BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions*

Vadim Zaytsev
Software Analysis and Transformation Team, Centrum Wiskunde en Informatica,
Amsterdam, The Netherlands
vadim@grammarware.net

ABSTRACT

Reusing existing grammar knowledge residing in standards, specifications and manuals for programming languages, faces several challenges. One of the most significant of them is the diversity of syntactic notations: without loss of generality, we can state that every single language document uses its own notation, which is more often than not, a dialect of the (Extended) Backus-Naur Form. In this paper we report on an approach to solve the diversity problem by providing a way to quickly and concisely specify all the parameters of a syntactic notation. The resulting "meta-ebnf" language was used to successfully recover many grammars from sources that use different syntactic notations.

Instead of adding another syntactic notation and arguing about its excellence, we propose to retain the diversity and to cope with it by formally defining syntactic notations and using such definitions to import existing grammars to grammar engineering frameworks and to export (pretty-print) existing grammars to any desired syntactic notation. This result effectively bridges programming language standards and parser generators. The conclusions presented in the paper, were drawn based on analysis of a large corpus of language documents, as well as on the success of its application in practice.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Grammarware*

General Terms

Design; Documentation; Languages; Reliability

*The title is a homage to an omnipresent graffiti sticker stating that "BNE WAS HERE". The identity of BNE remains unknown, unlike BNF which stands for Backus-Naur Form. The second part of the title is a direct reference to [26] which first described the problem we are solving in this paper.

SAC'12 March 26–30, 2012, Riva del Garda, Italy.
DOI TBA.

Keywords

EBNF, syntactic notations, metasyntax, grammar recovery, language documentation

1. INTRODUCTION

In this paper we present a set of constructs and conventions, the combination of which full defines an EBNF-like syntactic notation to an extent of enabling automated grammar processing. Currently formal grammars in most programming languages standards and reference manuals are specified using a notation specific to that one particular standard or reference. In fact, all these notations stem from the same root, namely Backus-Naur Form [2, 16], and are technically dialects thereof. It has been noted as early as in 1977 that the diversity of notation for syntax definitions is unnecessary [26], but as of today little has been done to minimize the diversity and to deal with it effectively. There was an attempt in 1996 to standardize the notation at ISO [11], but it only ended up adding yet another three dialects to the chaos.

We have analyzed a corpus of **38** programming language standards (ANSI, ISO, IEEE, W3C, etc), **23** grammar containing publications of other kinds (non-endorsed books, scientific papers, manuals) and **8** derivative grammar sources, exhibiting in total **42** syntactic notations while defining **77** grammars (from Algol and C++ to SQL and XPath). It quickly became apparent that a unified fully automated grammar extractor is impossible to construct, since semantic inference is impossible (e.g., "`a=b,c`" can define `a` as a sequence of `b` and `c` in one notation and assume a terminal symbol `","` between `b` and `c` in another).

After proposing a way to define every specific syntactic notation explicitly and concisely, we were able to automate the rest of grammar recovery activities and build a fault tolerant extractor which helped us to recover **64** grammars of industrial size (some of them containing over 300 nonterminal symbols and over 700 production rules) with minimum effort. This is a drastic improvement on prior work where every grammar recovery initiative took considerable individual effort, which could not be easily re-used in a similar project. Encapsulating syntactic notation details in a concise specification allows us to make generalizations and combines well with advanced error recovery techniques similar to ones presented in [19] or [20].

2. MOTIVATION

Consider the following scenarios where specifying a syntactic notation can be useful:

Grammar extraction from documentation.
Often information about a programming language is available in a form of a manual or an officially approved standard. This information can be extracted and used in many activities involving analysis and manipulation of source code in that language. If the syntactic notation used to define the language is either already specified or can easily be specified in such a way that the extraction process is automated, no time is wasted on developing an individual parser. Manual extraction is possible as well, but also it is a process that is tedious, error-prone and unrepeatable.

Generating correct documentation.
When grammar fragments, and potentially other parts of language documentation, are extracted and processed, they can evolve within a suitable environment to improve their quality, to update contents, to co-evolve with other artifacts, etc. After that, one wants to produce the documentation in more or less the original form, but using the corrected grammar as a baseline. Some research attempts were made at providing a general infrastructure for that [27], but none so far has addressed the issue that the grammar in the documentation should use the original syntactic notation. If that notation definition is available, it can be used to pretty-print grammars automatically.

