From Compilers
to Grammarware

Dr. Vadim Zaytsev
Introduction

From Compilers to Grammarware
Dr. Vadim Zaytsev

Compilers

Grammarware

Transformation

Grammar Zoo
- 974 fetched grammars
- 588 extracted
- 79 connected
- 9 adapted
+ metadata

Zaytsev, Grammar Zoo: A corpus of Experimental Grammars, SCP
Zaytsev, Grammar Maturity Model: A Survey of Grammar Zoo and Zoo-Driven Research, SCP

Maturity

Case study: JLS

Consistency

Understanding

Parsing in a broad sense
- abstract model
- concrete model
- parse graph
- parse forest
- roller picture

Reality vs. specification
- Obtain a grammar
- Construct as an oracle
- Extract from the tool
- Infer from the codebase
- Converge/diff.test

Stevenson, Candy: A Survey of Grammatical Inference in Software Engineering, SCP
Roos, Zaytsev: Repeatability of Methods of Grammatical Inference, ANER'16

Testing

Conclusion

Grammarware is more than just compilers
- Borrow methods from other domains
- Automate whenever possible
- Compare & combine
- Advance taxonomies & formalisms
- Bet on robust/tolerant methods
Introduction

- Vadim Zaytsev
- PhD in softw.lang.eng. (2010)
- Postdoc at CWI (2010–2013)
- Lecturer at UvA (2013–...
What is a compiler?
Nothing to be done for `all`.

Compiling LLParser.cpp for Debug+Asserts build
Compiling LLLexer.cpp for Debug+Asserts build
Compiling Parser.cpp for Debug+Asserts build
Building Debug+Asserts Archive Library libLLVMSwiftParser.a
Compiling BitReader.cpp for Debug+Asserts build
Compiling BitcodeReader.cpp for Debug+Asserts build
Compiling BitstreamReader.cpp for Debug+Asserts build
Building Debug+Asserts Archive Library libLLVMBitReader.a
Compiling BitWriter.cpp for Debug+Asserts build
Compiling BitcodeWriter.cpp for Debug+Asserts build
Compiling BitWriterPass.cpp for Debug+Asserts build
Compiling ValueEnumerator.cpp for Debug+Asserts build
Building Debug+Asserts Archive Library libLLVMBitWriter.a
Compiling AliasAnalysis.cpp for Debug+Asserts build
Compiling AliasAnalysisCounter.cpp for Debug+Asserts build
Compiling AliasAnalysisEvaluator.cpp for Debug+Asserts build
Compiling AliasDebugger.cpp for Debug+Asserts build
Compiling AliasSetTracker.cpp for Debug+Asserts build
Compiling Analysis.cpp for Debug+Asserts build
Compiling AssumptionTracker.cpp for Debug+Asserts build
Compiling BasicAliasAnalysis.cpp for Debug+Asserts build
Compiling BlockFrequency.cpp for Debug+Asserts build
Compiling BlockFrequencyInfo.cpp for Debug+Asserts build
Compiling BranchProbabilityInfo.cpp for Debug+Asserts build
Compiling CFG.cpp for Debug+Asserts build
Compiling CFGPrinter.cpp for Debug+Asserts build
Compiling CFAliasAnalysis.cpp for Debug+Asserts build
Compiling CGSCPPassManager.cpp for Debug+Asserts build
Compiling CaptureTracking.cpp for Debug+Asserts build
Compiling CodeMetrics.cpp for Debug+Asserts build
Compiling ConstantFolding.cpp for Debug+Asserts build
Compiling CostModel.cpp for Debug+Asserts build
Compiling Delinearization.cpp for Debug+Asserts build
Compiling DependenceAnalysis.cpp for Debug+Asserts build
Compiling DomPrinter.cpp for Debug+Asserts build
Compiling DominanceFrontier.cpp for Debug+Asserts build
Compiling FunctionTargetTransformInfo.cpp for Debug+Asserts build
Compiling IVUsers.cpp for Debug+Asserts build
Compiling InstCount.cpp for Debug+Asserts build
Compiling InstructionSimplify.cpp for Debug+Asserts build
Compiling Interval.cpp for Debug+Asserts build
Compiling IntervalPartition.cpp for Debug+Asserts build
Compiling JumpInstrTableInfo.cpp for Debug+Asserts build
Compiling LazyCallGraph.cpp for Debug+Asserts build
Compiling LazyValueInfo.cpp for Debug+Asserts build
Compiling LibCAllAliasAnalysis.cpp for Debug+Asserts build
Compiling LibCAllSemantics.cpp for Debug+Asserts build
Compiling Lint.cpp for Debug+Asserts build
Compiling Load.cpp for Debug+Asserts build
Compiling LoopInfo.cpp for Debug+Asserts build
Compiling LoopPass.cpp for Debug+Asserts build
Compiling MemDepPrinter.cpp for Debug+Asserts build
@license

