Notation-Parametric Grammar Recovery

Vadim Zaytsev

Software Analysis and Transformation Team,
Centrum Wiskunde & Informatica, The Netherlands

February 3, 2012

Abstract

Automation of grammar recovery is an important research area that received attention over the last decade and a half. Given the abundance of available documentation for software languages that is only going to keep increasing in the future, there is need for reliable extraction techniques that allow grammar engineers to derive useful information from it. This information can be further used to build grammarware, like parsers or test generators, or to perform grammar investigation. Grammars obtained systematically from existing sources always have preference over manually constructed ones due to traceability of their issues, including errors and design weaknesses. This paper focuses on automated grammar recovery from sources that utilise a family of metasyntaxes known as EBNF: many language specifications extend the well-studied Backus Naur Form in different directions, resulting in unnecessary diversity of syntactic notations. To enable manipulation of EBNF families, we use EDD, the EBNF Dialect Definition, a recently published DSL for notation specification, and base our approach on human-specified indications that guide the subsequent automated heuristic-based recovery process. Two separate scenarios are considered in the paper: a reliable syntactic notation and an unreliable one, with the latter being remarkably more difficult to handle, but also substantially more useful since it is so often encountered in practice. The proposed approach has been verified by two prototypes that were capable of extracting dozens of grammars written in 42 different syntactic notations.

1 Introduction

Software engineering, and grammarware engineering in particular, has been facing the problem of abundance of notation for syntactic definitions for quite a long time [18]. With many grammars in existence, it is desirable for a language engineer to reuse them instead of developing new ones by hand from scratch. Quite a number of grammar recovery projects were attempted and successfully performed during the last decade and a half [11, 12, 14, 16, 17, 19, 20, 21, 24, 22, 23].

When recovering a language grammar from an existing source, one needs to face various challenges ranging from character level issues (e.g., layout inconsistencies) to language level issues (e.g., grammar connectedness).

In the current paper, the main problem that we are solving is **how to reuse grammar artefacts that are written in different notations**. We wanted a method that reliably works on a big number of grammars of industrial size, obtained from unreliable sources, and that is easily reproducible for future derivatives. We have addressed these challenges of dealing with different notations for syntactic definitions ("metasyntaxes" from now on, dialects of (E)BNF) parametrically. We use the metametasyntax proposed in [24] as a stepping stone to enable manipulation of families of metasyntaxes. The approach is not specific to any metaprogramming (or rather, metametaprogramming in our context) language. The semi-automated notation-parametric grammar recovery tool is written in Rascal [8] and is publicly available through Sourceforge as `edd2rsc`[1]. The fully automated notation-parametric grammar recovery tool is written in Python and is publicly available through Sourceforge as well as the **Grammar Hunter**[2], with a Rascal version in development (will be released as a standard library after sufficient polishing and documenting).

The rest of the paper is structured as follows. §2 provides an extensive overview of previously existing work directly related to grammar recovery from manually constructed sources. In §3 we consider a semi-automated scenario which works smoothly only for reliable notations and requires interaction with a human grammar engineer otherwise. We have validated this approach on all notations that were specified during recovery of grammars for the Grammar Zoo [20], with one example of them provided in the same section (an interested reader can get the rest freely from the web). In §4 a substantially more advanced algorithm is demonstrated to enable full automation. The internal details of the prototype tool, **Grammar Hunter**, are presented in several blocks for the sake of reproducibility and reporting design details. **Grammar Hunter** was validated by successfully feeding it diverse language manuals and standards (also available at the Grammar Zoo). Conclusions are drawn in §5.

2 Grammar recovery progress and timeline

One of the first studies in grammar recovery dates back to 1996 and is reported by Sellink and Verhoef [16]: it concerns Message Sequence Charts, a DSL described in a Word document, which was converted to Postscript for the lack of API at that time. The Postscript document was converted to an ASCII file which was processed by a Perl script and produced BNF rules, which were in turn manually edited with all 14 changes claimed to be documented. Another script was used to generate a hypertext form of a grammar suitable for browsing.

[1] Software Language Processing Suite (SLPS), `http://slps.sf.net`; see `topics/recovery/edd2rsc` in particular.

[2] Ibid., see `topics/recovery/hunter`.