Linking documentation to compilers.
Having a specification for a syntactic notation used by a parser generator, we can automate generation of parser specifications from grammars. In many cases the generated specifications will need to be enhanced with idiosyncratic extensions, but mapping all features of metaprogramming frameworks is a different (and sometimes undecidable) problem which we leave outside the scope of this paper.

Grammar readability.
The same grammar can be easily pretty-printed in different syntactic notations for different users, allowing them to focus on the content and to abstract from the form(alism) by using the most familiar one.

Syntactic notation evolution.
Programming languages evolve over time, so do their grammars and tools around them, but so far there was no mechanism to aid the evolution of their syntactic notations. This scenario can be considered as a case of re-pretty-printing, but it does not have to stay within those boundaries: think of the co-evolution of grammars with their notations — e.g., introducing a star repetition will automatically refactor all production rules with explicit right or left recursion.

Documentation completion.
In many standards and specifications of programming languages the sections describing the syntax notation are either incomplete or lacking altogether. Having a specification for their syntactic notation of such quality that all grammar fragments are parseable with it, exhaustively proves that the specification is complete; with some simple automated analyses we can check for minimality as well. When such a specification is available, the notational description can be derived from it.

Grammar comparison.
Various techniques exist for comparing grammars, and they are quite powerful in a sense that they can detect and uncover many mismatches in naming, in style, in folding and factoring, etc. However, if we take grammars of the same language (say, Java or Cobol) from two different sources, a part of observed mismatches will be dictated by differences in syntactic notations used by the developers of the source grammars — they can be pointed out only if the specifications of those notations are available.

The abovementioned ISO standard of EBNF [11] made an attempt to (informally) identify various constructs commonly encountered in syntactic notations, but was not successful for a number of reasons, of which we list a few:

- The conclusions were drawn based on analysis of three syntactic notations taken from definitions of Algol [2], Cobol [12], Fortran [9], and an assumption that "most other languages use a variant of one of these metalanguages" [11, p.vi]. Two syntactic notations of these three are disregarded there for understandability and convenience, and the third is misquoted ("`::=`" instead of "`:≡`"). So effectively, the generalizations were made based on one imprecise understanding of a single syntactic notation.
- A reported collision of nonterminal marking metasymbols (angle brackets: "`<`" and "`>`") with possible usages of inequality signs in the language itself, is easily resolved by quoting terminals (which is proposed anyway in the standard, and is relatively common in practice, as we report below).
- Most double-character tokens like "`(/`" introduced by the standard are not currently in use by any existing syntactic notation and are therefore not intuitively understood by contemporary grammarware engineers.
- There is more acute demand for ways to specify a particular syntactic notation and for tools to cope with their abundance (automated mapping, universal import, etc) than for introducing a yet another syntactic notation, even if it will be used in all ISO language standards (which ISO EBNF is not).

3. DEFINING SYNTACTIC NOTATION

Inspired by the list of possible applications provided in the previous section, and having noted the reasons for ISO EBNF failure, we have assembled a larger set of existing language standards and specifications for analysis, developed a domain specific language for specifying differences between them and used that language as a foundation for a toolkit successfully applicable to a range of grammar recovery scenarios. The rest of this section is dedicated to reporting our observations to motivate our design decision in developing that language. It was not our direct goal to report statistical results, but we do present them when it helps to convey our point.

3.1 Grammar integration

When the first notation for metalinguistic formulae was introduced around 1960 [2], there were two issues Backus (understandably) did not consider: incorporation of grammar fragments in language documentation and incorporation of textual annotations in grammar fragments. Both issues are of little importance when the goal is to present a handcrafted grammar as a part of manually written document. However, in the last decades an engineering approach is being taken in software development and language engineering [15, 17, 27]. When a grammar becomes a full-fledged software artifact, incorporated in the whole language-based infrastructure, executable and automatically analyzable and transformable, one needs reliable ways to integrate and to annotate it.

