPanel southPanel = new JPanel();
    JButton cancelButton = new JButton("cancel");
    JButton okButton = new JButton("ok");

    southPanel.setLayout(new GridLayout(1,2));
    southPanel.add(cancelButton);
    southPanel.add(okButton);

    mainPanel.setLayout(borderLayout);
    mainPanel.add(southPanel, BorderLayout.SOUTH);
    mainPanel.add(helloLabel, BorderLayout.CENTER);

    frame.setContentPane(mainPanel);
  }
}
```

Defining the SWUL Syntax



Step 2

SWUL Syntax

Swul.sdf – Grammar for the embedded language

```
module Swul
exports
  context-free start-symbols Component

sorts Component ComponentType ComponentProps ComponentPropValues ComponentPropType
context-free syntax

  ComponentType ComponentProps? -> Component {cons("Component")}

  "panel"                  -> ComponentType {cons("JPanel")}
  "button"                 -> ComponentType {cons("JButton")}
  "border" "layout"        -> ComponentType {cons("BorderLayout")}
  "grid" "layout"          -> ComponentType {cons("GridLayout")}
  "frame"                  -> ComponentType {cons("JFrame")}

  "{" ComponentProp* "}"  -> ComponentProps {cons("ComponentProps")}
  "of" Component           -> ComponentProps {cons("DefaultPropsWithComponent")}

  ComponentPropType "=" ComponentPropValues -> ComponentProp {cons("ComponentProp")}
  "{" Component* "}"                    -> ComponentPropValues {cons("ComponentPropMultiValue")}

  "content"                -> ComponentPropType {cons("Content")}
  "title"                  -> ComponentPropType {cons("Title")}
  "row"                    -> ComponentPropType {cons("Row")}
  "south"                  -> ComponentPropType {cons("South")}
  "center"                 -> ComponentPropType {cons("Center")}
  "border"                 -> ComponentPropType {cons("Border")}
```

Embedding SWUL in Java

Java-Swul.sdf

```
module Java-Swul
imports Java-15-Prefixed Swul-Prefixed

exports
  sorts JavaExpr SwulComponent JavaBlock
  context-free syntax

    SwulComponent -> JavaExpr      {cons("ToExpr")}
```

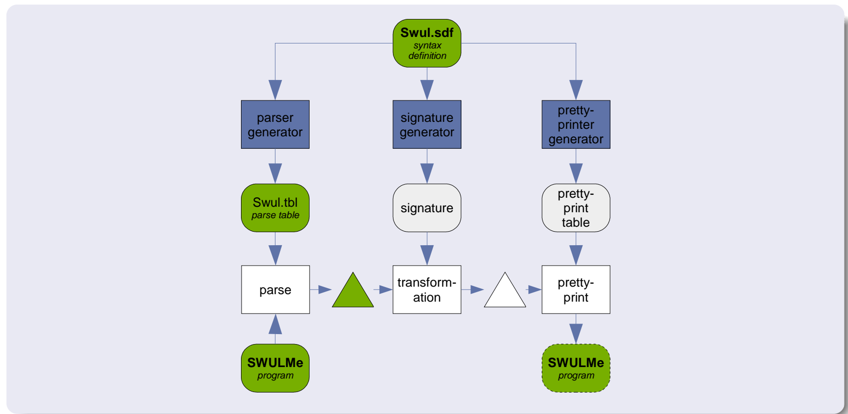
Embedding SWUL in Java

Java-Swul.sdf

```
module Java-Swul
imports Java-15-Prefixed Swul-Prefixed

exports
  sorts JavaExpr SwulComponent JavaBlock
  context-free syntax

    SwulComponent -> JavaExpr      {cons("ToExpr")}
```

```
    JavaExpr      -> SwulComponent {cons("FromExpr")}

    JavaBlock     -> SwulComponent {cons("FromBlock")}
```

Parsing SWULen Code



Step 3

From Concrete Syntax to Terms

Concrete Syntax

```
x := 1 + 2
```

$\Downarrow$ (*parser*)

Abstract Syntax Tree

```
Assign(x, Add(Int(1), Int(2)))
```

Terms

- A term is a *structured representation* of programs.
- Its structure is often derived from the syntax definition of the language.
- The syntax definition gives rules for the structural validation of terms.

Terms are primarily used to encode *abstract syntax trees*.

Annotated Terms

Structural Definition of Annotated Terms

