Grammar Normalization to Support Automated Merging

Rosetta Roberts and Isaac Griffith
Empirical Software Engineering Laboratory
Informatics and Computer Science
Idaho State University
Pocatello, Idaho, 83208
Email: {roberose@isu.edu, grifisaa@isu.edu}

***Abstract*—Introduction:**
Objective:
Methods:
Results: What are the main findings? Practical implications?
Limitations: What are the weaknesses of this research?
Conclusions: What is the conclusion?
***Index Terms*—Island Grammars, Automated Grammar Formation, Software Language Engineering**

I. Introduction

Software applications written in multiple programming languages have become more common. These multilingual software systems introduce additional tool development challenges. One such challenge is building parsers that allow multilingual-capable tools [1]. Currently, such tools use a separate parser or grammar for each supported language, or they use a limited intermediate representation (IR). IR-based tools are usually constrained to an ecosystem such as .NET or the JVM [2], while tools using multiple parsers are typically time-consuming to maintain [3].

One method of multilingual parsing uses specially constructed island grammars [4]. These island grammars identify only the portions of documents that are of interest to the application. These island grammars are still able to function in the presence of multiple languages. However, this requires manually combining portions of grammars. The goal of this research is as follows:

RQ1 Develop an approach which reduces the effort necessary for automatically merging grammars.

Apropos this goal, we've designed an algorithmic procedure of normalizing grammars to a form suitable for automated merging.

This paper presents this approach and is organized as follows. Sec. II discusses the theoretical foundations related to this work and the notation this paper uses. Sec. III details how the normalization procedure was designed, the meta-model used to represent grammars as they are transformed, and the algorithms used to perform the normalization. Sec. IV selects three grammars for a pilot study demonstrating the normalization process. The results of the pilot study are in Sec. V. Finally, sec. VII concludes this paper with a summary and description of future work.

II. Background

Context-free grammars are defined by $G = (V, \Sigma, P, S)$ [5]. V is the set of non-terminal symbols, Σ is the set of terminal symbols, P is the set of productions, and $S \in V$ is the start symbol [5]. Extended Backus-Naur Form (EBNF) is used to represent productions in this paper [6]. Each production is written as $\Phi \rightarrow R$ where Φ is a non-terminal symbol and R is an expression representing a rule. Each expression can be either a symbol, the empty string (ϵ), or expressions combined with an operator.

The concatenation operator is represented by concatenating the operands (e.g. ⟨A⟩ `a`). The alternation operator is represented by | seperating its operands (e.g. ⟨A⟩ | `a`). Parenthesis are used to delimit expressions when doing so prevents ambiguity (e.g. ⟨A⟩ (`a` | ϵ)). ⟨U⟩pper case letters enclosed within angle brackets denote non-terminal symbols while lowercase symbols in `monospace font` denote terminal symbols.

ISLAND GRAMMARS

CHOMSKY Normalization and a couple others - Appropriate for parsing, not particularly so for merging - Doesn't treat associativity of operators specially which can result in multiple results.

III. Approach

This section details the design, meta-model, and implementation of a normalization algorithm to simplify the automated merging of grammars. The automated merging of grammars requires that each of the productions of a grammar be normalized to one of two specific forms:

Form$_1$ The rule of the production only uses the concatenation operator to concatenate symbols. E.g. $\langle A \rangle \rightarrow \langle B \rangle\ b\ \ldots$.

Form$_2$ The rule of the production only uses the alternation (|) operator to combine symbols or the empty string. E.g. $\langle A \rangle \rightarrow \langle B \rangle \mid b \mid \epsilon \mid \ldots$.

Also, the grammar is constrained such that (i) each symbol in a rule cannot have the same form as the rule it is in, (ii) unit rules are not permitted except for the case of the start symbol producing a single terminal symbol, and (iii) no pair of productions have identical right-hand sides.

Algorithm 1 Simplify Productions

```
1: function SIMPLIFYPRODUCTIONS(G)
2:     for all ≀ ∈ OPERATORNODES(G) do
3:         if ISCONCATOPERATOR(≀) then
4:             for all {p ∈ OPERANDS(≀) | p = ϵ} do
5:                 REMOVEOPERAND(p)
6:             end for
7:         end if
8:     end for
9:     return G
10: end function
```

Algorithm 2 Collapse Productions

```
1: function COLLAPSEPRODUCTIONS(G)
2:     for all (p1, p2) ∈ ORDEREDPAIRS(G.P) do
3:         ≀1 ← ROOTOPERATOR(p1)
4:         φ2 ← SYMBOL(p2)
5:         if φ2 ∈ CHILDREN(≀1) then
6:             ≀2 ← ROOTOPERATOR(p2)
7:             if ASSOCIATIVE(≀1, ≀2) then
8:                 ≀1.REPLACECHILD(φ2, CHILDREN(≀2))
9:             end if
10:        end if
11:    end for
12:    return G
13: end function
```