In order to identify the island [22] with the grammar fragment in the textual water surrounding it, we mostly need to know two delimiters: a *start grammar metasymbol*, and an *end grammar metasymbol*. Not all documents in our corpus had those metasymbols even implicitly defined, most of the time one needs to consider indentation and other source-specific details: for instance, the Smalltalk standard formats its grammar fragments in framed boxes [24].

Out of 42 syntactic notations we have examined, 30 did not have any explicit annotations allowed within grammar fragments, 2 had notation for one line comments, 7 for multiline comments and 3 for both. Thus, we should be able to record a *start comment metasymbol*, an *end comment metasymbol*, and a *start one line comment metasymbol*.

3.2 Tokens

Before a grammar can be parsed as such and split into production rules, we can see it as a stream of tokens, which later will become identified as nonterminals, terminals and metasymbols.

Whitespace reliability is assumed when any two separate tokens are always separated by a space or a newline (or any other typical whitespace character). Whitespace unreliability is assumed when separate tokens are commonly glued together — i.e., one should take another route of transforming a character stream into a token stream.[1] Strictly speaking, most of the grammars found in our corpus are neither, they linger somewhere in between. However, it turns out that all borderline cases are handled well by other heuristics (explained in the next sections), but the decision whether to depart from an assumption of whitespace (un)reliability should come from a human grammar engineer.

The whole software language theory is mostly based on context free grammars, with some exceptions. We have spotted at least two context dependent kinds of notational rules: line continuations and masked terminals. A *line continuation metasymbol* is especially important for syntactic notations that rely on formatting and indentation: when a line is physically too long, one has to use a newline character, but it can be agreed that using it in a specific context has no meaning other than technical. For instance, the Fortran standard uses a halmos (■) at the end of the unfinished line and a halmos at the beginning of the next line to denote that the newline between them should be neglected [9].[2] By *masked terminals* we mean any other token sequences that should be treated as a single token (which later becomes a terminal symbol, hence the name). For instance, the Eiffel standard always uses double quote notation for non-alphanumeric terminals (e.g., "`","`"), unless the terminal is a double quote, in which case it is "`'"'`" [13].

Similar to a global decision of whitespace (un)reliability, mentioned above, an important design decision for a syntactic notation is *indentation*. 8 sources from our corpus require indentation information for successful grammar recovery (most use it to separate right hand side alternatives in production rules, see below), another 16 make it possible to extract information from indentation, and 18 do not rely on any indented notation agreements. In the case where indentation is considered, a *tabulation metasymbol* needs to be specified (usually it consists of 4 or 8 spaces, rarely of a true horizontal tabulation character).

3.3 Production rules

In order to map a token stream to a list of grammar production rules, we reuse defining symbol, terminator symbol and definition separator symbol from ISO EBNF [11]. The rest of this section briefly explains our view on them, some extensions we propose, as well as the syntactic notation details for production labels.

A *terminator metasymbol* is used to signal the end of a production rule, or possibly a boundary between two production rules. Most (20) of syntactic notations from our corpus have an empty line or just a newline as a terminator metasymbol, 12 use a semicolon ("`;`"), 5 use a dot ("."), 1 use a comma ("`,`") and 5 have no reliably identified terminator metasymbol.[3] We also found the need to specify a *possible terminator metasymbol* for some more lax syntactic notations that have a terminator convention that is utilized inconsistently.

A *defining metasymbol* is always placed between a left hand side (a nonterminal being defined) and a right hand side (a defining expression) of a production rule. All syntactic notations from our corpus have a defining metasymbol: the most commonly found ones are "`::=`" or "`:≡`" (15 times), followed by "`:`" or "`::`" (12), "`=`" (10) and an arrow ("`->`", "`>`" or "→", 5 times in total). The most exotic defining metasymbol is "≙" from the Eiffel standard [13]. Generally speaking, both terminator and defining metasymbols are not necessary for successful grammar recovery. Either is enough, and statistical analysis can be used to infer hypotheses about them even if none are known. However, specifying them explicitly helps with additional verification.

At least 4 sources from our corpus have a special metasymbol that was not anticipated by ISO EBNF: we call it a *multiple defining metasymbol*. It usually looks like an ordinary defining metasymbol with "one of" or "oneof" appended to it, and it changes the semantics of the following symbols: they are to be treated as a choice, not as a sequence. For example, if "a : b c;" defines `a` as `b` followed by `c`, "a : one

[1] The most efficient technique we have found is to assume token boundaries at places where an alphanumeric character is followed by a non-alphanumeric one, or vice versa (i.e., "`abc?def`" should be seen as a sequence of "`abc`", "`?`", "`def`").