Van den Brand, Sellink and Verhoef reported in 1997 on successfully obtaining a COBOL grammar capable of handling a range of language dialects [17]. The help of a Master student was used to convert 1100 production rules of the ANSI COBOL 85 standard to SDF. A long and sophisticated process of forced coupling followed, leading to (disciplined) changes brought both to the codebase and to the grammar, and resulting in capability of the adjusted grammar to parse the adjusted source code.

Switching System Language (SSL), also reported by Sellink and Verhoef in 2000 [16], was a proprietary DSL documented in a set of HTML files containing its grammar in an BNF dialect they called SBNF. The endeavour is remarkable for our current work in a way that it was an attempt to use precise parsing on an unreliable source. A range of (as we now know) typical issues arose such as naming convention violations and non-matching brackets, and significant amount of interactive grammar adjustments was needed. The project succeeded also due to development support of the ASF+SDF Meta-Environment, resulting again in the situation where an adjusted SBNF grammar was used to parse adjusted syntax rules.

Programming Language for EXchanges (PLEX) is another notable example, reported by Lämmel and Verhoef in 2001 [11]. It was a complex DSL consisting of 20 sublanguages (sectors) and having over 60 Mb of grammarware source code. The mining process delivered fragments of BNF found in the comments, which with the help of six parsers were transformed to pure aggregated BNF and subsequently to SDF, which was combined with a lexer. The project took only two weeks and resulted in parsing 8 MLOC of unmodified PLEX.

The case of IBM VS COBOL II is one of the most complicated among those reported in academic sources: it was described in a different paper by Lämmel and Verhoef in 2001 [12]. A raw grammar was extracted from the language documentation, which was already a bit tricky since it used "railroad track" kind of syntax diagrams instead of purely textual BNF. After static errors were taken into account and the lexical syntax was added, the project entered the phase of test-driven correction and completion. Several phases of grammar recovery followed, including beautification, modularisation, disambiguation and adaptation. The IBM VS COBOL II grammar is still freely available for reuse from the authors' website [10].

Until 2005 it could have been assumed that grammar recovery is only needed for legacy languages like COBOL and for badly documented DSLs developed in-house. However, a set of very similar problems arose with C#, the most modern language of the time, as reported in [19]. In order to parse C# code, the project involved manual transition from the ECMA-produced PDF to LLL and intensive grammar transformation with FST and GDK.

In 2009, Lämmel and Zaytsev demonstrated a different approach to grammar recovery [13]. They opted for the lightweight extraction with choosing only reliable sources as starting points (SDF, ANTLR, DCG, TXL, LLL), mapping the basic features to the target notation and abstracting out the rest. Grammars of FL, Fortran, Modula-3, BNF, EBNF, YACC were extracted using this approach: they have proven to be quite useful for grammar analysis, but they

are unsuitable for parsing as such. The main reasons are lacking lexical sections and the simple straightforward nature of the process that lets the extracted grammar stay true to the source while still containing its specific bugs.

In the next years the same authors extended their approach and added tolerance to layout inconsistencies and other lexical deviations of the source grammars. This shift back from extraction to recovery resulted in a successful grammar recovery and a detailed grammar analysis of Java 1.0, 1.2 and 5.0 [14], extraction of ISO-published grammars of C, C++ and C# [21], turning scattered fragments of the MediaWiki grammar into an operational artefact [22], as well as to similar results with languages like Ada, C++, Dart, Eiffel, Modula-3 [20]. The current paper presents experience collected during those various projects and reports on two tools (`edd2rsc` and **Grammar Hunter**) that perform interactive and robust grammar recovery correspondingly, while both relying on a set of indications provided by a grammar engineer as a specification for intended syntactic notation [24].

3 Semi-automatic notation-parametric recovery

Let us make an assumption that the notation of the source for grammar extraction to be **reliable**. Given a notation specification in EDD, our tool called `edd2rsc` automatically produces a Rascal grammar. This grammar can be used in many ways, in particular for parsing reliable sources from other metaprograms and for opening any well-formed grammar in this notation in Eclipse IDE with coloured syntax highlighting, as seen on Figure 1. Any notation violations are visible to the grammar engineer as parsing errors and can be treated one by one (hence "*semi*-automatic"). This method is particularly useful for bulk imports of grammars extracted from parser specifications: since those are executable artefacts of a different grammarware framework (i.e., parser generator), their syntax is fairly stable and can be directly mapped to the target notation. Many grammarware frameworks use notations that can be regarded as dialects of (E)BNF, among notable examples we can name ANTLR [15], GDK [9], JavaCC [3] and YACC [7].