@contributor{Jurgen J. Vinju - Jurgen.Vinju@cwi.nl - CWI}
@contributor{Tijs van der Storm - Tijs.van.der.Storm@cwi.nl}
@contributor{Paul Klint - Paul.Klint@cwi.nl - CWI}
@contributor{Vadim Zaytsev - vadim@grammarware.net - UvA}

//module list
module List
import Exception;
import Map;

public list[T] concat(list[list[T]] xs) =
   [] | it + xs | xs <= xs;

public map[T element, int occurs] distribution(list[T] lst) {
   res = while(isEmpty(lst)) {
      e = head(lst);
      occurs = size([el | el <= lst, el == e]);
      lst = rel[lst | el <= lst, el != e];
      yield e, occurs;
   }
}

public java list[T] delete(list[T] lst, int n);

public java list[T] repeat(int n, list[T] lst);
Rust and Go

I've been spending a bit of my time playing around with new languages—in particular, Rust has captured my imagination. The bulk of the code we write at Chef is in Ruby, Erlang, and Javascript (lately Angular.) There are things I like about all those languages:

- Ruby feels like it always hits the “whipuptitude” part of my brain. It’s easy to simply sit down and start typing, with very little in the way. It also has the expressiveness that I always loved in Perl. The more you understand the language, the more it feels like I can express myself in the same way I do with English.

- Erlang and OTP are glorious to operate. Things like pattern matching, actor concurrency, single assignment, and a lovely runtime make it a joy to run, manage, and debug production services. I think the syntax is awkward, but it too has a terse kind of beauty when you soak in it.

- Modern Javascript is becoming delightful in its own way. The ease with which you can grab community packages and frameworks, the sheer expressiveness of things like Angular, and the progressive slimming down of the often used parts of the language make the experience delightful again. It used to feel awful to me.

So—I decided to write a little Rust and, because everyone in my world seems swoony over it, Go.
Week 5 feedback (close when satisfied) #4

**grammarware** opened this issue on Oct 20 · 0 comments

**grammarware** commented on Oct 20

Ex.1: we expected the answer "quickchecking it is impossible since generating solvable sudokus is too hard", but you went ahead and implemented generation of 100-step-solvable sudoku ;) Other possible acceptable solutions include using Test.QuickCheck.Monadic library or code-cloning Week5.hs and hacking it to work with generators.

Ex.2: why would you write a function that runs forever? How is such a testing function useful?

Ex.3: you confuse minimality and (unambiguous) solveability. If we have a sudoku with four or more empty blocks, it would just be ambiguous, even if it is still valid (consistent) one.

Ex.4: seems legit. The code is okay, modulo some minor things like using `!! @` instead of `head`.

I have also noticed you have become a fan of dollars: even for two arguments you write `f $ g x` instead of `f (g x)`...

Ex.5: slow as hell, but seems to work.

Ex.6: simply because there was already a bonus question (an X-shaped megasudoku, mentioned during the lecture — which I missed, but there have been some submissions with it). I agree they could be swapped, but some people who do not consider reading papers as a special kind of torture ;) would argue that doing a surgery on a code clone to incorporate all the extra constraints of the megasudoku is more complex than setting on some strategy to either calculate complexity or generate puzzles of predefined complexity (or both).

The quality of the code for all exercises besides the last one easily makes it up to +.

