Language Engineering Tools

Stratego/XT

Martin Bravenboer
martin@cs.uu.nl

Institute of Information and Computing Sciences, University Utrecht, The Netherlands
Overview

Parsing and pretty-printing

*parse-unit*

syntax definition testing

*StrategoBox*

advanced pretty-printers

Applications of abstract syntax definitions

*sdf2rtg* and *rtg2sig*

*rtg2typematch* and *wf-checker*

*xml-interpret*

interoperability of XML and aterm tools
parse-unit

Testing Syntax Definitions for Fun
Syntax Definition is Software Engineering

- configuration management
  - build management
  - version management
  - deployment

- process
  - extreme syntax definition

- validation and verification
  - inspection and *testing*

- evolution

- metrics
Importance of Syntax Definition Testing

• usual arguments: deployment, build management, evolution

• conventional parsing techniques:
  ◦ definition is not a definition
  ◦ undo grammar hacking
  ◦ implicit disambiguation

• documentation for language implementers and users

• large, modular syntax definitions
  ◦ ambiguities not obvious
  ◦ unexpected results (e.g. embeddings)
Current Techniques for Syntax Definition Testing

1) by hand: waste of time

2) some larger inputs
   • no checking of results
   • poor error-reporting

⇒ apply unit-testing techniques!
   • verification of results
   • excellent error reports
   • documentation
Unit-Testing without Tool Support

from build system

1) files for input and output
   • poor overview
   • discourages a lot of atomic tests

2) automake tests: programs that succeed or fail

3) application from build system requires work/experience

from Stratego or shell-script:

1) escape special (?!) characters

2) no reuse for different purposes:
   • documentation
   • different parsing tools
   • testsuite analysis, such as coverage
Example parse-testsuite

testsuite Expressions
topsort Exp

test simple addition
"2 + 3" \rightarrow \text{Plus(Int("2"), Int("3"))}

test addition is left associative
"1 + 2 + 3" \rightarrow \text{Plus(Plus(_, _), _)}

test for lazy people
"1 + 2 + 3" succeeds

file large.exp succeeds

test
"x1" fails
SGLR Implementation and Invocation

```
parse-parse-testsuite -i Exp.testsuite |
  parse-unit -p Exp.tbl --verbose 1

executing testsuite Expressions with 5 tests

* OK : test 1 (simple addition)
* OK : test 2 (addition is left associative)
* OK : test 3 (for lazy people)
* OK : test 4 (large.exp)
sglr: error in d_0.tmp, line 1, col 2: character ‘1’ (\x31) unexpected
* OK : test 5 (x1)

results testsuite Expressions
successes : 5
failures : 0
```
Parse-unit is the Silver Bullet

- concise *overview* of input and expected result
- check results at different *levels of detail*
- *no escaping* of ‘special’ characters required
- *file* and *inline* input
- *reusable* for different parser implementations
- enables *reasoning* about testsuites
Future Work

• **coverage** of parse-testsuites
  - rule coverage
  - context-based branch coverage (work of Ralf Lämmel)

• **import** mechanism for modular testsuites

• different parsers
  - bison/glr producing aterms

• sglr specific features (attributes)
  - ambiguities
  - statistics

• **apath** based result checking
StrategoBox

Stratego Rules, also for Pretty-Printing
GPP: Generic Pretty Printer

- *pretty-printing* of parse and abstract syntax trees

- *box*
  - text layout language
  - output format independent
    - abox2text, abox2html, abox2latex

- *pretty-print table*
  - map constructor names to box templates
  - applied by ast2abox
  - can be generated from SDF2 syntax definition
Pretty-Print Table Example

<table>
<thead>
<tr>
<th>Module</th>
<th>V[\texttt{KW[&quot;module&quot;] 1} 2],</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module.2:iter-star</td>
<td>_1,</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors</th>
<th>V is=2  [H [\texttt{KW[&quot;constructors&quot;]}]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A (1 hs=1, 1 hs=1, 1 hs=1) [__1]],</td>
</tr>
<tr>
<td>Constructors.1:iter-star</td>
<td>_1,</td>
</tr>
</tbody>
</table>

| OpDecl    | R [__1 \texttt{KW[":"]} H hs=1 [__2]], |
| OpDeclInj | R ["" \texttt{KW[":"]} H hs=1 [__1]], |

| Match      | H hs=0[\texttt{KW["?"] 1}], |
| Build      | H hs=0[\texttt{KW["!"] 1}], |

| ScopeDefault | H hs=0[\texttt{KW["{" _1 KW["}]"}]], |
| Scope       | H hs=0[\texttt{KW["{" V[H[__1 KW[":"]]} 2] KW["}]"}], |
| Scope.1:iter-star-sep | H hs=0[__1 KW[","]], |
Problem

```c
if (bar)
{
    foo
}
else if (bar)
{
    foo
} ...
```

```c
if (bar)
{
    foo
}
else if (bar)
{
    foo
} ...
```

Pretty print rule for If:

```
If -- V vs=0 [ 
    H hs=0 [KW["if"] "(" _1 ")"]
    _2
    KW["else"]
    _3
] 
```
Pretty-Print Rules

- pretty-print table: \textit{selection} of pp rules by constructor name
  - no number of children
  - no patterns
  - no conditions
  - no context

- solution: StrategoBox
  - \textit{embed box} in Stratego
    \[ \Rightarrow \text{meta-programming with concrete object syntax} \]