[2] It should be noted here that the Fortran standard uses this notation inconsistently and utilizes other line continuation notations as well, without explaining them.

[3] When the numbers do not sum up to 42, it means some sources had multiple options possible. In this particular case, ISO EBNF §8.1 allowed for either "`;`" or "`.`" as a terminator metasymbol [11, p.8].

of b c;" defines `a` as either `b` or `c`.

A *definition separator metasymbol* separates multiple definitions of a nonterminal: in fact, it designates a top level choice. We found out that it is either based on indentation (in 7 cases), or is represented by a vertical bar ("`|`", 31 cases) and in that case equal to a metasymbol for inner choice (if any). Very rare exceptions look either like a bar ("`/`", "`!`") or like textual disjunction ("`or`", "$\overline{\texttt{or}}$").

Very rarely — in fact, we encountered it only once with SDF [7] — the "left" hand side and the "right" one are swapped, and productions start with a defining expression, followed by a defining metasymbol, and then by a nonterminal being defined. Hence, we need to specify the *definition direction* of the syntactic notation.

Most (34) of the sources from our corpus do not name their productions. However, there are 4 that give every production rule a unique name and 4 that have names in some of their productions. Hence, we need an (optional) *start label metasymbol* and an *end label metasymbol* to record the notation for labeling production rules. During grammar recovery production labels can be preserved or disregarded.

3.4 Nonterminal symbols

In general, any symbol in the grammar can be comprehended as having one of three roles: a terminal symbol, a nonterminal symbol or a metasymbol. During the process of grammar recovery, one more class is possible for unknown symbol type, and by the end of extraction all unknown symbols should either assume one of the basic three roles, or be deemed negligible and disregarded. This section will concern itself with nonterminal symbols, or nonterminals.

In order to identify nonterminals in context, we can use a *start nonterminal metasymbol* and an *end nonterminal metasymbol*: most (33) of syntactic notations that we have seen do not use them, but adopt a special naming convention instead that can be used to tell nonterminals from terminals. 8 remaining syntactic notations use angle brackets ("⟨" and "⟩"), plus the Smalltalk standard uses double angle brackets, single angle brackets or no brackets at all to mark different kinds of nonterminals [24]. Another useful assumption in identifying nonterminals during grammar recovery is creating a dictionary with all left hand sides: if a particular token occurs in that dictionary, we have high confidence that it is indeed a reference to that nonterminal — hence, we speak of a *nonterminal if defined* convention.

The allowed character set for nonterminal names can vary greatly: 14 syntactic notations that we have examined, allow spaces, 8 allow dashes ("`-`"), 5 allow underscores ("`_`") and 1 (LLL [18]) allows slashes ("`/`"). Of course, allowing, say, a space to be a part of a nonterminal name implies either explicit start and end metasymbols as discussed above, or concatenation metasymbol that we will discuss below. In fact, some characters can be allowed (and extensively used) in nonterminal names, but not allowed (or not occurring) in terminal names, and recording this convention explicitly can be of great help in grammar recovery. Hence, we speak of characters a nonterminal name *may contain*, and of characters that we can conclude a *nonterminal if contains*.

Naming conventions can be much more sophisticated than just allowing extra characters in the names. In modern standards nonterminals are distinguished by printing them in different color (e.g., in blue in the Eiffel standard [13]), font variant (e.g., italics in 8 different syntactic notations), etc. Case is of particular importance: in 10 syntactic notations in our corpus *camelcase*, *mixed case* and capitalized tokens are exclusively nonterminals, in 7 notations *lowercase*, and in 2 cases *uppercase* ones.

Built-in nonterminals can be used in a syntactic notation as predefined nonterminals or constructions like an *empty sequence notation*, that in the formal language theory is traditionally written as ε, λ, or ϵ. For instance, the Scheme standard denotes it as "⟨`empty`⟩" [8]. Special entities that are not necessarily like nonterminals in some aspects, can have a special notation for them, usually with a *start special metasymbol* and an *end special metasymbol* — we have encountered them at least 18 times in our corpus. Special entities can define anything that was not covered by regular features provided by a syntactic notation, examples include "any character except ..." (many occurrences), "implementation defined ..." [24], "as specified by standard ..." [1].

