

Infrastructure for Program Transformation Systems

Master Course Program Transformation 2005-2006

Martin Bravenboer

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

February 07, 2006

Stratego

- Language for program transformation
- Suitable for implementing complete programs

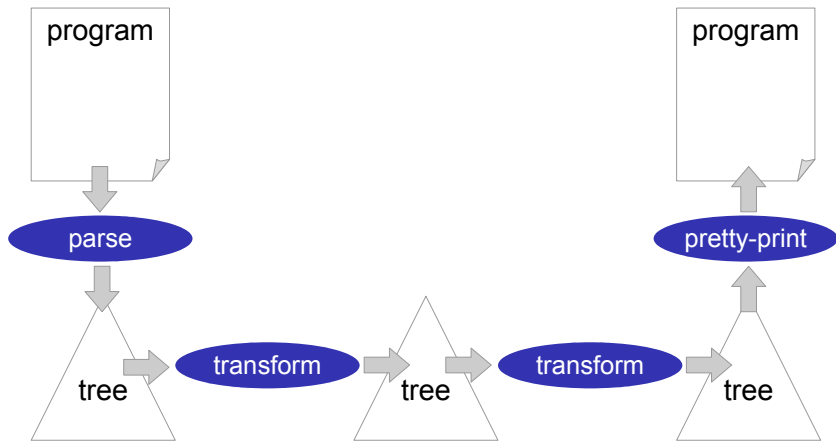
XT

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

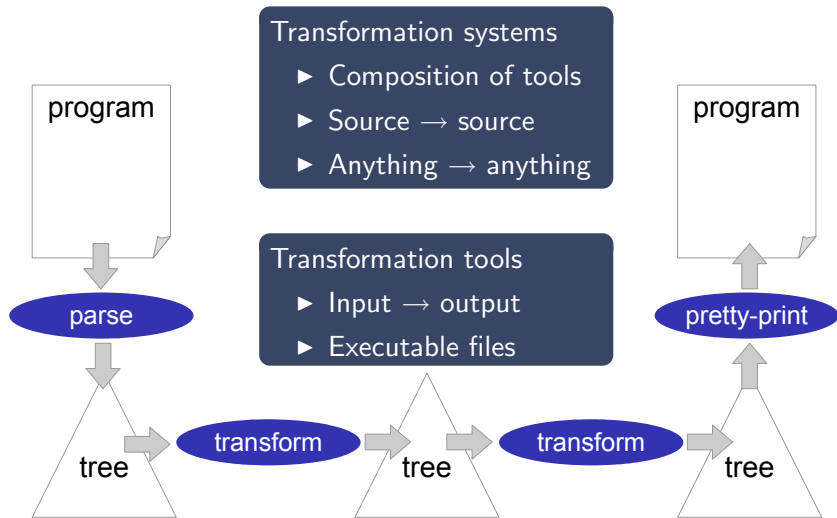
XT Orbit

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

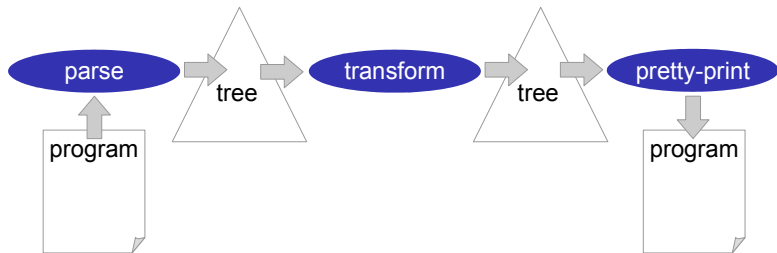
Program Transformation Pipeline



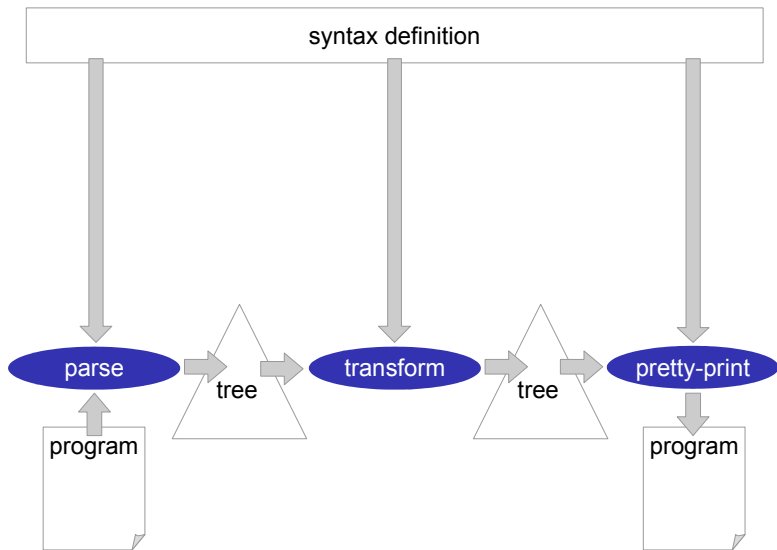
Program Transformation Pipeline



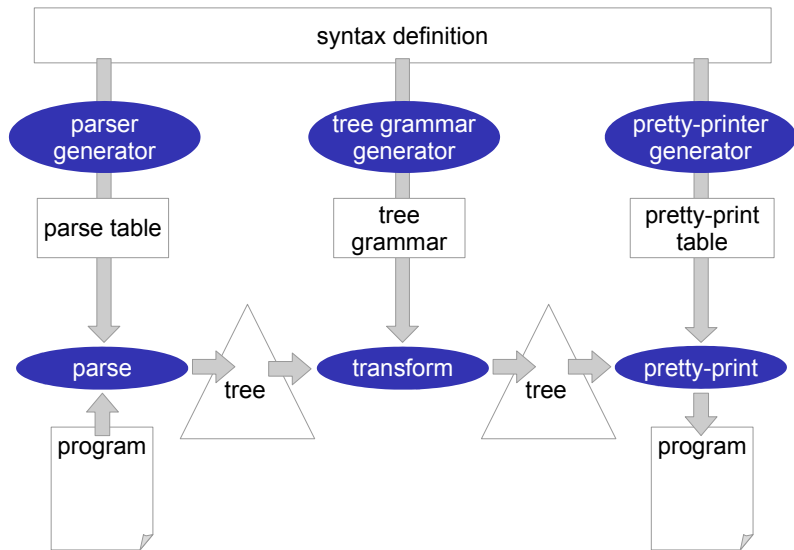
Architecture of Stratego/XT



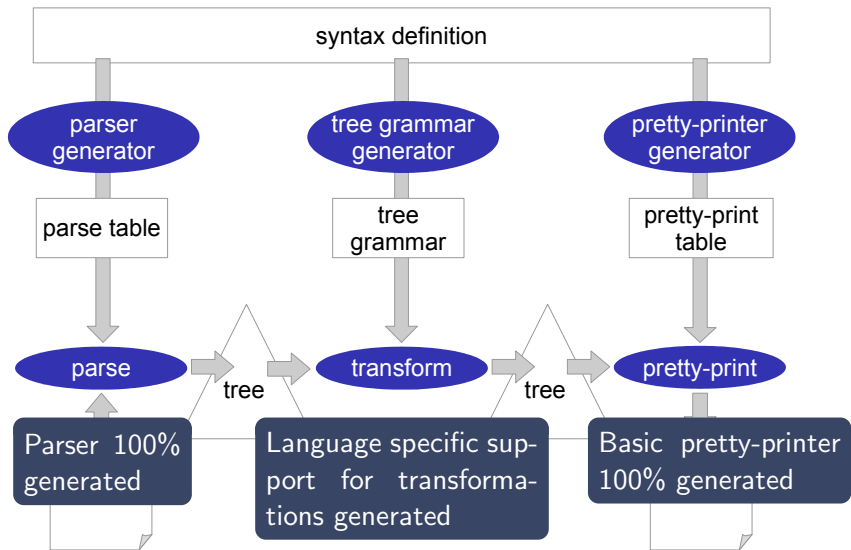
Architecture of Stratego/XT



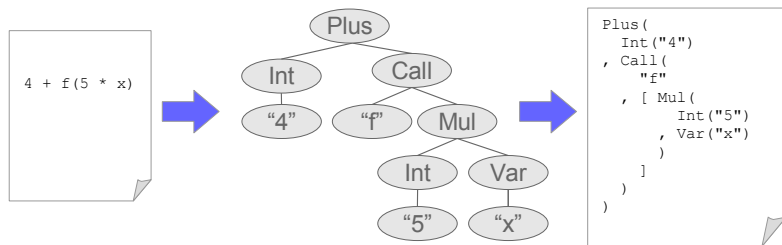
Architecture of Stratego/XT



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"))]))
```

A Term Format

Application	<code>Void(), Call(<i>t</i>, <i>t</i>)</code>
List	<code>[], [<i>t</i>, <i>t</i>, <i>t</i>]</code>
Tuple	<code>(<i>t</i>, <i>t</i>), (<i>t</i>, <i>t</i>, <i>t</i>)</code>
Integer	<code>25</code>
Real	<code>38.87</code>
String	<code>"Hello world"</code>