Any Backus-Naur Form, notwithstanding its extendedness, works as follows: a defining nonterminal is written on the left hand side, followed by a so called defining metasymbol (such as ":"), followed by the right hand side, followed by a terminator metasymbol (such as ";") that signals the end of a production rule. A grammar may consist of several subsequent production rules. The right hand side of any production is a list of alternatives separated by a definition separator metasymbol (such as "|"), and every alternative is a list of symbols separated by a concatenation metasymbol. If the expected values of all metasymbols are known from the notation specification, it is easy to capture the contents of this paragraph in a grammar.

Any notation specification already available at the Grammar Zoo, any notation derived by notation transformation (our future research focus) or any other notation specification can be provided as an input for `edd2rsc`. To demonstrate

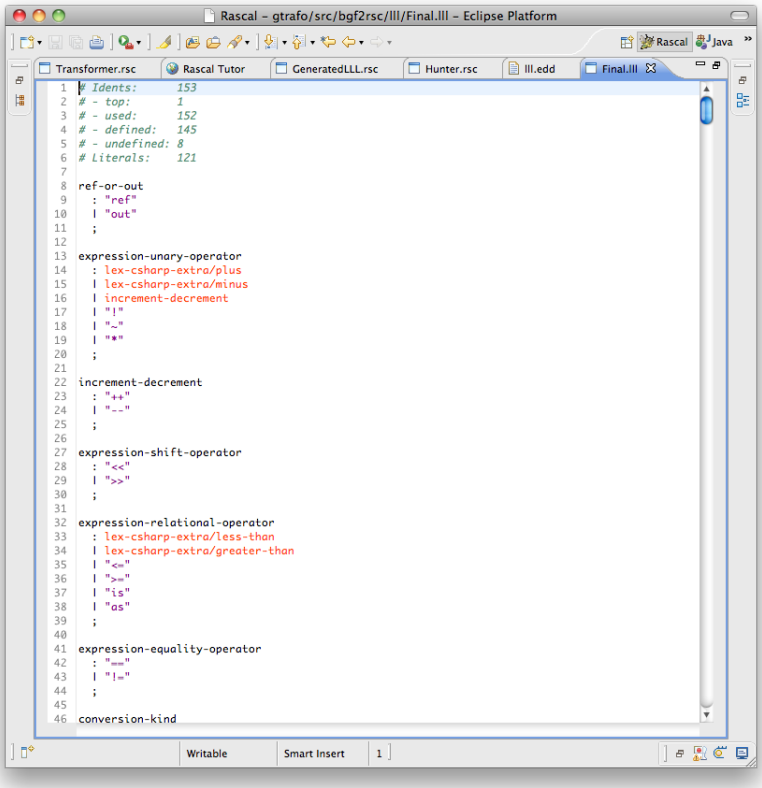

Figure 1: A C# grammar [19] in LLL opened in Eclipse.

how it works, let us consider one of them, namely LLL, a notation of the Grammar Deployment Kit [9], which was chosen purely for its simplicity and for being LDTA material. Interested readers are once again recommended to try the tool themselves.

Figure 2 shows LLL expressed in LLL: the definition is taken from the online manual since it reflects a newer version compared to the original paper; with all editions freely available for viewing and downloading at the Grammar Tank project page [23]. Figure 3 expresses the same notation in terms of EBNF Dialect Definition (EDD), a DSL introduced in [24], with metasymbol names heavily influenced by ISO EBNF [5]. From that specification, a Rascal grammar can be generated with `edd2rsc`, a tool that covers much more functionality than needed for this simple example, and still is hardly longer than 200 lines of Rascal code. The resulting grammar can be seen on Figure 4: being small and straightforward, it still provides nice IDE support, as we have seen on Figure 1, and a good start for metaprograms that need to manipulate LLL grammars.

```
specification : rule+;
rule          : ident ":" disjunction ";";
disjunction   : {conjunction "|"} +;
conjunction   : term+;
term          : basis repetition?;
basis         : ident
              | literal
              | alternation
              | group
              ;
repetition    : "+" | "*" | "?";
alternation   : "{" basis basis "}" repetition;
group         : "(" disjunction ")" ;
```

Figure 2: LLL expressed in LLL, taken from the GDK reference manual.