3.5 Terminal symbols

Just like nonterminals, terminals also need to be identified in context, which is quite popularly done with a *start terminal metasymbol* and an *end terminal metasymbol*: 12 of syntactic notations that we have seen used double quotes, 6 used single quotes and 2 had both, leaving 22 syntactic notations without explicitly marked terminal symbols. Similar to a "nonterminal if defined" convention advocated in the previous section, we found it reasonable and profitable to have a *terminal if undefined* convention that can rid us automatically of many typesetting mistakes typically encountered during grammar recovery. Even when that convention is not in use, any tokens that contain characters that are not allowed in nonterminal names and that cannot be identified as metasymbols, are bound to become terminals.

Naming conventions also are common practice for terminals. In modern standards terminals are distinguished by printing them in different color (e.g., in green in the Eiffel standard [13]), font face (e.g., fixed width in the C++ standards [10]), font variant (e.g., bold in 6 different syntactic notations), etc. Depending on the language being defined, case can be of importance: in 6 syntactic notations in our corpus *uppercase* tokens were exclusively nonterminals, in 2 cases (Eiffel [13] and Algol [2]) *lowercase* and in 1 case (Algol [23]) *camelcase* and capitalized ones.

Quality of some grammars, especially those found in sources with unreliable whitespace, can be increased substantially by an agreement to *glue consecutive terminals*: this heuristic is damaging only to constructs like "... `":"` `"("` ... `")"` ...", which are uncommon and if present, can be fixed with post-extraction grammar transformation.

3.6 Symbol combinations

Just like with boundaries between consecutive tokens, boundaries between consecutive symbols can also be an issue, especially if the syntactic notation lacks a set of start and end metasymbols for terminals and nonterminals. The ISO standard for EBNF [11] and other sources, for a total of 5 syntactic notations, propose to have an explicit *concatenation metasymbol* that must be placed between any two adjacent symbols (all of them agree on a comma, but this is purely coincidental). Usually the concatenation metasymbol is not placed immediately before or after other metasymbols.

Extreme cases of unreliable whitespace, combined with inability to distinguish font variants during grammar re-

covery, can lead to situations when several consecutive symbols are perceived to be glued together and appear to be one symbol. In that case *decomposition of symbols* is necessary, which is a heuristic technique used to propose hypotheses about a suspicious symbol (e.g., an undefined nonterminal) being split into several symbols that make more sense in context. For example, a definition of `block_statement` copied from the Ada 2005 standard [14, p.683] contains `declaredeclarative_part` and `handled_sequence_of_statementsend`, in both instances newlines being processed incorrectly, leading to terminals `declare` and `end` being erroneously concatenated with the names of valid nonterminals defined elsewhere. Similar problems occur in abundance in the C++ standard [10].

When any kind of grouping symbols is possible, there may be a need for an *inner choice metasymbol*. However, it can be present in a syntactic notation and not be equal to a definition separator metasymbol that we mentioned above. For instance, the Java specification [6] uses indentation for top choice and a bar ("`|`") for inner choice.

Denoting optionality of symbols is a rather popular simplifying notation: 13 of the syntactic notations from our corpus preferred a *postfix optionality metasymbol* (6 used "$_{opt}$" and 7 used "`?`") and only 6 did not use any. Of the remaining 24, 20 used square brackets as a *start optionality metasymbol* and an *end optionality metasymbol*, one used round brackets, one — curly brackets and two (both from the EBNF standard [11]) used "`(/`" and "`/)`".

