The Life Cycle of Grammarware

CWI Scientific Meeting
Vadim Zaytsev, SWAT, CWI
2012
Grammarware
@grammarware

Vadim Zaytsev
language engineering freak, university maniac, programmer, hacker, automation enthusiast, wiki addict, grammar nazi, blues fan
Yurup · http://grammarware.net

Tweets

Vadim Zaytsev @grammarware
Usually experimental hacking is followed by experimental development: my tool needs to produce executable artefacts instead of text. 4h

Vadim Zaytsev @grammarware
Experimental hacking phase is done: my algorithm does what is expected from it. Now back to making slides and rehearsing tomorrow’s talks! 4h

Paul Klint @PaulKlint
10% budget cut on Dutch research. Dutch politicians forget that innovation is the source of prosperity! ow.ly/1FmpM5 7h

Vadim Zaytsev @grammarware
@zef @guwac finally, the complaints of my roommates @DavyLandman & @hilisma were heard, and I’m being replaced to @tvdstorm’s room. In reply to Zef Hemel 7h

Vadim Zaytsev @grammarware
The way I use it, a tablet is an extremely private device. Mail inbox & the to-do list with ideas on future papers right on the main screen! 7h

Vadim Zaytsev @grammarware
That’s what I always say: don’t make jokes on twitter! thedailywh.at/2012/01/30/thi 21h

Vadim Zaytsev @grammarware
Good bye, @cwini L224, you’ve been a great home for more than a year! (@ Centrum Wiskunde & Informatica (CWI) [pic] 4sq.com/y757Fe 21h
Software Languages
Language: make

all:
  make clean
  make build
  make test

test:
  ./converge.py master.bgf base/

build:
  cp ../convergence/fl/snapshot/*.bgf tests/
  rsc2bgf ../fl/rascal1/FL.rsc tests/rascal.bgf
  ls -1 tests/*.bgf | xargs -n1 ./testperform
  cp normal/*.normal.bgf base/

clean:
  rm -f tests/*.bgf xbgf/*.xbgf normal/* base/*
import types.*;
import org.antlr.runtime.*;
import java.io.*;
public class TestEvaluator {
    public static void main(String[] args) throws Exception {
        ANTLRFileStream input = new ANTLRFileStream(args[0]);
        FLLexer lexer = new FLLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        FLParser parser = new FLParser(tokens);
        Program program = parser.program();
        input = new ANTLRFileStream(args[1]);
        lexer = new FLLexer(input);
        tokens = new CommonTokenStream(lexer);
        parser = new FLParser(tokens);
        Expr expr = parser.expr();
        Evaluator eval = new Evaluator(program);
        int expected = Integer.parseInt(args[2]);
        assert expected == eval.evaluate(expr);
    }
}
<?xml version="1.0" encoding="UTF-8"?>
<bgf:grammar xmlns:bgf="http://planet-sl.org/bgf">
  <root>Program</root>
  <root>Fragment</root>
  <bgf:production>
    <nonterminal>Program</nonterminal>
    <bgf:expression>
      <plus>
        <bgf:expression>
          <selectable>
            <selector>function</selector>
            <bgf:expression>
              <nonterminal>Function</nonterminal>
            </bgf:expression>
          </selectable>
        </bgf:expression>
      </plus>
    </bgf:expression>
  </bgf:production>
</bgf:grammar>
Language: SDF

context-free syntax

Function+  -> Program
Name Name+ "=" Expr Newline+ -> Function
Expr Ops Expr  -> Expr {left, prefer, cons(binary)}
Name Expr+  -> Expr {avoid, cons(apply)}
"if" Expr "then" Expr "else" Expr  -> Expr {cons(ifThenElse)}
"(" Expr ")"  -> Expr {bracket}
Name  -> Expr {cons(argument)}
Int  -> Expr {cons(literal)}