defining metasymbol	:	definition separator metasymbol	|
terminator metasymbol	;	start one line comment metasymbol	#
postfix optional metasymbol	?	nonterminals may contain	- /
postfix star metasymbol	*	postfix plus metasymbol	+
start terminal metasymbol	"	end terminal metasymbol	"
start group metasymbol	(	end group metasymbol	)
start separator list star metasymbol	{	end separator list star metasymbol	} *
start separator list plus metasymbol	{	end separator list plus metasymbol	} +

Figure 3: LLL notation specification, pretty-printed from `lll1.edd` for publishing purposes.

```
module LLL
import util::IDE; // needed only for advanced IDE support (see last two lines)
start syntax LLLGrammar = LLLLayoutList LLLProduction* LLLLayoutList;
syntax LLLProduction = LLLNonterminal ":" {LLLDefinition "|"}+ ";";
syntax LLLDefinition = LLLSymbol+;
syntax LLLSymbol
 = @category="Identifier" nonterminal: LLLNonterminal
 | @category="Constant" terminal: LLLTerminal
 | group: "(" LLLDefinition ")"
 | optional: LLLSymbol "?"
 | star: LLLSymbol "*"
 | plus: LLLSymbol "+"
 | sepliststar: "{" LLLSymbol LLLSymbol "} *"
 | seplistplus: "{" LLLSymbol LLLSymbol "} +";
lexical LLLTerminal = "\"" LLLTerminalSymbol* "\"";
lexical LLLTerminalSymbol = ![\"];
lexical LLLNonterminal = [A-Za-z_01-9\-/]+ !>> [A-Za-z_01-9\-/];
layout LLLLayoutList = LLLLayout* !>> [\t-\n \r \ ] !>> "#";
lexical LLLLayout = [\t-\n \r \ ] | LLLComment ;
lexical LLLComment = @category="Comment" "#" ![\n]* [\n];
Tree getLLL(str s,loc z) = parse(#LLLGrammar,z);
public void registerLLL() = registerLanguage("LLL","lll",getLLL);
```

Figure 4: LLL expressed as a Rascal grammar, generated automatically from the notation specification.

4 Automatic notation-parametric recovery

Suppose a source in a form of an electronic language reference manual. It probably includes an explicit grammar of the language, but presents it with its own peculiar notation. The grammar text can be either copy-pasted or OCRed from it. Using both the section of the language document that describes the notation and the grammar fragments, we can reverse engineer the syntactic notation and specify it in EDD. However, an attempt to parse such a grammar file precisely with it will fail, because symbols will be misspelled, misplaced, left out, etc — unintentional mistakes and inconsistencies encountered regularly in handcrafted grammars.

An alternative approach to the one presented in the previous section, is to not always rely on the syntactic notation. Indeed, we can formulate a list of heuristics that can be used to overcome notation deviations without human expert intervention: based on context and circumstances, certain hypotheses can be formulated and verified, and correcting actions can be taken based on the outcome. For example, if a production rule lacks terminator metasymbol, it is added; if a metasymbol is misspelled and there is enough evidence to infer the correct one, it is repaired; if indentation and markup information is lost, tokens are identified based on layout-independent criteria. The experience we have collected with recovering grammars semi-automatically, was channelled into development of a tool called **Grammar Hunter**. It requires a notation specification as a parameter and delivers the recovered grammar to the best of its knowledge. Post-extraction correction is needed in cases where the necessary information is not present in the source and cannot possibly be inferred (e.g., a production omitted from a standard completely).

The solution we propose is laid out in detail in the following sections, with the whole process split into different subsequent blocks:

Block 1: Selective line reading.
Reads the file, fetches grammar fragments, applies line continuation rules to relevant lines, filters out comments, delivers the list of characters.

Block 2: Composition of tokens from characters.
Transforms the list of characters into the list of tokens, while taking quoting rules into account.

Block 3: Tokens classification.
Classifies each token as a terminal, nonterminal or a metasymbol.

Block 4: Token groups normalisation.
Converts postfix/prefix to confix, delivers the list of grammar rules.

Block 5: Context-dependent reconsideration.
Performs correction heuristics: decomposes and assembles symbols, rebalances symmetric metasymbols, ignores negligible leftovers.

Readers who prefer to have a running example and can go through extreme amounts of tiny implementation details, are redirected to a report on MediaWiki grammar recovery [22], which was also done with Grammar Hunter.