Similarly, two kinds of repetition are commonly encountered in syntactic notations: a "star" (Kleene star, a transitive reflexive closure, zero or more) and a "plus" (a transitive closure, one or more). A star is more common: 10 syntactic notations from our corpus used a *postfix star repetition metasymbol* (variations of `*`, ⋆, *, etc), 14 used curly brackets for a *start star repetition metasymbol* and an *end star repetition metasymbol*, 2 (both from ISO EBNF [11]) used "`(:`" and "`:)`" and 13 did not have any. The remaining three used "`{`" and "`}`..." [23], "`{`" and "...`}`" [4], and "`[`" and "`]`..." [9]. A plus is only present in 14 syntactic notations, out of which 8 have "`+`" a *postfix plus repetition metasymbol*, 2 feature "`...`" in the same role, 3 have "`{`" and "`}+`" as a *start plus repetition metasymbol* and an *end plus repetition metasymbol* and 1 has "`{`" and "...`}+`" [4]. We mention all this in such detail to signify the diversity and the seeming randomness of choice for notation — the main reasons why syntactic notation cannot be inferred automatically and why we have to find means to specify it formally and unambiguously.

Very few relatively modern syntactic notations feature "separator lists": a repetition where a special separator symbol is inserted between any two adjacent list items. In our corpus we have only encountered it three times: as "`{`" and "`}+`" as a *start plus separator list metasymbol* and an *end plus separator list metasymbol* in LLL [18] and SDF [7], and as "{" and "...$\}^+$" in Eiffel [13]. (Same three instances have a similar notation for a star variant of a separator list).

A *start group metasymbol* and an *end group metasymbol* are used to denote a subexpression that is effectively an unnamed nonterminal unfolded in place. This notation is quite extensively used by sources that use postfix optionality and repetition, or have many inner choices. Groups are mostly (18 syntactic notations from the corpus) denoted by round brackets, but square brackets were found in one wiki notation [21] and curly brackets are used in the SQL standard [1].

We found an *exception metasymbol* at least 6 times in our corpus: it is usually infix and means that `a-b` should be possible to parse as `a` but impossible to parse with `b`. The semantics of exception is hard to define and to combine properly with other constructs, which explains its lack of popularity.

4. CONCLUSION AND FUTURE WORK

As it was stated in the introduction, we have analyzed **38** standards, **23** other grammar containing publications and **8** derivative grammar sources, exhibiting in total **42** syntactic notations while defining **77** grammars of Ada, Algol, ANTLR, Basic, BNF, C, C++, C#, Dart, EBNF, Eiffel, Fortran, Java, JavaScript, LLL, Modula, Pascal, Rascal, Scala, Scheme, SDF, Smalltalk, SQL, Wiki, WSN, XPath, and YACC.

We have proposed the way to define any syntactic notation explicitly by specifying:

Confix constructs (start & end metasymbols):
grammar, comment, label, nonterminal, terminal, special, group, optionality, star repetition, plus repetition, star separator list, plus separator list

Infix metasymbols:
terminator, possible terminator, defining, multiple defining, definition separator, concatenation, inner choice, exception

Postfix metasymbols:
optionality, star repetition, plus repetition

Prefix metasymbols:
start one line comment

Other metasymbols:
line continuation, tabulation, empty sequence

Conventions:
whitespace reliability, indentation, definition direction, nonterminal if defined, nonterminal if contains, glue consecutive terminals, decomposition of symbols, uppercase nonterminals, lowercase nonterminals, camelcase nonterminals, mixed case nonterminals, uppercase terminals, lowercase terminals, camelcase terminals, mixed case terminals

Predefined sets:
masked terminals, nonterminals may contain, built-in nonterminals

As a direct result, we were able to automate the rest of grammar recovery activities and build a fault tolerant extractor which helped us to recover **64** grammars of industrial size with minimum effort.[4] **11** of the remaining grammars are in a form less suitable for machine processing (e.g., badly OCR-ed or only in hard copy) and present technical difficulties that we leave out of scope of this report. Only **one** grammar, namely the SDF [7] definition of SDF [25], presents a challenge, and we reserve it for future work. **One** more

[4]All recovered grammars, all definitions of their syntactic notations, as well as the Grammar Hunter tool in its Python and Rascal incarnations are openly released via a complementary website: `http://grammarware.net/did/hunter`.

syntactic notation, namely TXL [5], was excluded from consideration altogether because its notation was too far from any EBNF dialect.

Grammar recovery as performed by Grammar Hunter is largely automatic, with its robustness assisted by the specification of a syntactic notation (a few user provided indications). This implies that recovering future versions of the same grammars or grammars of other languages using the same notations as some of these, becomes a matter of obtaining access to them and re-running Grammar Hunter.