```
t  := bt                      -- basic term
    | bt { t }                -- annotated term

bt := C                       -- constant
    | C(t1,...,tn)            -- n-ary constructor
    | (t1,...,tn)             -- n-ary tuple
    | [t1,...,tn]             -- list
    | "ccc"                   -- quoted string
    | int                     -- integer
    | real                    -- floating point number
    | blob                    -- binary large object
```

Why *abstract* syntax trees?

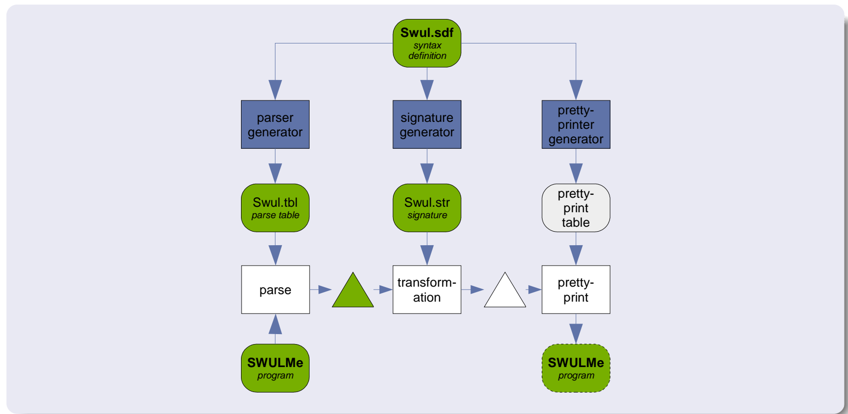
Concrete Syntax

$$x := 1 + 2$$

Accurate Concrete Syntax Tree

```
parsetree(appl(prod([cf(opt(layout)),cf(sort("Program")),cf(opt(layout))],sort("<START>"),no-at
trs),[appl(prod([],cf(opt(layout)),no-attrs),[]),appl(prod([cf(iter-star(sort("Stat")))],cf(sor
t("Program")),attrs([term(cons("Program"))])),[appl(list(cf(iter-star(sort("Stat")))),[appl(pro
d([lit("var"),cf(opt(layout)),cf(sort("Id")),cf(opt(layout)),lit(";")],cf(sort("Stat")),attrs([
term(cons("Declaration"))])),[lit("var"),appl(prod([cf(layout)],cf(opt(layout)),no-attrs),[32])
,appl(prod([lex(sort("Id"))],cf(sort("Id")),no-attrs),[appl(list(iter-star(char-class([range(0,
255)]))),[120])]),appl(prod([],cf(opt(layout)),no-attrs),[]),lit(";")]),appl(prod([cf(layout)],
cf(opt(layout)),no-attrs),[10]),appl(prod([cf(sort("Id")),cf(opt(layout)),lit(":="),cf(opt(layo
ut)),cf(sort("Exp")),cf(opt(layout)),lit(";")],cf(sort("Stat")),attrs([term(cons("Assign"))])),
[appl(prod([lex(sort("Id"))],cf(sort("Id")),no-attrs),[appl(list(iter-star(char-class([range(0,
255)]))),[120])]),appl(prod([cf(layout)],cf(opt(layout)),no-attrs),[32]),lit(":="),appl(prod([c
f(layout)],cf(opt(layout)),no-attrs),[32]),appl(prod([cf(sort("Exp")),cf(opt(layout)),lit("+"),
cf(opt(layout)),cf(sort("Exp"))],cf(sort("Exp")),attrs([term(cons("Add")),assoc(assoc)])),[appl
(prod([cf(sort("Int"))],cf(sort("Exp")),attrs([term(cons("Int"))])),[appl(prod([lex(sort("Int")
)],cf(sort("Int")),no-attrs),[appl(list(iter-star(char-class([range(0,255)]))),[49])])]),appl(p
rod([cf(layout)],cf(opt(layout)),no-attrs),[32]),lit("+"),appl(prod([cf(layout)],cf(opt(layout)
),no-attrs),[32]),appl(prod([cf(sort("Int"))],cf(sort("Exp")),attrs([term(cons("Int"))])),[appl
(prod([lex(sort("Int"))],cf(sort("Int")),no-attrs),[appl(list(iter-star(char-class([range(0,255
)]))),[50])])])]),appl(prod([],cf(opt(layout)),no-attrs),[]),lit(";")])])]),appl(prod([cf(layou
t)],cf(opt(layout)),no-attrs),[10])]),0)
```

Programs as Data



Step 4

Signatures

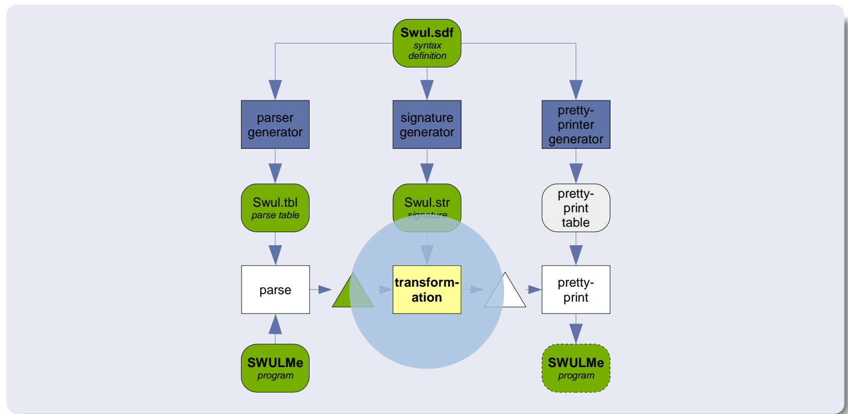
- In Stratego, signatures are *data declarations* that define the *structure* of terms.
- Terminology comes from the field of *universal algebra*; sorts with operations.
- A *signature* is a set of *constructors* on the form:
  - `name : sort * sort * ... -> sort`

Signatures

- In Stratego, signatures are *data declarations* that define the *structure* of terms.
- Terminology comes from the field of *universal algebra*; sorts with operations.
- A *signature* is a set of *constructors* on the form:
  - `name : sort * sort * ... -> sort`