**ckonig** closed this on Oct 20
Editing Compiler

Content that violates any copyrights will be deleted. Encyclopedic content must be verifiable. Work submitted to Wikipedia can be edited, used, and redistributed—by anyone—subject to certain terms and conditions.

A "compiler" is a [computer program] (or set of programs) that transforms [source code] written in [programming language] (the source language) into another computer language (the target language, often having a binary form known as [object code]).

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language) or [machine code]. If the compiled program can run on a computer whose [CPU] or [operating system] is different from the one on which the compiler runs, the compiler is known as a [cross-compiler]. More generally, compilers are a specific type of [translator] (computing) that statically generates code.

A program that translates from a low level language to a higher level one is a [decompiler]. A program that translates between high-level languages is usually called a [source-to-source compiler] or transliterator. A language [rewriting] or [rewriter] is usually a program that translates the form of expressions without a change of language. The term [compiler-compiler] is sometimes used to refer to a [parser generator], a tool often used to help create the [lexical analysis/parser] and [parser].

A compiler is likely to perform many or all of the following operations: [lexical analysis], [preprocessing], [parsing], semantic analysis ([Syntax-directed translation]), [code generation] (compiler|code generation), and [code optimization]. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest significant effort to ensure [compiler correctness].

Insert  Cite your sources: <ref></ref>

Edit summary (Briefly describe the changes you have made)

This is a minor edit

By clicking the "Save page" button, you agree to the Terms of Use and you irrevocably agree to release your contribution under the CC BY-SA 3.0 License and the GFDL with the understanding that a hyperlink or URL is sufficient for CC BY-SA 3.0 attribution.
Language processing

- Internal structures
  - databases, configurations, tables, ...
- External structures
  - protocols, interfaces, bytecode, ...
- Software language
  - programming, modelling, markup, ...
Compiler

FRONT END

MIDDLE END

BACK END
Multi-language compiler

- Front End
- Front End
- Front End
- Middle End
- Back End
Multi-target compiler

Front End

Front End

Front End

Back End

Back End

Back End

Middle End
Grammarware
Compilers transform between languages

Grammarware commits to grammatical structure
Kinds of grammarware

- Parser
- Compiler
- Interpreter
- Prettyprinter
- Scanner
- Browser
- Static checker
- Struct.editor

- IDE
- DSL
- Preprocessor
- Postprocessor
- Validator
- Model checker
- Refactorer
- Code slicer

- API
- XMLware
- Modelware
- Lang.
- RE
- Benchmark
- Recommender
- Renovation tool

Klint, Lämmel, Verhoef, Toward an Engineering Discipline for Grammarware
Languages vs. grammars

Declerative Multi-Purpose Language Definition

- Syntax Definition
- Name Binding
- Type Constraints
- Dynamic Semantics
- Transform

Languages vs. grammars

The Java™ Language Specification
Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Glad Bracha
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness
Sophia desserton and Susan Eisenbach
Department of Computer Science, Engineering and Development

1 Introduction

Java combines the experience from the development of several object-oriented languages, such as C++, C++, and Smalltalk. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justified a study of the Java formal semantics. The use of interfaces, record types as shown in Smalltalk is a simplification of the signatures extension for C++ and is - to the best of our knowledge - novel. The mechanisms for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk approach whereby all object variables are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of object-oriented programming language features or of object-oriented programming languages, there have been not many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.
What is good grammarware?
Case study: JLS

Lämmel, Zaytsev, Recovering Grammar Relationships for the
What is good grammarware?

What is good software?
What is good software?

- functional
- reliable
- usable
- efficient
- maintainable
- portable

ISO/IEC 9126.
What is good grammarware?