Annotated term	<code><i>t</i>{<i>t</i>, <i>t</i>, <i>t</i>}</code>

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

Structured Data: comparable to XML

Stratego: internal is external representation

Simple Expression Language

Id → [a-z]⁺

IntConst → [0-9]⁺

Exp → *Id*

IntConst

Exp + *Exp*

Exp - *Exp*

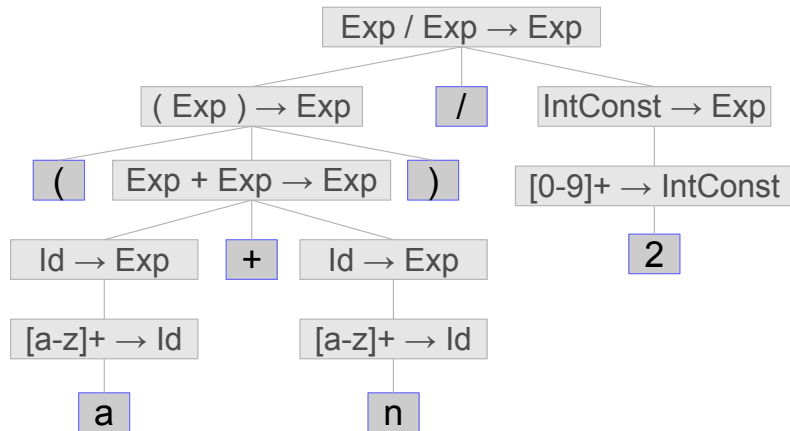
Exp * *Exp*

Exp / *Exp*

(*Exp*)

Parse Trees

$(a + n) / 2$



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

Declarative

- Important for code generation
- Completely define the syntax of a language

Modular

- Syntax definitions can be composed!

Context-free and lexical syntax

- No separate specification of tokens for scanner

Declarative disambiguation

- Priorities, associativity, follow restrictions

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
```

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 Syntax

Declaring reserved keywords: reject certain productions

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

Solves ambiguity between variable and boolean constant.

```
$ echo "true" | sglri -p Example.tbl
amb([Bool("true"),Var("true")])
```

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]