"-"  -> Ops {cons(minus)}
"+"  -> Ops {cons(plus)}
"=="  -> Ops {cons(equal)}
Language: Ecore

- **Program**
  - function : Function
- **Function**
  - name : EString
  - argument : Argument
  - definition : Exp
- **Argument**
  - name : EString
- **LiteralExp** → Exp
- **ArgumentExp** → Exp
- **IfThenElseExp** → Exp
- **ApplyExp** → Exp
- **BinaryExp** → Exp
- **PlusExp** → BinaryExp
- **MinusExp** → BinaryExp
- **EqualExp** → BinaryExp

- **IfThenElseExp** → Exp
  - if : Exp
  - then : Exp
  - else : Exp

- **ApplyExp** → Exp
  - function : Function
  - argument : Exp

- **BinaryExp** → Exp
  - left : Exp
  - right : Exp
Language: syntax diagram

```
[NO] STICKY-LINKAGE "integer" STICKY-LINKAGE
```

```
[NO] STICKY-PERFORM
```

Formal Languages
Known from theory (1/2)

- What is a language?
  - set of allowed words
  - often requires set comprehension to define
  - infinite for realistic cases

- How to document a software language?

- How to express language evolution?
What is a grammar?
- a set of nonterminals
- a set of terminals
- a set of production rules
- a start symbol

What is a good grammar?

How to combine or decompose grammars?
Software languages

- General purpose programming languages
- Domain specific languages
- Modelling and metamodelling languages
- Data description languages, data models, schemata
- Ontologies
- APIs and libraries
Grammarware

- Parser
- Compiler
- Interpreter
- Pretty-printer
- Scanner
- Browser
- Static checker
- Structural editor
- IDE
- DSL framework
- Preprocessor
- Postprocessor
- Model checker
- Refactorer
- Code slicer
- API
- XMLware
- Modelware
- Language workbench
- Reverse engineering tool
- Benchmark
- Recommender
- Renovation tool
Life Cycle
Grammar recovery

• Given is an artefact containing grammar knowledge:
  • a grammar
  • a parser specification
  • a metamodel
  • grammarware source code
  • a data schema
  • documentation
• Question: how to extract a grammar from it?
• Answer: with tolerant grammar recovery techniques!
## Extraction of Java grammars

<table>
<thead>
<tr>
<th></th>
<th>impl1</th>
<th>impl2</th>
<th>impl3</th>
<th>read1</th>
<th>read2</th>
<th>read3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary lexical decisions</td>
<td>2</td>
<td>109</td>
<td>60</td>
<td>1</td>
<td>90</td>
<td>161</td>
<td>423</td>
</tr>
<tr>
<td>Well-formedness violations</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>Indentation violations</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>Recovery rules</td>
<td>3</td>
<td>12</td>
<td>18</td>
<td>2</td>
<td>59</td>
<td>47</td>
<td>141</td>
</tr>
<tr>
<td>○ Match parentheses</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>○ Metasymbol to terminal</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>27</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>○ Merge adjacent symbols</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>○ Split compound symbol</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>○ Nonterminal to terminal</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>11</td>
<td>29</td>
</tr>
<tr>
<td>○ Terminal to nonterminal</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>13</td>
<td>33</td>
</tr>
<tr>
<td>○ Recover optionality</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Purge duplicate definitions</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>51</td>
</tr>
<tr>
<td>Total</td>
<td>11</td>
<td>123</td>
<td>92</td>
<td>24</td>
<td>181</td>
<td>238</td>
<td>669</td>
</tr>
</tbody>
</table>

---

Notation-parametric recovery

- Compose a notation specification
- Perform a robust heuristic-based recovery process
- Successful for grammars of Ada, C, C++, C#, Dart, Eiffel, Modula, MediaWiki, LLL, EBNF, etc.

