Grammar Composition & Extension

vadim@grammarware.net

Symposium on Language Composition and Modularity
Vadim Zaytsev, SWAT, CWI
CC BY 2012
Introduction to grammarware
What is a grammar?

• Structural description in software systems
• Description of structures used in software systems
Grammar use case

• Abstract use case
  • syntax definition
  • exchange format
  • interaction protocol
  • data model
  • domain model
  • metamodel
  • …

• Concrete use case
  • parsing
  • serialisation
  • renovation
  • refactoring
  • static analysis
  • reengineering
  • …

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
Grammars everywhere
Grammarware
Grammarware
Grammarware
Grammarware
Grammarware
Grammar decomposition
Quo usque tandem?

- Lexical syntax:
  - character level (tokenisation)
  - block level (indentation)
- Preprocessing syntax
  - comments
  - directives
- Base syntax
- Syntax highlighting
- Processing order
- Filtering / disambiguation
- Error handling
- Tree construction
- AST format

http://commons.wikimedia.org/wiki/File:Cicero.PNG
Grammar composition
Behind the Screen / Unknown Chaplin

Fair (re)use, picture is the courtesy of chaplin.bfi.org.uk.
Grammar composition

syntax A = B+;
syntax B = “b”;
syntax D = C*;
syntax C = “c”;
syntax A = B+;
syntax D = C*;
syntax B = “b”;
syntax C = “c”;
Grammar composition

- syntax $A = B \ C$;
- syntax $B = "b"$;
- syntax $D = C \ B$;
- syntax $C = "c"$;
Grammar composition

```plaintext
syntax A = B C;
syntax B = "b";
syntax C = "c";
```

```plaintext
syntax A = B C;
syntax B = "b";
syntax C = "c";
```

{"bc"}
Grammar composition

- syntax $A = B \ C$;
- syntax $B = "b"$;
- syntax $C = "c"$;

- syntax $A = C \ B$;
- syntax $B = "c"$;
- syntax $C = "b"$;

\{"bc",
  "cb"\}
Grammar composition

```
syntax A = B | CA;
syntax B = "b";
syntax C = "c";
```

```
syntax A = B | CB;
syntax B = "b";
syntax C = "c"+
```

\{"c^n b"\},
n=0,1,2,...
Grammar composition

syntax A = B | BA;
syntax B = “b”;

syntax A = AB | B;
syntax B = “b”;

\{“b^n”\},
n=1,2,…
Grammar composition

syntax A = B | C;
syntax C = D+;
syntax ... = ε;

syntax C = E;
syntax E = B+;
syntax ... = ε;

XML
Grammar composition

syntax $A = B \, C \{\ldots\};$

syntax $A = B \, C \{\ldots\};$
Grammar composition

syntax A = B C;
syntax B = “b”;
syntax C = “c”;
syntax C ≠ “c”;
syntax C = D E;
Adjacent topics:
pgt gc ngt no
Programmable Grammar Transformations
Programmable G xformation

Ad hoc megamodel shown at IPA Spring Days
Programmable G xformation

- known semantics, well-defined algorithm
- rename, fold, factor, inject, remove, …

Ad hoc megamodel shown at IPA Spring Days
Programmable G xformation

Arguments

- what exactly to rename/factor/inject/…?
Input grammar

- determines applicability
Programmable G xformation

Ad hoc megamodel shown at IPA Spring Days
XBGF Operator Suite

- Semantic-preserving operators
  - fold, unfold, extract, inline, massage, factor, deyaccify, …
- (Some) semantic-preserving operators
  - permute, abstractize, concretize, designate, anonymize
- Language-increasing operators
  - add, appear, widen, upgrade, unite
- Language-decreasing operators
  - remove, disappear, narrow, downgrade
- Revising operators
  - redefine, inject, project, replace, …

V. Zaytsev, BGF Transformation Operator Suite v.1.0, online, 2010.
References


Guided Grammar Convergence
The most trivial case

- Equal grammars

- Algebraically equivalent grammars

- Nothing to do here

Structural resolution

- Nonterminal vs. value
  - A vs. string
- Sequence permutations
  - A B B A vs. B B A A
- Lists of symbols
  - A* vs. A+
- Separator lists… irrelevant