```

Rejects unintended split of identifier

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

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 keyword

```
$ 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

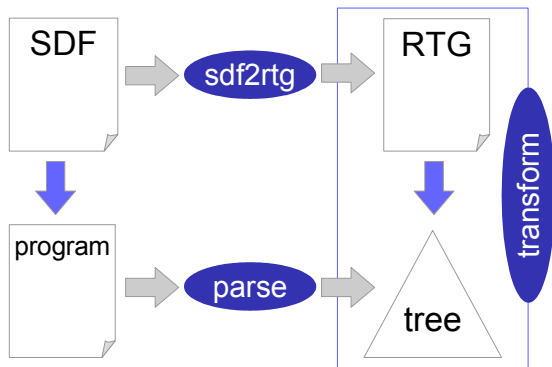
test
  file foo.exp succeeds
```

```
$ 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
```

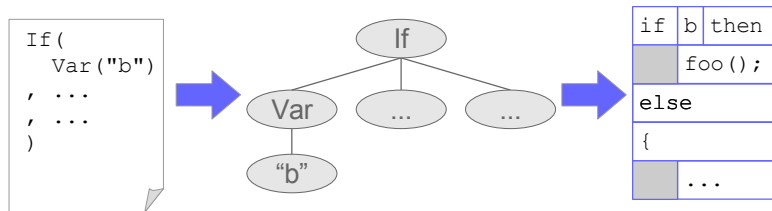
```
martin@logistico:~> format-check --rtg Exp.rtg -i exp3.trm --vis
error: cannot type Int(1)
  inferred types of subterms:
    typed 1 as <int>
error: cannot type Div(1,Var("c"))
  inferred types of subterms:
    typed 1 as <int>
    typed Var("c") as Exp
Plus(
  Mul(Int(1), Var("a"))
, Minus(Var("b"),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

H hs=x [B B B] → B ↔ B ↔ B

V vs=x is=y [B B B] → B ↔ B ↔ B

A hs=x vs=y [R [B B B] R [B B B]]

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[","]
]
```

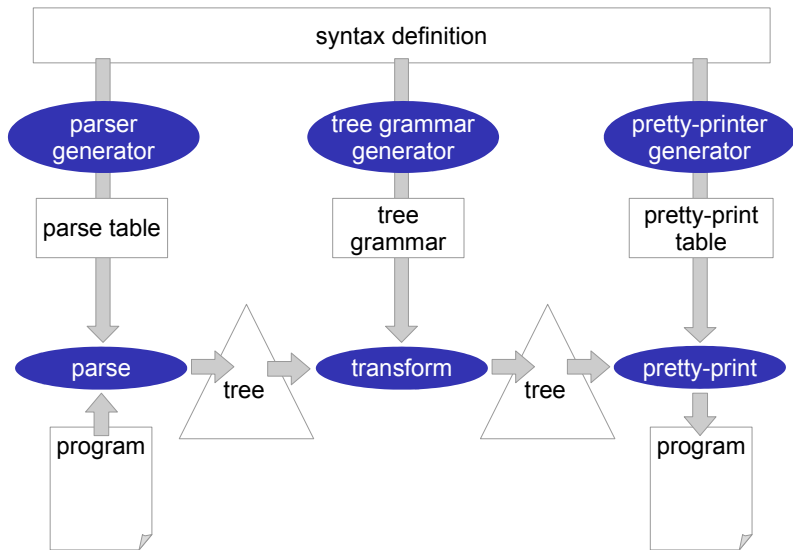
- ast2abox accepts sequence of pretty-print tables
- Tables can be combined and reused

```
$ 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



- **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