```
signature
 sorts
  Exp
 constructors
  Plus       : Exp * Exp                         -> Exp
  Var        : Id                                -> Exp
  Assign     : Id * Exp                          -> Exp
  Int        : String                            -> Exp
  For        : Exp * Exp * Exp * List(Exp)       -> Exp
  If         : Exp * List(Exp) * List(Exp)       -> Exp
```

Writing Transformations



Step 5

Language Features for Transformation

- Patterns – on abstract and concrete syntax
- Rewrite Rules – with and without conditions
- Strategies – primitive, composed and higher-order

Patterns

Structural matching on terms are done with *term patterns*.

```
If(e, [th | th*], ls)
```

matches

```
If(Int(0), [Assign("x", Plus(Int(1), Var("y")))], [])
```

Bindings

- *e* – `Int(0)`,
- *th* – `Assign("x", Plus(Int(1), Var("y")))`
- *th*∗ – `[]`
- *ls* – `[]`

Rewrite Rules

Definition

A rewrite rule

```
R : l -> r where c
```

replaces the pattern *l*, called the *left-hand side*, with *r*, the *right-hand side*, when:

- *l* matches the *current term*
- *c* holds

Informally

Rewrite rules are basic units of tranformation, taking one term to another, based on structural pattern matching.

Rewrite Rules

```
Simplify:
  If(Int("0"), th, ls) -> th

Simplify:
  Plus(e, Int("0")) -> e

Simplify:
  Plus(Int(x), Int(y)) -> Int(z)
  where <addS> (x, y) => z

Simplify:
  Plus(Int(x), Int(y)) -> Int(<addS> (x, y))
```

Controlling Rule Application

Language Features Supporting Term Navigation

- Strategies
- Generic Traversals
- Congruences (not presented)
- Dynamic Rules (not presented)

Strategies

What is a Strategy?

- Any rule
- `fail`, `id`
- `!` (build), `?` (match)
- Any congruence, later
- Any generic traversal, next

Strategies

What is a Strategy?

- Any rule
- `fail`, `id`
- `!` (build), `?` (match)
- Any congruence, later
- Any generic traversal, next

Composing Strategies

- Sequential composition: s_1 `;` s_2
- Deterministic (left) choice: s_1 `<+` s_2
- Choice: s_1 `<` s_2 `+` s_3
- `try(s) = s <+ id`

Generic Traversals

Primitive Strategies

- `one(`s`)` – applies strategy s to one direct sub-term
- `some(`s`)` – applies strategy s to some direct sub-terms
- `all(`s`)` – applies strategy s to all direct sub-terms

Generic Traversals

Primitive Strategies

- `one(s)` – applies strategy *s* to one direct sub-term
- `some(s)` – applies strategy *s* to some direct sub-terms
- `all(s)` – applies strategy *s* to all direct sub-terms

Examples