Nominal resolution

<table>
<thead>
<tr>
<th>Production rule in the master grammar</th>
<th>Production signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 = p(‘’, \mathtt{program, +})(\mathtt{function}) )</td>
<td>{\langle \mathtt{function, +} \rangle}</td>
</tr>
<tr>
<td>( p_2 = p(‘’, \mathtt{function, seq([str, +(str), expr])} )</td>
<td>{\langle \mathtt{expr, 1} \rangle, \langle \mathtt{str, 1+} \rangle}</td>
</tr>
<tr>
<td>( p_3 = p(‘’, \mathtt{expr, str}) )</td>
<td>{\langle \mathtt{str, 1} \rangle}</td>
</tr>
<tr>
<td>( p_4 = p(‘’, \mathtt{expr, int}) )</td>
<td>{\langle \mathtt{int, 1} \rangle}</td>
</tr>
<tr>
<td>( p_5 = p(‘’, \mathtt{expr, apply}) )</td>
<td>{\langle \mathtt{apply, 1} \rangle}</td>
</tr>
<tr>
<td>( p_6 = p(‘’, \mathtt{expr, binary}) )</td>
<td>{\langle \mathtt{binary, 1} \rangle}</td>
</tr>
<tr>
<td>( p_7 = p(‘’, \mathtt{expr, cond}) )</td>
<td>{\langle \mathtt{cond, 1} \rangle}</td>
</tr>
<tr>
<td>( p_8 = p(‘’, \mathtt{apply, seq([str, +(expr)])} )</td>
<td>{\langle \mathtt{expr, +}, \langle \mathtt{str, 1} \rangle}</td>
</tr>
<tr>
<td>( p_9 = p(‘’, \mathtt{binary, seq([expr, operator, expr])}))</td>
<td>{\langle \mathtt{expr, 11}, \langle \mathtt{operator, 1} \rangle}</td>
</tr>
<tr>
<td>( p_{10} = p(‘’, \mathtt{cond, seq([expr, expr, expr])}) )</td>
<td>{\langle \mathtt{expr, 111} \rangle}</td>
</tr>
</tbody>
</table>

**Table 1.** Production rules of the master grammar for FL, with their production signatures.

Definitions

- Nonterminal footprint
- Production signature
- Prodsig-equivalence
- Weak prodsig-equivalence
- Nominal resolution