Algorithm 3 Eliminate Unit Productions

```
1: function ELIMINATEUNITPRODUCTIONS(G)
2:     for all p ∈ G.V \ {G.S} do
3:         if ISSYMBOL(RULE(p)) then
4:             REPLACEUSES(p, RULE(p))
5:         end if
6:     end for
7:     if ISNONTERMINALSYMBOL(RULE(G.S)) then
8:         REPLACEUSES(G.S, RULE(G.S))
9:     end if
10:    return G
11: end function
```

Algorithm 4 Merge Equivalent Productions

```
1: function MERGEEQUIVPRODUCTIONS(G)
2:     for all {p1, p2} ∈ UNORDEREDPAIRS(G.P) do
3:         if RULE(p1) = RULE(p2) then
4:             ρ ← COMBINESYMBOLS(p1, p2)
5:             G.REPLACEUSES(p1, ρ)
6:             G.REPLACEUSES(p2, ρ)
7:         end if
8:     end for
9:     return G
10: end function
```

A. Normalization Foundations

The constraints were chosen through the identification and resolution of five different issues. These problems revolved around equivalent ways of writing rules that make it difficult for a merging procedure to detect and then combine similar productions across grammars.

The first issue is spurious usage of the empty string. An example of this would be the production $\langle \text{A} \rangle \rightarrow \langle \text{B} \rangle \ \epsilon \ \texttt{b}$. The empty string in the middle of the expression is unnecessary. These unnecessary empty strings are eliminated in Algorithm 1.

The second problem arises from the associative property of the concatenation and alternation operators. For example, productions S_1 and S_2 are equivalent in the following productions.

$$\begin{aligned} \langle S_1 \rangle &\rightarrow \texttt{a} \ \langle \text{B} \rangle \\ \langle \text{B} \rangle &\rightarrow \texttt{b c} \\ \langle S_2 \rangle &\rightarrow \langle \text{A} \rangle \ \texttt{c} \\ \langle \text{A} \rangle &\rightarrow \texttt{a b} \end{aligned}$$

To address this, we added the constraint that a production cannot use non-terminal symbols that are the same form. Algorithm 2 of the normalization process transforms each grammar to meet this constraint.

The commutative property of the alternative operator causes the third problem. For example, the following two productions are identical, but are represented differently:

$$\begin{aligned} \langle \text{A} \rangle &\rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \\ \langle \text{A} \rangle &\rightarrow \texttt{c} \mid \texttt{b} \mid \texttt{a} \end{aligned}$$

We considered two different approaches to removing this ambiguity. The first approach involved lexicographically ordering the terms. This is unwieldy because it is difficult to define a lexicographic ordering. Instead, we adjusted our domain object model to use a set container for the terms. This eliminated the problem by removing the ordering from our representation.

The fourth issue is the use of unit productions. A unit production is a production where the right hand side is a single symbol. Here is an example:

$$\begin{aligned} \langle \text{A} \rangle &\rightarrow \langle \text{B} \rangle \\ \langle \text{B} \rangle &\rightarrow \texttt{a b c} \end{aligned}$$

In almost all cases, this is better represented by removing the unit production. The one exception is when the start symbol directly produces a single terminal symbol. Removal of unit productions is performed in Algorithm 3.

The fifth and last issue is caused by duplicate productions. Duplicate productions result in multiple symbols producing the same rule. Replacing these multiple symbols with the same symbol can enable other simplifications. In the following example $\langle \text{A} \rangle$'s production can be reduced to a unit rule by replacing the symbols $\langle \text{B} \rangle$ and $\langle \text{C} \rangle$ with a single symbol:

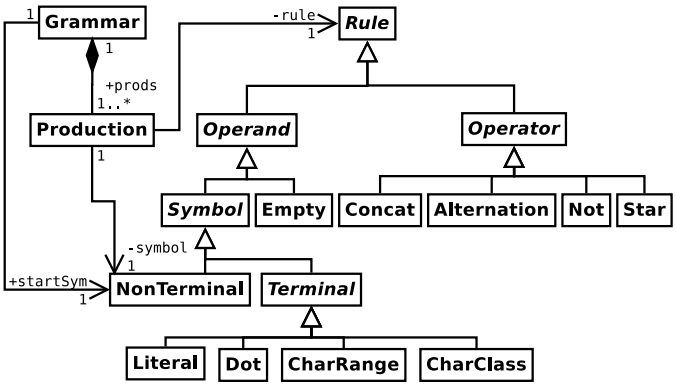

Fig. 1. Class diagram of grammar meta-model.

TABLE I
ANTLR FEATURES NOT IN BNF

construct	example	description
$+$	a$+$	one or more repetitions
$*$	a*	zero or more repetitions
$\sim$	$\sim$ (a|b)	not one of a set of characters
?	a?	zero or one repetitions
'.'	'.'	any character
...	$a \ldots z$	a character range
$\backslash p\{\}$	$\backslash p\{Symbol\}$	a character class

$$\begin{aligned} \langle \text{A} \rangle &\rightarrow \langle \text{B} \rangle \mid \langle \text{C} \rangle \\ \langle \text{B} \rangle &\rightarrow \texttt{a b c} \\ \langle \text{C} \rangle &\rightarrow \texttt{a b c} \end{aligned}$$

Because of this, we add the condition that normalized grammars cannot have two different symbols that produce the same expression. Detection and removal of duplicate productions is detailed in algorithm 4 of the normalization process.