4.1 Block 1: Selective line reading

The main purpose of Block 1 is to get from a file or any other form of input stream to a list of textual lines. Since our main focus is on recovering grammars from documentation, it usually involves working with physical lines, filtering and cropping them, but one can imagine much more intricate algorithms if our approach is reapplied to mining or carving. At the end of Block 1, we would have applied all notational policies formulated based on lines, with all the remaining ones relying on characters and their specific relative positions.

Grammar Hunter starts by reading the lines of the textual part of the source. It turns out that many notation policies are based on the notion of a physical line of code, and those policies are the ones that need to be accounted for before anything else. If grammar fragment delimiters are known, then **Grammar Hunter** collects all fragments, otherwise it treats the whole input as one big fragment. Ignored lines are filtered out: one can possibly introduce an advanced mechanism for that (regular expressions, mini-grammars, etc), but for all scenarios we have encountered so far it was enough to filter undesirable lines by keywords. For example, the grammar text copied from the ISO C++ standard [4] contains a header line "ISO/IEC 14882:1998(E) © ISO/IEC" several times, and it is rather straightforward to drop them on the level of working with lines.

In the worst situation, grammar fragment delimiters are not known and not used, but the input file still contains a lot of auxiliary information. It happens with sources that were taken as flat textual files with no structural clues. In this case, we need to step down to the next best notation-driven heuristic, which is to rely on defining and terminator metasymbols. A grammar fragment is then defined as anything starting with a token that can be a nonterminal name, followed by a defining metasymbol, and ending with a terminator metasymbol. For instance, consider the following code:

```
Formal Parameters
Every function declaration includes a formal parameter list, which consists ...
The following can be simplified to:
formalParameterList
   : '(' normalFormalParameters (, optionalFormalParameters)? ')'
   ;
optionalFormalParameters
: restFormalParameter |
  namedFormalParameters
;
normalFormalParameters:
      normalFormalParameter (',' normalFormalParameter)*
   ;
Positional Formals
A positional formal parameter is a simple variable declaration.
```

(The source fragment is taken from the "download as text" result from [2] and slightly edited for the sake of simplicity). If we know that defining metasymbol is "`:`" and terminator metasymbol is "`;`", we will be able to extract production rules for nonterminals `formalParameterList`, `optionalFormalParameters` and `normalFormalParameters`.

Many language specifications, especially the ones created in the 70s and 80s, had to deal with specific format limitations of their target architectures even on the textual level. One of the popular limitations was a fixed maximum character capacity of one line: i.e., when the line needed to be longer than that maximum, it had to be explicitly split into several lines. This is done by using a line continuation policy, which we need to process in this block.

Just as with any programming language, comments can and should also be (dis)regarded as whitespace. Most of the (E)BNF dialects used for language documentation either do not have any commenting notation or have one-line comments, but it is not uncommon for executable notations used in compiler compilers to have multiline comments. Since they can have specific characters inside them, which will impede the recovery process later on, we remove all comments in this block. The remaining lines go through a trivial step of being converted to a list of single characters.

4.2 Block 2: Composition of tokens from characters

This block derives a list of tokens from the list of characters. A token is usually a nonterminal symbol, a terminal symbol or a metasymbol, but depending on the notation, a token can also be a special indentation marker or just an unknown type of symbol. The output expected from Block 2 is a list of unclassified tokens.

In [24], we speak of *whitespace reliability* as a decision point: whether to trust whitespace to separate one token from the next one, or to assign token boundaries based on other heuristics. In general, these are two fundamentally different ways to approach layout. Pretty-printed grammars like the one we have seen on Figure 1 are the most reliable with respect to layout, but in the case of LLL it is not crucial since that notation was specifically designed by professional grammar engineers. In many cases like the notation used in Java grammars which we considered in detail in [14], different alternatives on the right hand side of every production rule are separated by indentation only, so one must rely on such whitespace to identify them, but at the same time one should disregard any other whitespace because tokens often appear glued together anyway. Problems arise when one (crucial) kind of whitespace cannot be told from the other (negligible) kind.