- functional: commits to the language
- reliable: tolerant to errors
- usable: the language is learnable
- efficient: fast (live?) and responsive
- maintainable: can be tested and evolved
- portable
Certified Language Engineer
Capability Maturity Model

- Level 1 — Chaotic
- Level 2 — Repeatable
- Level 3 — Defined
- Level 4
- Level 5 — Optimising
Grammar Zoo

- 974 fetched grammars
- 588 extracted
- 79 connected
- 9 adapted

+metadata

http://slebok.github.io/zoo

Zaytsev, Grammar Maturity Model
Zaytsev, Grammar Zoo: A Corpus of Experimental Grammarware
Improving quality

- Manual inline editing
- Refactorings
- Programmed transformations
  - +Differs
- Grammar mutations
- Inference of transformation/mutation steps
How to transform

expr : ...;
atom : ID | INT | (' expr ');

abstractize

expr : ...;
atom : ID | INT | expr;

vertical

expr : ...;
atom : ID;
atom : INT;
atom : expr;

unite

expr : ...;
expr : ID;
expr : INT;
expr : expr;

abridge

expr : ...;
expr : ID;
expr : INT;
expr : expr;

Lämmel, Zaytsev, An Introduction to Grammar Convergence, IFM’
How to mutate

- Grammar has no starting symbol?
  - Reroot2top
- Need abstract syntax from concrete syntax?
  - RetireTs
- Grammar productions written in an
  - DeyaccifyAll
- Change naming convention?
  - RenameAllNLower2Camel
How to be guided

• Equality & algebraic equivalence
• Prodsig-equivalence
  • signatures based on nonterminal patterns
  • tolerant to permutations
  • weak equivalence tolerant to iteration kinds
• Abstract Normal Form
  • no terminals, labels, markers
  • consistent disjunctive style

Zaytsev, Guided Grammar Convergence,
How to be guided

\[
\begin{align*}
\textit{pmaster} &= p(\varepsilon, \text{expr}, \text{expr} \cdot \text{operator} \cdot \text{expr}) \\
\textit{pantlr} &= p(\varepsilon, \text{binary}, \text{s}(l, \text{atom}) \cdot *(\text{s}(o, \text{ops}) \cdot \text{s}(r, \text{atom}))) \\
\textit{pdcg} &= p(\text{binary}, \text{expr}, \text{atom} \cdot *(\text{ops} \cdot \text{atom})) \\
\textit{pemf} &= p(\varepsilon, \text{Binary}, \text{s}(\text{ops}, \text{Ops}) \cdot \text{s}(\text{left}, \text{Expr}) \cdot \text{s}(\text{right}, \text{Expr})) \\
\textit{pjaxb} &= p(\varepsilon, \text{Binary}, \text{s}(\text{Ops}, \text{Ops}) \cdot \text{s}(\text{Left}, \text{Expr}) \cdot \text{s}(\text{Right}, \text{Expr})) \\
\textit{pom} &= p(\varepsilon, \text{Binary}, \text{s}(\text{ops}, \text{Ops}) \cdot \text{s}(\text{left}, \text{Expr}) \cdot \text{s}(\text{right}, \text{Expr})) \\
\textit{ppython} &= p(\varepsilon, \text{binary}, \text{atom} \cdot *(\text{operators} \cdot \text{atom})) \\
\textit{padt} &= p(\varepsilon, \text{FLEexpr}, \text{s}(\text{binary}, \text{s}(\text{e1}, \text{FLEexpr}) \cdot \text{s}(\text{op}, \text{FLOp}) \cdot \text{s}(\text{e2}, \text{FLEexpr}))) \\
\textit{prascal} &= p(\text{binary}, \text{Expr}, \text{s}(\text{leexpr}, \text{Expr}) \cdot \text{s}(\text{op}, \text{Ops}) \cdot \text{s}(\text{reexpr}, \text{Expr})) \\
\textit{psdf} &= p(\text{binary}, \text{Expr}, \text{Expr} \cdot \text{Ops} \cdot \text{Expr}) \\
\textit{ptxl} &= p(\varepsilon, \text{expression}, \text{expression} \cdot \text{op} \cdot \text{expression}) \\
\textit{pxsd} &= p(\varepsilon, \text{Binary}, \text{s}(\text{ops}, \text{Ops}) \cdot \text{s}(\text{left}, \text{Expr}) \cdot \text{s}(\text{right}, \text{Expr}))
\end{align*}
\]
What we want in general

