Press PgDn to start.
TOWARDS AN ENGINEERING DISCIPLINE
FOR GRAMMARWARE

Ralf Lämmel, VU und CWI, Amsterdam
GRAMMARWARE — DEFINITION

The term ‘grammarware’ comprises grammars and grammar-driven software, i.e., software artifacts that directly involve grammar knowledge.
SOFTWARE RE-/REVERSE ENGINEERING NEEDS ENGINEERING OF GRAMMARWARE — THINK OF TRANSFORMATION ENVIRONMENTS
SOFTWARE RE-/REVERSE ENGINEERING NEEDS GRAMMAR ENGINEERING — THE EFFORT DISTRIBUTION

(length of arrow represents effort)
SO MUCH GRAMMARWARE, SO LITTLE ATTENTION

Grammar-dependent software

- Compiler
- Analysis tooling
- Application generators
- Client/server applications
- Weaving tools
- Industrial standards
- Modification tooling
- Reflection APIs
- CASE tools
- Document processors
- Software models

Actual grammar notations

- DGL
- PRECC
- ASDL
- YACC
- SDF
- CTS.NET
- XML schema

Principle grammar forms

- BNF
- EBNF
- Class dictionary
- Signature
**WHAT IS ENGINEERING OF GW?**

**LET’S LOOK AT SCENARIOS.**

- As a developer of database applications, you want to make a transition to a *new screen definition language*.

- As a developer of Commercial Off-The-Shelf software, you want to *import user profiles* used by competing products; think of web browsers.

- As an object-oriented application developer, you want to systematically *migrate to a new release of some API*.

- As a developer of an inhouse domain-specific language (DSL), you want to provide a tool to *convert existing DSL programs* to the evolved DSL.

- As an online service provider, you want to support new import/export formats (maybe using XML schemas) or new protocols for system use.
GRAMMAR FORMS AND NOTATIONS

- Definitions of interchange and storage formats
- Interface descriptions
- Specifications of interaction protocols
- Concrete and abstract syntax definitions
- Definitions of intermediate and internal representations
GRAMMAR-DRIVEN SOFTWARE

- Application generators, tools for generative programming, aspect-oriented weavers, tools for automated software engineering
- Distributed or component-based applications where grammars occur in the sense of interfaces and I/O formats
- Functionality in language implementations, e.g., compilers, animators, documentation generators, profilers, debuggers, ...
- Functionality for automated software re-documentation, analysis and modification, pre- and post-processors
- Reference manuals, style guides, and industrial standards
GRAMMARWARE REALITY — HACKING

Given the omnipresence of grammarware, one may expect that grammarware is treated as an engineering artifact — subject to reasonable common or best practices. In reality, grammarware is predominantly treated in an *ad-hoc* manner.
EXAMPLE PARSER DEVELOPMENT

"Grammarware hacking"

Grammar knowledge

Manual coding

"Grammarware engineering"

Automated recovery

Grammar specification

Automated generation

tooling

Parser

Parser

Parser

...
LACK OF BEST PRACTICES

- There is no established approach for performing grammar evolution in a traceable manner — not to mention the even more difficult problem of co-evolution of grammar-driven software.

- There is no established approach for maintaining consistency between the incarnations of conceptually the same grammar.

- Grammars are immediately implemented using specific technology, which implies the use of idiosyncratic notations.
Lack of comprehensive foundations

- no comprehensive theory of testing grammarware
- ... of transforming ...
- version management?
- quality assessment?
- design patterns?
- debugging?
THE GRAMMAR DILEMMA

Improving on hacking sounds like such a good idea! Why did it not happen?

Myth “Grammars are a buried subject”
Myth “Engineering of grammarware = parser generation”
Myth “XML is the answer”
Myth “Grammarware is all about programming languages”
IN NEED OF A PARADIGM SHIFT

Perspective "Lines of code"  Perspective "Impact ratio"

All software assets

Grammars in compiler frontends

Grammars in the widest sense

Fig.: The mythical view and the proposed view on grammarware
ENGINEERING OF GRAMMARWARE — PRINCIPLES

• Abstraction
• Customisation
• Modularity
• Evolution
• Assessment
• Automation
ENGINEERING OF GRAMMARWARE — A LIFE CYCLE

Evolution

Customisation

Base-line grammar

Evolution

Implementation

Parser specification

Frontend

XML serialiser

Visitor framework

Customised class hierarchy

Rendered manual

Semi-structured document
A CASE STUDY IN GRAMMAR RECOVERY AND DEPLOYMENT — VS COBOL II

- Input: IBM’s industrial standard for VS COBOL II
- Result: public, normative COBOL grammar

Time for a demo.
Grammar Transformations as a Paradigm

- Grammar re-/reverse engineering
  - Correction
  - Completion
  - Restructuring
  - Style conversion
**CORRECTION BY TRANSFORMATIONS**

Informal rule in IBM’s reference: “Level-number, data-name-1, and FILLER are not part of the REDEFINES clause itself, and are included in the format only for clarity.”

**Transformation**

```plaintext
focus on REDEFINES-clause do
delete level-number (data-name | FILLER)
```
COMPLETION BY TRANSFORMATIONS

Informal rule in IBM’s Reference: “A series of imperative statements can be specified whenever an imperative statement is allowed.”