In the original BNF [1], nonterminals were enclosed in `<` and `>`, and anything unenclosed was considered to be a terminal. In the original EBNF [18], terminals were quoted, and anything unquoted was considered to be a nonterminal. All syntactic notations ever since choose either of these ways or both. The unintentional case when neither nonterminals nor terminals are delimited must be experienced when the documentation creators decide to mark them with a specific font, but the extraction procedure cannot extract that kind of information from the document. In any case, when any of the delimiters are known, we can immediately start composing multi-character tokens, also paying attention to escape rules (e.g., for quotes between quotes) if they are present. Multi-character metasymbols are also be assembled in this block, since they rarely occur in unquoted form outside their deserved context.

4.3 Block 3: Tokens classification

This block finalises the process of classification of single tokens, all ambiguous roles get resolved here. We emphasise the focus being on single tokens, since the following two blocks concern themselves with classification of tokens or adjusting their roles based on context.

There are several commonly encountered naming conventions for nonterminals. The most important and well-known ones are `UPPERCASE`, `lowercase`, `CamelCase` and `mixedCase`. Properly cased words can be glued together or concatenated with a space, and underscore or a dash as a separator. For more details and usage statistics the interested reader is referred to [24]. If any convention is known, **Grammar Hunter** can utilise it to classify particular tokens as nonterminal symbols.

In grammar engineering practice, it is unheard of, for nonterminals to have non-alphanumeric names: hence, all non-alphanumeric tokens of unknown type can be assigned a role of a terminal (unless this particular combination of characters can also be a metasymbol). One should of course be cautious with borderline characters like "`-`", "`_`" or "`/`" that can sometimes be a *part* of a nonterminal name (i.e., "`class-name`" is most probably a valid nonterminal name, while "`--`" most probably is not). This heuristic is applied aggressively: even if something like a curly bracket is marked as a nonterminal, this is bound to be a mistake that needs to be corrected, since a curly bracket is not a valid nonterminal name unless otherwise specified. Conventions like these can be manually written or derived by application of machine learning techniques.

4.4 Block 4: Token groups normalisation

This block searches for specific patterns of occurrence of symbols and metasymbols, and performs normalisation on them. Its input is a heterogeneous list of tokens, but its output is already a structured grammar, with separate production rules, alternatives, explicit grouping of symbols and similar features that make it less of a list and more of a tree.

Conceptually, the most important normalisation is composition of grammar production rules from sublists of tokens. This is done with the help of terminator and defining metasymbols, and in total there are four scenarios possible:

Only terminator metasymbol is known.
: Since terminator metasymbols were originally meant to separate production rules, we can use them directly to slice the tokens list in pieces in order to treat each nontrivial piece as a production rule. This heuristic is very straightforward and flexible, but the less reliable the notation is, the more errors are introduced by relying only on terminator metasymbols (they can be easily forgotten or misspelt in manually created grammars).

Only defining metasymbol is known.
: Similarly, **Grammar Hunter** will rely on defining metasymbol to slice the token list into productions. Combined with checking for the alphanumeric

nature of the token directly preceding the defining metasymbol (the best candidate for the defining nonterminal), this proves to be quite a reliable heuristic. It should be noted here that knowing a location of defining metasymbol is slightly more reliable with respect to identifying the left and the right hand sides of the production, for a number of reasons: the token immediately following the terminator metasymbol, is not necessarily the defining nonterminal of the next production (it can be an optional part of the same terminator metasymbol, a production label or anything else), and the token immediately following the left hand side, is not necessarily the defining metasymbol (unreliability may cause it to spread over several tokens or be completely lacking).

Both terminator and defining metasymbols are known.
Besides using this information to make the recovery process more stable and precise, our tool will also perform double checks in cases like this one when more information is provided than usual, and report on any mismatches between the metasymbol values expected from the specification and the metasymbol values expected after analysing the source grammar text.

Neither terminator nor defining metasymbols are known.
Even when no information is provided by the notation specification, **Grammar Hunter** can still infer enough information to complete the recovery process: in particular, frequency analysis was observed to be among the most reliable techniques: for each unique token, we count how many times it occurs in the grammar being recovered. In big grammars the most commonly encountered tokens are usually either layout or terminator and defining metasymbols. **Grammar Hunter** takes the most popular ones and tries them out in various combinations. When a decision like this is taken, the certainty is reported to the end user.

Many contemporary language documents use the so called multiple defining metasymbol ("one of") which quite often remains undocumented. It is used instead of the normal defining symbol and changes the semantics of the right hand side of its production rule: the list of symbols are treated as a choice, not as a sequence.