```
<all(!0)> A(1,2,3) ⇒ A(0,0,0)
<some(!0)> A(1,2,3) ⇒ A(0,0,0)
<one(!0)> A(1,2,3) ⇒ A(0,2,3)
```

Generic Traversals

Primitive Strategies

- `one(s)` – applies strategy *s* to one direct sub-term
- `some(s)` – applies strategy *s* to some direct sub-terms
- `all(s)` – applies strategy *s* to all direct sub-terms

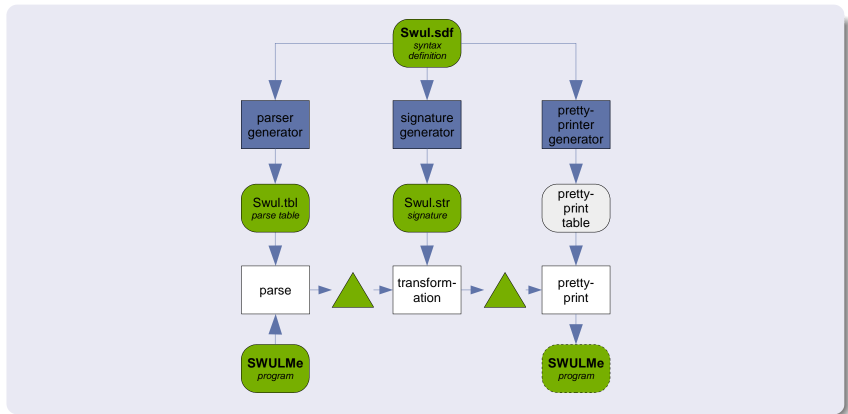
Examples

```
<all(!0)> A(1,2,3) ⇒ A(0,0,0)
<some(!0)> A(1,2,3) ⇒ A(0,0,0)
<one(!0)> A(1,2,3) ⇒ A(0,2,3)
```

Traversal Strategies

```
bottomup(s)  = all(bottomup(s)); s
topdown(s)   = s; all(topdown(s))
downup(s)    = s; all(downup(s)); s
alltd(s)     = s <+ all(alltd(s))
innermost(s) = bottomup(try(s; innermost(s)))
```

Assimilation 101



Step 6

Assimilating SWUL

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

Assimilating SWUL

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

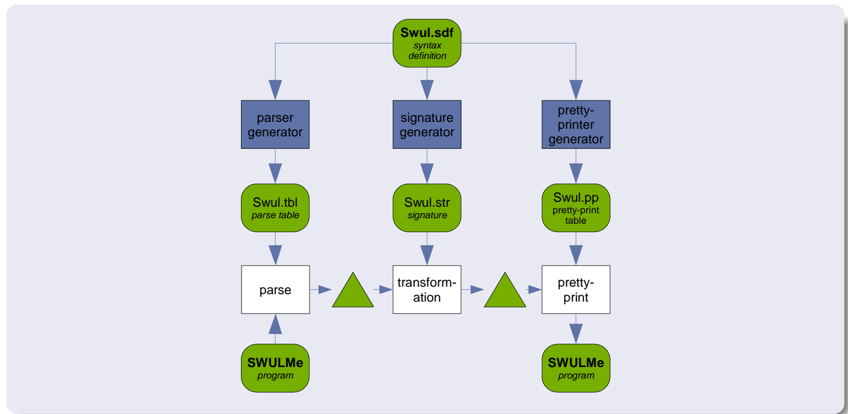
```
swul-expression =
      ?ToExpr(swul)
    ; {|  local-declarations :
              <SwulAs-Component> swul => e
            ; bagof-local-declarations => bstm*
      |}
    ; <?[] < !e + !java|[ | bstm* | e | ]|> bstm*
```

Assimilating SWUL

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

```
swul-expression =
      ?ToExpr(swul)
    ; {|  local-declarations :
            <SwulAs-Component> swul => e
          ; bagof-local-declarations => bstm*
      |}
    ; <?[] < !e + !java|[ | bstm* | e | ]|> bstm*
```

```
SwulAs-Component  = SwulAs-Container  + SwulAs-JavaExpr
SwulAs-Container  = SwulAs-JComponent
SwulAs-JComponent = SwulAs-JPanel  + SwulAs-JToolBar + SwulAs-JSplitPane + ...