## Nominal resolution example

<table>
<thead>
<tr>
<th>Production rule</th>
<th>Production signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 ) = ( p('', \text{Fragment}, \text{Expr}) )</td>
<td>( { \langle \text{Expr}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_2 ) = ( p('', \text{Program}, + (\text{Function})) )</td>
<td>( { \langle \text{Function}, + \rangle } )</td>
</tr>
<tr>
<td>( q_3 ) = ( p('', \text{Function}, \text{seq}([\text{str}, +(\text{str}), \text{Expr}])) )</td>
<td>( { \langle \text{str}, 1+ \rangle, \langle \text{Expr}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_4 ) = ( p('', \text{Expr}, \text{int}) )</td>
<td>( { \langle \text{int}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_5 ) = ( p('', \text{Expr}, \text{str}) )</td>
<td>( { \langle \text{str}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_6 ) = ( p('', \text{Expr}, \text{Expr}_1) )</td>
<td>( { \langle \text{Expr}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_7 ) = ( p('', \text{Expr}, \text{Expr}_2) )</td>
<td>( { \langle \text{Expr}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_8 ) = ( p('', \text{Expr}, \text{Expr}_3) )</td>
<td>( { \langle \text{Expr}, 1 \rangle } )</td>
</tr>
<tr>
<td>( q_9 ) = ( p('', \text{Expr}_1, \text{seq}([\text{Ops}, \text{Expr}, \text{Expr}])) )</td>
<td>( { \langle \text{Ops}, 1 \rangle, \langle \text{Expr}, 11 \rangle } )</td>
</tr>
<tr>
<td>( q_{10} ) = ( p('', \text{Expr}_2, \text{seq}([\text{Expr}, \text{Expr}, \text{Expr}])) )</td>
<td>( { \langle \text{Expr}, 111 \rangle } )</td>
</tr>
<tr>
<td>( q_{11} ) = ( p('', \text{Expr}_3, \text{seq}([\text{str}, +(\text{Expr})])) )</td>
<td>( { \langle \text{str}, 1 \rangle, \langle \text{Expr}, + \rangle } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prerequisite</th>
<th>Match</th>
<th>( p_i \odot q_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>roots</td>
<td>( p_1 \neq q_1 )</td>
<td>{ \langle \text{program, Fragment} \rangle }</td>
</tr>
<tr>
<td>roots</td>
<td>( p_1 \equiv q_1 )</td>
<td>{ \langle \text{program, Program} \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{str}, \text{str} \rangle } )</td>
<td>( p_2 \equiv q_3 )</td>
<td>{ \langle \text{function, Function} \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle, \langle \text{str}, \text{str} \rangle } )</td>
<td>( p_3 \equiv q_5 )</td>
<td>{ \langle \text{str, expr, Expr} \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle } )</td>
<td>( p_4 \equiv q_4 )</td>
<td>{ \langle \text{int, int} \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle, \langle \text{str}, \text{str} \rangle } )</td>
<td>( p_5 \equiv q_8 )</td>
<td>{ \langle \text{apply, Expr}_3 \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle } )</td>
<td>( p_6 \equiv q_6 )</td>
<td>{ \langle \text{binary, Expr}_1 \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle } )</td>
<td>( p_7 \equiv q_9 )</td>
<td>{ \langle \text{operator, Ops} \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle } )</td>
<td>( p_8 \equiv q_{11} )</td>
<td>{ \langle \text{cond, Expr}_2 \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle } )</td>
<td>( p_9 \equiv q_9 )</td>
<td>{ \langle \text{cond, Expr}_2 \rangle }</td>
</tr>
<tr>
<td>( { \langle \text{expr, expr} \rangle } )</td>
<td>( p_{10} \equiv q_{10} )</td>
<td>{ \langle \text{cond, Expr}_2 \rangle }</td>
</tr>
</tbody>
</table>

**Table 2.** On the left: production rules of the servant grammar for FL, derived from the XML schema, with their production signatures. On the right: the process of derivation of the nominal resolution relation \( p_i \odot q_j \). Note how two hypotheses must be formed and one of them rejected, because this servant grammar has two roots and both need to be checked for prodsig-equivalence with the root of the master grammar. Other than that, all production rules are matched with strong equivalence.
Nominal resolution example

<table>
<thead>
<tr>
<th>Production rule</th>
<th>Production signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1=p(\text{<code>}, \text{Program}, + (\text{Function})\text{</code>})$</td>
<td>{{\text{Function}, +}}</td>
</tr>
<tr>
<td>$r_2=p(\text{<code>}, \text{Function}, \text{seq}([\text{Name}, + (\text{Name}), \text{Expr}, + (\text{CR})])\text{</code>})$</td>
<td>{{\text{CR}, +}, {\text{Expr}, 1}, {\text{Name}, 1+}}</td>
</tr>
<tr>
<td>$r_3=p(\text{<code>}, \text{Expr}, \text{Expr}_1\text{</code>})$</td>
<td>{{\text{Expr}_1, 1}}</td>
</tr>
<tr>
<td>$r_4=p(\text{<code>}, \text{Expr}, \text{Expr}_2\text{</code>})$</td>
<td>{{\text{Expr}_2, 1}}</td>
</tr>
<tr>
<td>$r_5=p(\text{<code>}, \text{Expr}, \text{Expr}_3\text{</code>})$</td>
<td>{{\text{Expr}_3, 1}}</td>
</tr>
<tr>
<td>$r_6=p(\text{<code>}, \text{Expr}, \text{Name}\text{</code>})$</td>
<td>{{\text{Name}, 1}}</td>
</tr>
<tr>
<td>$r_7=p(\text{<code>}, \text{Expr}, \text{Int}\text{</code>})$</td>
<td>{{\text{Int}, 1}}</td>
</tr>
<tr>
<td>$r_8=p(\text{<code>}, \text{Expr}_1, \text{seq}([\text{Expr}, \text{Ops}, \text{Expr}])\text{</code>})$</td>
<td>{{\text{Ops}, 1}, {\text{Expr}, 11}}</td>
</tr>
<tr>
<td>$r_9=p(\text{<code>}, \text{Expr}_2, \text{seq}([\text{Name}, + (\text{Expr})])\text{</code>})$</td>
<td>{{\text{Expr}, +}, {\text{Name}, 1}}</td>
</tr>
<tr>
<td>$r_{10}=p(\text{<code>}, \text{Expr}_3, \text{seq}([\text{Expr}, \text{Expr}, \text{Expr}])\text{</code>})$</td>
<td>{{\text{Expr}, 111}}</td>
</tr>
</tbody>
</table>