One of the future research directions for us is to find a generic syntactic notation, which will be extensible on a meta-level: i.e., it will be possible to stay within the notation while introducing new constructs of metasyntactic sugar or even completely new ones. The particular challenge here is that most notations that stem from introducing arbitrary extensions, turn out to be too powerful (e.g., the exception metasymbol allows to go beyond context-free grammars and define undecidable loops; introduction of meta-identifiers can take us to Turing complete van Wijngaarden grammars, etc).

Identifying grammar fragments related to one specific topic (i.e., statements) is commonly encountered in language documents, but as of now, lacks complete technological support because few grammar manipulation frameworks have sufficiently advanced modularity. However, we should consider this functionality in our future work and deeply investigate its implementability.

Another possible research direction based on our result concerns detailed analysis of notation for syntactic definitions from the expressivity perspective. Currently neither grammar classes based on Chomsky hierarchy nor parsing technology driven grammar classes like LL(k) or LALR(1) have any correspondence with classes of grammars that are possible to define with a given metasyntax. This misalignment leads to excessive additional checks the parser generator must perform in order to report various errors and warnings, which still does not halt the existence of ambiguous grammars (cf. [3]). Without research it is impossible to even predict whether creating a syntactic notation that can only be used to define, say, GLL grammars, is reachable, feasible and useful.

5. REFERENCES

[1] ANSI/ISO/IEC 9075-1:1999(E). *Information Systems. Database Language SQL. Part 1: Framework*, 1999.

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

[3] H. J. S. Basten. Tracking Down the Origins of Ambiguity in Context-free Grammars. In *ICTAC'10*, pages 76–90, Berlin, Heidelberg, 2010. Springer.

[4] E. Bezault. Eiffel: The Syntax. `http://www.gobosoft.com/eiffel/syntax`, 1999.

[5] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Grammar Programming in TXL. In *SCAM'02*. IEEE, 2002.

[6] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005. `http://java.sun.com/docs/books/jls`.

[7] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF. Reference Manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[8] IEEE Std 1178–1990. *IEEE Standard for the Scheme Programming Language*, Approved December 10, 1990. Reaffirmed December 12, 1995.

[9] ISO 1539:1980, ANSI X3J3/90.4. *Information Technology. Programming Languages. Fortran*, 1980.

[10] ISO/IEC 14882:1998(E). *Information Technology. Programming Languages. C++*, 1998.

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

[12] ISO/IEC 1989:1985. *Information Technology. Programming Languages. COBOL*, 1985.

[13] ISO/IEC 25436:2006(E). *Information Technology. Eiffel: Analysis, Design and Programming Language*, first edition, 2006.

[14] ISO/IEC 8652/1995(E), Ed. 3. *Information technology. Programming Languages. Ada. Consolidated Ada Reference Manual*, 2006.

[15] P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM ToSEM*, 14(3):331–380, 2005.

[16] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Commun. ACM*, 7(12):735–736, 1964.

[17] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[18] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. G. J. van den Brand and R. Lämmel, editors, *ENTCS 65*. Elsevier, 2002.

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

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

[21] MediaWiki. Markup spec. BNF. Noparse block. `http://www.mediawiki.org/wiki/Markup_spec/BNF/Noparse-block`, 2010.

[22] L. Moonen. Generating Robust Parsers using Island Grammars. In *WCRE'01*, pages 13–22. IEEE Computer Society Press, Oct. 2001.

[23] V. M. Pentkovsky. *Elbrus Autocode. El-76. Language Design Principles and User Manual*. Nauka, 1982.

[24] Y.-P. Shan, G. Krasner, B. Schuchardt, and R. DeNatale. *NCITS J20 DRAFT of ANSI Smalltalk Standard, Revision 1.9*, December 1997. Available at `http://wiki.squeak.org/squeak/uploads/standard_v1_9-indexed.pdf`.

[25] J. Vinju, G. R. Economopoulos, and P. Klint. SDF2 defined in SDF. Available in the `sdf-library` of `http://www.meta-environment.org`, 2006–2010.

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

[27] V. Zaytsev and R. Lämmel. A Unified Format for Language Documents. In B. Malloy, S. Staab, and M. G. J. van den Brand, editors, *SLE'10, LNCS 6563*, pages 206–225, Berlin, Heidelberg, 2011. Springer.