Transformation

\texttt{generalise imperative-statement to \{imperative-statement\}+}
GRAMMAR TRANSFORMATIONS AS A PARADIGM

- Grammar implementation
  - Style conversion
  - Disambiguation
  - Conflict resolution
  - Preprocessing
  - Grammar minimalisation
GRAMMAR TRANSFORMATIONS AS A PARADIGM

- Grammar maintenance
  - Evolution
  - Optimisation
  - Migration
A FRAMEWORK FOR GRAMMAR TRANSFORMATIONS

- Suitable notion of grammars
- Primitive operators
- Combinators
- Conditions
- A suite of defined operators
Some Operators for Grammar Construction

\[
\begin{align*}
\text{generalise } P \text{ in } F & \text{ to } P' \equiv P' \text{ covers } P \land \text{ replace } P \text{ by } P' \backslash F \land P' \text{ covers } P \\
\text{include } P \text{ for } N & \equiv \text{ defined } N \land \text{ add } N \rightarrow P \\
\text{resolve } N \text{ as } P & \equiv \text{ bottom } N \land \text{ add } N \rightarrow P \\
\text{unify } N \text{ to } N' & \equiv \text{ bottom } N \land \neg \text{ fresh } N' \land \text{ replace } N \text{ by } N'
\end{align*}
\]
## Transformation Operators; Their Properties

<table>
<thead>
<tr>
<th>Operator</th>
<th>Preservation</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refactoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>preserve</td>
<td>strict</td>
<td>preserve</td>
</tr>
<tr>
<td>fold</td>
<td>introducing</td>
<td>unfold</td>
</tr>
<tr>
<td>unfold</td>
<td>strict</td>
<td>fold</td>
</tr>
<tr>
<td>introduce</td>
<td>introducing</td>
<td>eliminate</td>
</tr>
<tr>
<td>eliminate</td>
<td>eliminating</td>
<td>introduce</td>
</tr>
<tr>
<td>rename</td>
<td>modulo renaming</td>
<td>rename</td>
</tr>
<tr>
<td>Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>generalise</td>
<td>increasing</td>
<td>restrict</td>
</tr>
<tr>
<td>include</td>
<td>increasing</td>
<td>exclude</td>
</tr>
<tr>
<td>resolve</td>
<td>resolving</td>
<td>reject</td>
</tr>
<tr>
<td>unify</td>
<td>essentially resolving</td>
<td>separate</td>
</tr>
<tr>
<td>Destruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restrict</td>
<td>decreasing</td>
<td>generalise</td>
</tr>
<tr>
<td>exclude</td>
<td>decreasing</td>
<td>include</td>
</tr>
<tr>
<td>reject</td>
<td>rejecting</td>
<td>resolve</td>
</tr>
<tr>
<td>separate</td>
<td>essentially rejecting</td>
<td>unify</td>
</tr>
<tr>
<td>delete</td>
<td>“zig-zag”</td>
<td></td>
</tr>
</tbody>
</table>
A PROCESS FOR GRAMMAR RECOVERY

- Lack of a reference grammar for some intended language $L$.
- Availability of an approximative grammar $\gamma_0$ for $L$.
- Availability of representative portfolio $C$ for $L$.
- Derive the ultimate grammar $\gamma_n$ for $L$ from $\gamma_0$.
- $\gamma_n$ is (absolutely) complete if $L \subseteq \mathcal{L}(\gamma_n)$.
- $\gamma_n$ is (absolutely) correct if $\mathcal{L}(\gamma_n) \subseteq L$.
- $\gamma_i$ is derived from $\gamma_{i-1}$ by transformations $t_i$ for correction and completion of $\gamma_0$. 
ENGINEERED RECOVERY

Role of portfolio $C$
The $\gamma_i$ accept more and more of $C$.

Completeness
$\gamma_n$ parses all of $C$.

Correctness
$\gamma_n$ is covered by $C$.

Preservation
$\gamma_n$ preserves $\gamma_0$ as much as possible.

Operators
The $t_i$ are weakly semantics-preserving.

Completion
Missing branches are added.

Correction
Invalid branches are removed.
ENGINEERING OF GRAMMARWARE — SHOWCASES OTHER THAN GRAMMAR RECOVERY AND PARSER DEVELOPMENT

- Component-specific, tolerant parsers
- Portfolio-based grammar specialisation
- Transposition to XML: format evolution
- API-fication
- ...


COMPONENT-SPECIFIC, TOLERANT PARSERS

For all $x t y$ in $L(G)$
PORTFOLIO-BASED GRAMMAR SPECIALISATION

Full Grammar

Parser Generator

Parser

ASTs

Coverage Analyser

Metrics

Grammar Minimaliser

Portfolio

Minimalised Grammar
XML ENGINEERING

- XML increasingly used
  - Storage formats
  - Exchange formats
  - in component-based, distributed applications.

- DTD/XML-Schemas are grammars.
- XML schemas do evolve.
- XML challenges grammar engineering.
- Format evolution $\approx$ DB scheme evolution.
TWO LEVELS OF XML TRANSFORMATION

Input

doc. processor

Transformation

XML stream

Transformation

XML schema

Transformation

Output

doc. processor

XML schema

XML stream
XML DEMO:
STEPWISE TRANSFORMATION OF A FORM
Step 1

fold \((A_1 A_2)\) to \(A_{12}\)
Step 2

fold \((A_3 \ A_4)\) to \(A_{34}\)
Step 3

Unfold $A$
THINGS TO REMEMBER

- Recover base-line grammars
- Derive grammar uses
- Assess quality of grammars
- Keep track of grammar adaptation
- Co-evolve grammar-based software
- Treat grammars as contracts
LAST SLIDE: CORNERSTONES OF THE SOFTWARE EVOLUTION AGE

- Component-based
- Executable specifications
- Alg. for program analysis
- Generative programming
- Generic programming
- Grammar-based
- Pragmatics
- Methodology
End of slide show.