Zaytsev, What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions, SAC/PL 2012.
Language evolution

• Given is a (baseline) grammar

• The grammar needs to be engaged in:
  • correction
  • evolution
  • adaptation
  • beautification

• Question: how to apply stable, disciplined, reproducible, automated, possibly bidirectional transformations?

• By programmable grammar transformations!
XBGF transformation suite

- Semantics-preserving operators:
  - fold, unfold, rename, factor, massage, ...
- Semantics-increasing/decreasing operators:
  - appear/disappear, narrow/widen, add/remove, ...
- Semantics-revising operators
  - inject, permute, replace, redefine, ...

Zaytsev, BGF transformation operator suite v.1.0, July 2010.
Language convergence

- Given are several grammars of related languages
- We want to investigate their relationships:
  - equivalent
  - dialects
  - backward compatible versions
- Question: how to reverse engineer grammar relationships?
- With grammar convergence!
Convergence tree

Convergence graph

Metalanguage evolution

- Given is a grammar written in a specific notation
- This notation can be different:
  - lexically
  - semantically
  - expressively
- Question: how to migrate grammars to another notation?
- By coupled evolution of syntax and metasyntax!
Coupled evolution megamodel

Matching metamodel entities

• Given are two grammars that model the “same” language
• Grammars have entities (e.g., nonterminals) that have:
  • different names
  • different structure
• Question: how to match such entities?
• With grammar-based combinatorial differential testing!
Visualised nonterminal matching
Language documentation

- Given are:
  - a grammar for a software language
  - explanations in a natural language
  - executable code samples
  - known relationships between concepts
- Question: how to do language documentation properly?
- Answer: by generating it from structured data!
## Unified model for language docs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>synopsis</td>
<td>~</td>
<td>~</td>
<td>synopsys</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>description</td>
<td>~</td>
<td>~</td>
<td>motivation</td>
<td>definition</td>
<td>usage</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>syntax</td>
<td>— &lt;sup&gt;a&lt;/sup&gt;</td>
<td>syntax</td>
<td>structure</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>[NN]&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>constraints</td>
<td>~</td>
<td>constraints</td>
<td>applicability</td>
<td>errors</td>
<td>restrictions</td>
<td>~</td>
<td>~</td>
<td>constraints</td>
</tr>
<tr>
<td>references</td>
<td>~</td>
<td>~</td>
<td>related patterns</td>
<td>references</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>relationship</td>
<td>~</td>
<td>~</td>
<td>consequences</td>
<td>return value, refinement</td>
<td>related</td>
<td>return type</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>semantics</td>
<td>~</td>
<td>semantics</td>
<td>collaborations</td>
<td>~</td>
<td>important</td>
<td>~</td>
<td>semantics</td>
<td>~</td>
</tr>
<tr>
<td>rationale</td>
<td>~</td>
<td>notes</td>
<td>implementation</td>
<td>rationale</td>
<td>GLS, ES&lt;sup&gt;c&lt;/sup&gt;</td>
<td>note</td>
<td>rationale</td>
<td>note</td>
</tr>
<tr>
<td>example</td>
<td>examples</td>
<td>examples</td>
<td>sample code, known uses</td>
<td>~</td>
<td>~</td>
<td>example</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>update</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>default</td>
<td>~</td>
<td>~</td>
<td>note</td>
<td>default values</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>value</td>
<td>~</td>
<td>~</td>
<td>also known as</td>
<td>conforms to</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>list</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>messages, parameters</td>
<td>terminals</td>
<td>~</td>
<td>properties</td>
<td>~</td>
</tr>
<tr>
<td>section</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>subtopic</td>
<td>~</td>
<td>types</td>
<td>participants</td>
<td>~</td>
<td>fields</td>
<td>~</td>
<td>operations</td>
<td>functions</td>
</tr>
</tbody>
</table>

### Coverage of LDF

![Coverage of LDF](image)

Bibliography


- Vadim Zaytsev, *What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions*, SAC/PL 2012, March 2012.


Discussion