Table 3. On the left: production rules of the servant grammar for FL, derived from a corresponding SDF syntax definition, with their production signatures. On the right: the process of derivation of the nominal resolution relation $p_i \diamond r_j$. Note how a special lexical nonterminal for CR nonterminal remains unmatched due to weak equivalence of production rules that contain it.
Abstract Normal Form

(1) lack of labels for production rules
(2) lack of named subexpressions
(3) lack of terminal symbols
(4) maximal outward factoring of inner choices
(5) lack of horizontal production rules
(6) lack of separator lists
(7) lack of trivially defined nonterminals (with $\alpha$, $\epsilon$ or $\varphi$)
(8) no mixing of chain and non-chain production rules
(9) the nonterminal call graph is connected, and its top nonterminals are the starting symbols of the grammar

V. Zaytsev, Guided Grammar Convergence, CoRR abs/1207.6541, draft for 2013.
Grammar design mutation

- Deyaccification

- $B = CB | C$ vs. $B = C^+$

- Layers vs. priorities

- $X = \ldots | Y; Y = \ldots | X;$ vs $X = \ldots | \ldots$;

- Associativity

- $A OA$ vs. $A (OA)^*$
Unresolved

- Aggregation

<table>
<thead>
<tr>
<th>Master grammar</th>
<th>Ecore</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp: STR exp⁺</td>
<td>ApplyExp: Function Exp⁺</td>
</tr>
</tbody>
</table>

- Meaningful chain rules

<table>
<thead>
<tr>
<th>Master grammar</th>
<th>Ecore</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp: exp op exp</td>
<td>BinaryExp:</td>
</tr>
<tr>
<td></td>
<td>PlusExp</td>
</tr>
<tr>
<td></td>
<td>MinusExp</td>
</tr>
<tr>
<td></td>
<td>EqualExp</td>
</tr>
<tr>
<td></td>
<td>Exp Exp</td>
</tr>
<tr>
<td></td>
<td>Exp Exp</td>
</tr>
<tr>
<td></td>
<td>Exp Exp</td>
</tr>
</tbody>
</table>
Grammar
transformation
composition
Adaptation through tolerance
Adaptation through tolerance

Adaptation through tolerance

Adaptation through tolerance

Adaptation through adjustment

Adaptation through adjustment

Adaptation through adjustment

Adaptation through adjustment

Grammar transformations

• Suite of well-defined well-studied operators
• Partial evaluation of transformation operators
• Classic grammar transformation:
  • inapplicable? error! halt!
  • vacuous? error! halt!
  • transform! next!
Negotiated transformations

- Negotiated grammar transformation:
  - applicable & non-vacuous? transform! next!
  - vacuous? suggest to do nothing!
  - not applicable? suggest better arguments!
  - keep negotiating until applicability or surrender
Negotiated transformations

- Variability limits as a part of transformation command
- Interactive transformation (ask the user)
- Display a warning and proceed with minimal adjustment
- Proceed with one, save other options for fallback
- Halt and recomment
- ...
To summarise

- Grammars define structure
- Grammarware works on grammars & languages
- Too much stuff in the grammar
- Decomposition
- Composition
- Adjacent topics?
Questions?

vadim@grammarware.net