SwulAs-JPanel:
  swul c |[ panel { ps* } ]|{x} -> expr |[ {| x = new JPanel(); bstm* | x |} ]|
  where <map(SwulAs-JPanelProp(|x)> ps* => bstm*

SwulAs-JPanelProp = ... + SwulAs-ContainerProp + ...

SwulAs-ContainerProp(|x) :
  swul cp |[ layout = c ]| -> bstm |[  x.setLayout( e ) ; ]|
  where <SwulAs-LayoutManager(|x)> c => e
```

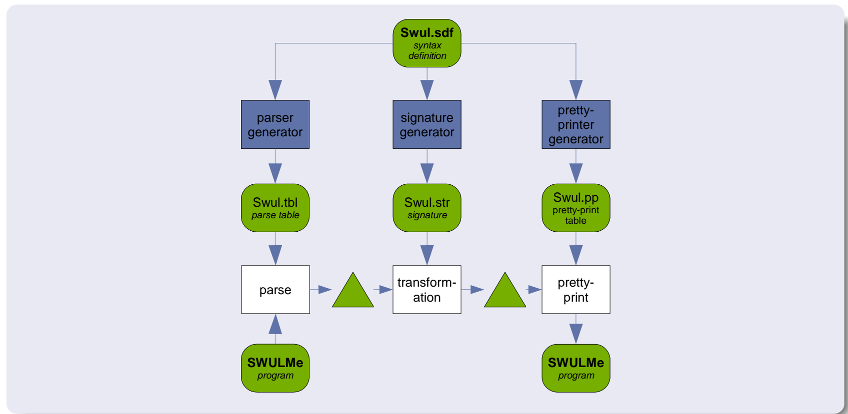


Step 7

Pretty Printing SWUL Code

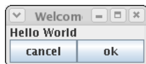
```
...

swul-to-box :
  JPanel() -> |[ H [ KW["panel"] ] ]|

swul-to-box :
  BorderLayout() -> |[ H [ KW["border"] KW["layout"] ] ]|

swul-to-box :
  Component(b1,b2,b3) -> |[ V vs=0 [ H [ b1 b2 ] b3  ] ]|

swul-to-box :
    ComponentPropMultiValue(xs) -> |[ V vs=0 [KW[""] ~*xs KW[""] ] ]|

...
```

Compilation



Step 8

Execution (= Profit!)



Step 9

Part III

Experience

Transformation Systems Based on Stratego

- BibTeX Tools – Bibliography handling for BibTeX
- CodeBoost (UiB) – Domain-specific optimization for C++
- Dryad – Open compiler for Java (in progress)
- OctaveC – Compiler for Octave, a Matlab clone
- Plover (OGI) – Theorem prover for Haskell variant
- Proteus (Lucent) – Transformation framework for C++
- Tiger – demonstration compiler for Tiger to MIPS
- Transformers (EPITA) – Transformation framework for C/C++

Tools and Extensions for Stratego

- AspectStratego - An aspect extension to Stratego
- Spoofax – An editor for Stratego in Eclipse
- xDoc – Source code documentation system for Stratego

Applications of Stratego

- Compilation
  - Desugaring, variable renaming, translation, instruction selection., data-flow optimizations, type checking, escape analysis
- Optimization
  - Mutification, vectorization, partial evaluation and specialization, automatic derivation
- Verification
  - Ad-hoc theorem proving (on Haskell)
- Reengineering
  - Layout preserving transformations (on COBOL)
- Execution
  - interpretation (of Stratego)

Conclusion

- Stratego/XT 0.16
  - Works *on* and *for* Linux, Unix, OSX and Windows
  - Licensed under the LGPL
  - Tutorial, examples and API documentation available online.
- Spoofax 0.4.0
  - Syntax highlighting, source code navigation, content assistance, source code outliner.

`http://www.stratego-language.org`

`http://www.spoofax.org`