- Maintenance assistants
- infer whatever possible
- provide advice on the rest
- Not necessarily “request => result or fail”
- pending
- negotiated
Negotiating the result

rename(expr,Expr)
no expr!
rename(exp,Exp)
ok

Zaytsev, Negotiated Grammar Evolution
Key points

• For grammarware, we need
  • consistency
  • a clear quality model
  • improvement processes
  • automation

• Also,
  • understanding user scenarios
Parsing in a broad sense

- Grouped tokens
- Abstract model
- Visual diagram
- Typed tokens
- Concrete model
- Graph model
- Slices/tokens
- Parse graph
- Vector drawing
- Raw string
- Parse forest
- Raster picture

Zaytsev, Bagge, Parsing in a Broad Sense, MoDELS'
So, grammarware is based on grammars...

...can we test/validate it based on grammars?
Grammar-based testing

- Purdom’s generator
  - builds the shortest conforming term
- Maurer’s generator
  - randomly selects alternatives
- Coverage criteria
  - TC, NC, PC, BC, UC, CDBC
- Negative cases?

Fischer, Lämmel, Zaytsev, Comparison of CFGs Based on ... Test Data
Combinatorial explosion

- Java (Habelitz)
- Java (Parr)
- Java (Stahl)
- Java (Studman)
- TESCOL (00001)
Combinatorial explosion

Figure 5.7: Number of generated sentences for UC

The number of generated sentences is mostly higher than unfolding criteria with few equalities as shown in figure 5.9.

Figure 5.8: PC ratio for CDBC

Figure 5.9: Number of sentences for CDBC

Figure 5.10 summarizes the number of sentences generated per coverage criteria for each grammar sample.
Nonterminal matching
Badly matched:

Fischer, Lämmel, Zaytsev, Comparison of CFGs Based on ... Test Data
Differential methods

- Oracles are unnecessary
- Comparing grammars
  - of varying structure, style, etc
- across TSs
- Investigate disagreements

McKeeman, Differential Testing for Software
What is a bug?

- Grammarware processes languages
- A bug is a program
- Inspect programs to deal with bugs

The First Computer Bug
Photo #_________
Reality vs. specification

- Obtain a grammar
- Construct as an oracle
- Extract from the tool
- Infer from the codebase
- Converge/diff.test

Stevenson, Cordy, A Survey of Grammatical Inference in Software Engineering
Roș
Antipatterns

• Some ways lead to bugs faster
• Detect them => predict defects
• Smells
  • left/right recursion
  • ambiguous x*?

Taba, Khomh, Zou, Hassan, Nagappan, Predicting Bugs Using Antipatterns, ICSM 2013
Sajnani, Saini, Lopes, A Comparative Study of Bug Patterns in Java, SCAM 2014
Trubiani, Di Marco, Cortellessa, Mani, Petriu, Exploring Synergies...
Process improvement

- find defects
- fix defects
- learn how to fix defects
- learn to tolerate defects
- learn to avoid
# Semiparsing

- ad hoc lexical analysis
- hierarchical lexical analysis
- lexical conceptual structure
- iterative lexical analysis
- fuzzy parsing
- parsing incomplete sentences
- island grammars
- lake grammars
- robust multilingual parsing
- gap parsing
- noise skipping
- bridge grammars
- skeleton grammars
- breadth-first parsing
- iterative syntactic analysis
- grammar relaxation
- agile parsing
- permissive grammars
- hierarchical error repair
- panic mode
- noncorrecting error recovery
- practical precise parsing

Zaytsev, Formal Foundations for Semi-parsing, CSMR-WCRE’
Grammarware is more than just compilers
Borrow methods from other domains
Automate whenever possible
Compare & combine
Advance taxonomies & formalisms
Bet on robust/tolerant methods
Thank you!

Questions?

- Sources:
  - Figures used from own papers & talks
    - + Eelco Visser’s keynote @ MODULARITY
    - + Tobias Baanders
    - + JLS book covers (Fair Use)
  - Self-made screenshots
  - All photos from public domain
  - Comfortaa: font
Introduction

Compilers

Grammarware

Transformation

Maturity

Consistency

Understanding

Testing

Conclusion