- advantages
  - \textit{pattern-matching} and \textit{conditions}
  - \textit{strategy} controls the application of pp rules
No Problem

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
  \[KW["else"] b5 \]
]
Quick Introduction: Compiling

```plaintext
module pretty-print
imports Box ...
```

**pretty-print.meta**

```plaintext
Meta([Syntax("Stratego-Box")])
```

**by hand**

```plaintext
strc -i pretty-print.str -I ${GPP}/share/sdf/gpp \
- I ${GPP}/share/gpp
```

**using Makefile.xt**

```plaintext
STRINCLUDES = -I $(GPP)/share/sdf/gpp -I $(GPP)/share/gpp
```
Quick Introduction: Implementation

- *quotation* of Stratego to Box: choose
- *anti-quotation* of Box to Stratego: $\sim$ or $\sim^*$

\[
\begin{align*}
\text{expr-to-box: Plus}(e_1, e_2) & \rightarrow \quad \text{H} \hspace{1em} \text{hs}=1 \quad [\sim e_1 \ "+" \sim e_2] \\
\text{expr-to-box: Plus}(e_1, e_2) & \rightarrow \quad [\text{H} \hspace{1em} \text{hs}=1 \quad [\sim e_1 \ "+" \sim e_2] ] | \\
\text{expr-to-box: Plus}(e_1, e_2) & \rightarrow \text{box} \quad [\text{H} \hspace{1em} \text{hs}=1 \quad [\sim e_1 \ "+" \sim e_2] ] | \\
\end{align*}
\]

\[
\begin{align*}
\text{java-to-box:} \\
\quad \text{Try}(\text{block}, \text{catches}, \text{finally}) & \rightarrow \\
\quad \text{V} \hspace{1em} \text{vs}=0 \quad [\text{KW}["try"] \sim \text{block} \sim \ast \text{catches} \quad \text{KW}["finally"] \sim \text{finally}] \\
\end{align*}
\]
Quick Introduction: Implementation

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

java-to-box:
    Try(b1, b2*, b3)
    ->
    V vs=0 [KW["try"] b1 b2* KW["finally"] b3]

java-to-box:
    SuperField(s) -> H hs=0 [KW["super"] "." s]

More examples:

- `jtree2abox` in java-front/pp
- `xml-doc2abox` in xml-tools/pp
- `pp-aterm` in aterm-tools/pp
Future Work

- make using multiple embeddings more easy
- separate reusable parts in pp library
  - block structure
  - lists and separators
  - Box specific traversals
- configuration of pretty-printers
- derive Box expressions from ‘example’
- applications
  - pretty-print concrete object syntax fragments
Application of Abstract Syntax Definitions
Grammars as Contracts

Diagram showing the relationship between source code, concrete syntax definition, abstract syntax definition, parser, and pretty printer.
Abstract Syntax Definitions in Stratego/XT

*Stratego/XT: abstract syntax trees*
- Stratego signature: abstract syntax definition
- generated from SDF2 concrete syntax definition

Do we use it?
- No, even format checkers are written by hand!

Why not?
- every generated signature is incorrect!
  - lexical syntax, injections, aliases, renamings
- no separate language

Solution: stratego-regular
ATerm: Tree Languages and Grammars

- tree language – subset of terms over *ranked alphabet*
- aterm application – *fixed number* of children

**regular tree grammar**

```plaintext
start  Section
productions
  Section -> section (Title?, {Para})
  Title  -> title  (<string>)
  Para   -> para   (<string>)
```

**regular tree grammar**

```plaintext
start  Exp
productions
  Exp   -> Plus (Exp, Exp)
  Exp   -> Call (Var, {Exp})
  Exp   -> Var
  Var   -> Var   (<string>)
```
XML: Hedge Languages and Grammars

- *hedge* – sequence of trees
- children of xml element – sequence of *varying length*

**regular hedge grammar**

```
start Section
productions
Section → section (Title? Para*)
Title → title (<string>)
Para → para (<string>)
```

```
start Exp
productions
Exp → Plus (Exp Exp)
Exp → Call (Var Exp*)
Exp → Var
Var → Var (<string>)
```
Application: Code Generation

- **sdf2rtg** – generate rtg from SDF
  
  sdf2rtg -i Example.def -m Example

- **rtg2sig** – generate Stratego Signature from rtg
  
  rtg2sig -i Example.rtg --module Example

- **rtg2typematch** – generate type predicate strategies
  
  parse-rtg|rtg2typematch --module Example

widely used: java-front, sdf-front, stratego-shell, xml-tools
Application: Validation

history:

- developed *fc-gen* in 2002
- Stratego signature input ⇒ nobody uses it

stratego-regular:

- *wf-checker*: a term in the language of an rhg
- no need to write a format checker by hand
- to do
  - rename to format-checker
  - define subsets of an rtg?
  - generate code by partial evaluation?
Application: Exchange

exchange

→ from xml systems invoke aterm tools
← invoke xml tools from aterm systems

problem: differences between xml and aterm
  • aterm has a more explicit structure
  • aterm has primitive data types
  • aterm has structured annotations

solution: use same abstract syntax definition for xml and aterm
  • rtg for aterm
  • rhg for xml
Application: Exchange

aterm to xml

- drop explicit structure
- data2xml-doc | pp-xml-doc

xml to aterm

- add explicit structure
- xml-interpret --rhg Example.rhg

interoperability

→ aterm tools as xml tools using generic data2xml-doc
← xml tools as aterm tools using xml-interpret