**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
- parse
- tree
- transform
- tree
- transform
- tree
- pretty-print
- program
Program Transformation Pipeline

Transformation systems
- Composition of tools
- Source → source
- Anything → anything

Transformation tools
- Input → output
- Executable files
Architecture of Stratego/XT

parse → tree

transform → tree

pretty-print → program

program → parse

program → transform

program → pretty-print
Architecture of Stratego/XT

syntax definition

parse → tree → transform → tree → pretty-print

program → tree → tree → program
Architecture of Stratego/XT

- Syntax definition
  - Parser generator
    - Parse table
      - Parse
  - Tree grammar generator
    - Tree grammar
      - Transform
  - Pretty-printer generator
    - Pretty-print table
      - Pretty-print

- Parser 100% generated
- Language specific support for transformations generated
- Basic pretty-printer 100% generated
Trees are represented as terms in the ATerm format

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Void(), Call(t, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>[], [t, t, t]</td>
</tr>
<tr>
<td>Tuple</td>
<td>(t, t), (t, t, t)</td>
</tr>
<tr>
<td>Integer</td>
<td>25</td>
</tr>
<tr>
<td>Real</td>
<td>38.87</td>
</tr>
<tr>
<td>String</td>
<td>&quot;Hello world&quot;</td>
</tr>
<tr>
<td>Annotated term</td>
<td>t{t, t, t}</td>
</tr>
</tbody>
</table>

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

**Structured Data**: comparable to XML
**Stratego**: internal is external representation
## Simple Expression Language

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>→ [a-z]+</td>
</tr>
<tr>
<td>IntConst</td>
<td>→ [0-9]+</td>
</tr>
<tr>
<td>Exp</td>
<td>→ Id</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exp → IntConst
Exp → Exp + Exp
Exp → Exp - Exp
Exp → Exp * Exp
Exp → Exp / Exp
( Exp )
(a + n) / 2

Parse Trees

```
(a + n) / 2

Exp / Exp → Exp

( Exp ) → Exp

/ IntConst → Exp

( ) [0-9]+ → IntConst

( )

/ Exp / Exp → Exp

IntConst → Exp

[0-9]+ → IntConst

Exp + Exp → Exp

Id → Exp

[ a-z ]+ → Id

2

a

n

Id → Exp +

Exp + Exp → Exp

2

( Exp ) → Exp

/a

Exp IntConst → Exp

[a-z]+ → Id

[a-z]+ → Id

(a + n) / 2
```
AsFix: A Term language for parse trees

- Describes Applications of productions
- All characters of the input
  - Even whitespace and comments!
- Yield parse tree to text
  
  $\texttt{asfix-yield \ -i \ exp.asfix}$
  $(a + n) / 2$

Abstract Syntax Trees

- Remove literals, whitespace, comments
  
  $\texttt{implode-asfix \ -i \ exp.asfix}$
  $\text{Div(Plus(Var("a"), Var("n")), Int("2"))}$
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

- $e_1 + e_2 \ast e_3$
  - $(e_1 + e_2) \ast e_3$
  - $e_1 + (e_2 \ast e_3)$

- $e_1 + e_2 + e_3$
  - $(e_1 + e_2) + e_3$
  - $e_1 + (e_2 + e_3)$

- $++a$
  - $+(+a)$
  - $+ + a$

- null
  - Keyword or identifier?

- if $e_1$ then if $e_2$ then $e_3$ else $e_4$
  - if $e_1$ then (if $e_2$ then $e_3$) else $e_4$
  - if $e_1$ then (if $e_2$ then $e_3$ else $e_4$)
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
module Lexical
exports
  lexical syntax
...

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

module Main
imports Expressions
exports
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
Lexical syntax is defined with ordinary productions.

```plaintext
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
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]
Declaring reserved keywords: reject certain productions

**lexical syntax**

"true" -> Id {reject}
"false" -> Id {reject}

Solves ambiguity between variable and boolean constant.

```bash
$ 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]
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]
```

Recoilest uineanted split of identifier

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

Require layout after a keyword

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

classical 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")}
$ 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

```bash
$ echo "1 + 2 * 3" | sglri -p Example.tbl
```

```amb
amb([ Mul(Plus(Int("1"), Int("2")), Int("3")),
     Plus(Int("1"), Mul(Int("2"), Int("3")))) ])
```

class `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
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
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 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
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
- 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

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

Other boxes: HV, ALT, KW, VAR, NUM, C
while a do
  if b then
    foo();
  else
    {
      ...
    }
Pretty-print Tables

- List of pretty-print rules
- Applied by constructor name (\texttt{cons} attribute)

### Example Pretty-Print Table

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

- \texttt{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 (sdf2parenthesizer).
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