Grouping tokens in productions is one of the two most important activities of Block 4. It is just as important to convert all postfix (and much more rare prefix) metasymbols to confix ones[3]. Normalising all metasymbols that affect the structure of the grammar, notwithstanding the arity, to the confix form, gives more power to the heuristics of the next block, as well as more structure that needs to be recovered anyway.

[3] A **postfix** metasymbol occurs immediately after a symbols it affects (e.g., "`t*`"). A **prefix** metasymbol occurs right before a symbols it affects (e.g., "`!t`"). **Confix** metasymbols form a pair that both precede and follow the affected symbols (i.e., it is a bracketing construction).

4.5 Block 5: Context-dependent reconsideration

The role of Block 5 is to reconsider particular metaroles of symbols based on the context where they occur. The heuristics exercised in this block are not necessarily involved in the grammar extraction process as such, but because we specifically address the unreliable syntactic notation scenario, it is useful to have a round of notation-driven corrections.

There are several situations when one token that has survived through all the previous blocks, needs to be decomposed into two tokens. The most common situation occurs when a postfix metasymbol is alphanumeric (i.e., "opt" instead of "?" for marking optionality) and the documentation creators were using a different font variant to explicitly mark it, but that information could not be propagated to the extractor. For example, in the Java Language Specification `ClassBodyopt` should be disassembled into a nonterminal `ClassBody` and a postfix optionality metasymbol `opt` [14, p.352].

The opposite situation occurs when the font change erroneously happens in the middle of a token, if that font change is perceived as a token boundary. This deviation is common for handcrafted documentation which creation process is prone to misclicks. For example, in a different version of the Java Language Specification we have seen "continu e" and "S witchBlockStatementGroups" — the former was turned into a terminal symbol because it matched the naming convention by being completely lowercase; the latter was turned into a nonterminal symbol because such a hypothesis was formed and verified by finding a definition of the nonterminal SwitchBlockStatementGroups [14, p.351]. It is crucial that the hypothesis needs to have a good reason to be formulated and only then verified, since it is not uncommon for grammars in language documentation to have a nonterminal symbol and a terminal symbol share a name: for example, the Ada grammar has "pragma", "range" and "body" [6].

There are more assumptions that can be formulated about the symbol roles: for example, that confix metasymbols should have some symbols between them (unless that is a special notation for ε); that infix metasymbols should not occur as the first or the last in a sequence; that postfix metasymbols should not start a sequence and prefix ones should not end it; that confix metasymbols should occur in pairs. These assumptions are verified and if not satisfied, the suspicious symbols need to change their metarole.

Symmetric (confix) metasymbols can be very efficiently balanced: once forward and once backward. Forward balancing scans tokens from the occurrence forward to the end of production in search for the matching nonterminal. If the metasymbol cannot be balanced, **Grammar Hunter** attempts to substitute it with another metasymbol with whom they share lexical representation (e.g., on Figure 3 we see a separator list star and a separator list plus are both started with "{"). Similar algorithm is applied backwards, when we scan the context from the end metasymbol occurrence toward the start. If all hypotheses fail, the role of an unbalanced metasymbol is changed to a terminal.

When a grammar is meant to be complete and fully connected (i.e., with one top nonterminal and no bottom nonterminals), we can adjust the notation

specification with a policy to treat all undefined nonterminals as terminals. Heuristics like this reside in Block 5 since in order to calculate top, bottom and defined nonterminals, one has to have confidence in the rest of the grammar.

5 Conclusion

Based on the generalised way to specify a syntactic notation as EDD [24], we enhance the technique of grammar recovery [12] by applying a set of heuristics extended compared to [14] and parametrised with the details of the assumed syntactic notation. Two prototype notation-parametric grammar recovery tools have been developed and presented: a semi-automatic `edd2rsc` that works best with families of source grammars using the same syntactic notation reliably; and a fully automatic **Grammar Hunter** that contains five blocks of heuristics and performs tolerant scanning and parsing of unreliable sources. In order to evaluate the chosen methodology, we have used it to recover dozens of grammars of Ada, Basic, C, C++, C#, Dart, EBNF, Eiffel, Fortran, Java, LLL, Modula-3, Wiki, WSN, XPath and YACC, with most being of industrial size. **Grammar Hunter** successfully works on grammars with notation deviations and successfully overcomes the majority of problems posed by unreliable syntactic notations often found in handcrafted manuscripts, as our experiments show. Both tools, as well as all recovered grammars, are released as open source and made available through Software Language Processing Suite (SLPS), a Sourceforge project that can be found on `http://slps.sf.net`. The grammars recovered with Grammar Hunter form a considerable part of the Grammar Zoo [20]. After some finishing touches, extensive testing and polishing documentation, Grammar Hunter will be officially released as a standard library of the Rascal meta-programming language [8].

Semi-automatic notation-parametric grammar recovery is most suitable for grammar engineers who prefer to edit their grammars in place: `edd2rsc` provides them with an easy way to get the IDE support for their activities. The same approach is also perfect for importing executable parser specifications in bulk. Highly reliable and error-tolerant *automatic* notation-parametric grammar recovery tool **Grammar Hunter** can be used by grammar engineers who seek balance between automation and traceability.

Just as with any scenario involving imprecise mapping, there are two fundamentally different approaches to normalisation (performed by Grammar Hunter as described in subsection 4.4). First, we could try to express all encountered syntactic constructions in terms of the target syntactic functionality. Alternatively, we could attempt to fit as much of the original constructions into the target functionality, extending it if necessary. As it turns out, this choice does not matter for our normalisations (composing production rules from a heterogeneous stream of tokens and converting metasymbols to a confix form).

The list of yet to be solved problems includes systematic evolution of syntactic notations, which should be coupled to the evolution of grammars written in that notation. We also plan to continue extending the Grammar Zoo.

References

[1] J. W. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In S. de Picciotto, editor, *Proceedings of the International Conference on Information Processing*, pages 125–131, Unesco, Paris, 1960.

[2] Gilad Bracha. *The Dart Programming Language Specification, Draft Version 0.05*. The Dart Team, November 2011. http://www.dartlang.org/docs/spec/dartLangSpec.html.

[3] Tom Copeland. *Generating Parsers with JavaCC*. Centennial Books, second edition, 2007.

[4] ISO/IEC 14882:1998(E). *Information Technology—Programming Languages—C++, First Edition*, 1998. Available at http://www-d0.fnal.gov/~dladams/cxx_standard.pdf.

[5] ISO/IEC 14977:1996(E). *Information Technology—Syntactic Metalanguage—Extended BNF*. Available at http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[6] ISO/IEC 8652/1995(E) with Technical Corrigendum 1. *Consolidated Ada Reference Manual. Language and Standard Libraries*, 2006.

[7] S. C. Johnson. *YACC—Yet Another Compiler Compiler*. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[8] Paul Klint et al. *Rascal Tutor*. CWI, SWAT, 2011. http://tutor.rascal-mpl.org/Courses/Rascal/Rascal.html.

[9] Jan Kort, Ralf Lämmel, and Chris Verhoef. The Grammar Deployment Kit. In M. G. J. van den Brand and R. Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[10] R. Lämmel and C. Verhoef. VS COBOL II Grammar Version 1.0.4. Available at www.cs.vu.nl/grammars/browsable/vs-cobol-ii, 1999.

[11] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[12] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[13] R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proceedings of 7th International Conference on Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.

[14] Ralf Lämmel and Vadim Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal*, 19(2):333–378, March 2011.

[15] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[16] M. P. A. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 151–160. IEEE Computer Society Press, March 2000. Available at http://www.cs.vu.nl/~x/cale.

[17] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In M. P. A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Berlin, 1997. Springer-Verlag.

[18] Niklaus Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[19] Vadim Zaytsev. Correct C# Grammar too Sharp for ISO. In *Preproceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005), Part II, Participants Workshop*, pages 154–155, Braga, Portugal, July 2005. Technical Report, TR-CCTC/DI-36, Universidade do Minho. Extended abstract.

[20] Vadim Zaytsev. Software Language Processing Suite. Grammar Zoo. http://slps.sf.net/zoo, 2009–2011.

[21] Vadim Zaytsev. *Recovery, Convergence and Documentation of Languages*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 2010.

[22] Vadim Zaytsev. MediaWiki Grammar Recovery. *Computing Research Repository*, abs/1107.4661:1–47, July 2011.

[23] Vadim Zaytsev. Software Language Processing Suite. Grammar Tank. http://slps.sf.net/tank, 2011.

[24] Vadim Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In *Proceedings of the 27th ACM Symposium on Applied Computing (SAC'2012), Technical Track on Programming Languages*, March 2012. To appear.