B. Domain Object Model

The meta-model, depicted in fig. 1, representing our grammars mirrors the structure of BNF grammars. The right-hand side of each production with an expression tree of operands and operators.

Additional operator and operand types can easily be added to extend our model. For example, we defined operators and operands specific to ANTLR [7]. Table I shows ANTLR features that we took care to represent correctly.

To support these features, we implemented additional terminal types for `.`, character ranges, and character classes. We also added operators for $*$ and $\sim$.

The + and ? operators are instead substituted with equivalent expressions while being parsed. Expressions of the form $\mathcal{A}+$ are replaced with $\mathcal{A}\ \mathcal{A}^*$, while expressions of the form $\mathcal{A}?$ are replaced with $(\epsilon|\mathcal{A})$. $\mathcal{A}$ denotes an arbitrary expression.

Algorithm 5 Normalization Algorithm

```
1: procedure NORMALIZE(G)
2:     repeat
3:         G ← ELIMINATEUNUSEDPRODUCTIONS(G)
4:         G ← SIMPLIFYPRODUCTIONS(G)
5:         G ← MERGEEQUIVPRODUCTIONS(G)
6:         G ← ELIMINATEUNITPRODUCTIONS(G)
7:         G ← EXPANDPRODUCTIONS(G)
8:         G ← COLLAPSEPRODUCTIONS(G)
9:     until UNCHANGED(G)
10:    return G
11: end procedure
```

Algorithm 6 Eliminate Unused Productions

```
1: function ELIMINATEUNUSEDPRODUCTIONS(G)
2:     W ← G.V ∩ DEPTHFIRSTSEARCHFROM(G.S)
3:     Q ← {(w, G.P(w)) | w ∈ W}
4:     H ← (W, G.Σ, Q, G.S)
5:     return H
6: end function
```

After parsing, every application of the $*$ operator is replaced with a new production. Expressions of the form $\mathcal{A}^*$ are replaced with the production $\langle \text{A} \rangle \rightarrow \langle \text{A} \rangle \mid \epsilon$.

Of special note are the empty string operand and the alternation and concatenation operators. Firstly, the empty string operand is the only operand that is not a symbol. Secondly, the alternation and concatenation operators are not binary operators, but rather n-ary operators. Lastly, the alternation operator aggregates its operands in a set object. This removes the ordering information per the second problem identified earlier regarding the commutative property of the alternation operator.

C. Normalization Algorithm

Algorithm 5 defines the approach for normalizing grammars. A grammar representing using our meta-model is the algorithm's input. The algorithm's output is a grammar which is equivalent to the input grammar and also has the properties enumerated in our design process.

The normalization process, as defined in Algorithm 5, repeatedly executes six steps until the grammar stabilizes. These six processes are: i) eliminating unused rules, ii) simplifying productions, iii) merging equivalent rules, iv) eliminating unit rules, v) expanding productions, and vi) collapsing compatible productions.

The first process removes all productions that are not produced, directly or indirectly, from the start production. This is accomplished by enumerating all symbols producible from the start symbol via a depth first search and then creating a new grammar using only the enumerated symbols as shown in Algorithm 6.

The second process simplifies productions by removing unnecessary empty strings (ε). These are those that are operands

Algorithm 7 Expand Productions

```
1: function ExpandProductions(G)
2:     for all p ∈ G.P do
3:         for all ι ∈ NonRootOpNodes(Rule(p)) do
4:             G.ReplaceWithNewRule(ι)
5:         end for
6:     end for
7:     return G
8: end function
```

of the concatenation operator. This process is incarnate in Algorithm 1.

The third process replaces productions that have identical rules with a single production. The algorithm for this is shown in Algorithm 4. The process replaces symbols by scanning the entire grammar and then replacing each old symbol with the new symbol. How the new symbol's name is constructed affects only the readability of the resulting grammar. In our implementation, the names of the old productions are concatenated.

The fourth process removes all unit productions unless the production is the start symbol producing a single non-terminal. To do this, it first identifies unit productions. It then replaces symbols on the left of each production with their right-hand symbols. Algorithm 3 describes this process.

The fifth process converts each production to one of Form 1 or Form 2. Each non-root operator node of the expression tree of the rule is pulled into a distinct production, as presented in Algorithm 7.

The sixth and final step of the normalization process combines associative operators. For BNF grammars, only the concatenation and alternation operators are associative. Algorithm 2 details this step.

IV. Pilot Study

To evaluate the above approach, we performed a small pilot study on three grammars selected from the ANTLR [7] grammar repository[1]. The grammar selection criteria were as follows: (i) selected grammars should be of varying sizes with at least one grammar of a size small enough to be easily inspected, and (ii) grammars should be of varying application. These criteria led to the selection of the Brainfuck, XML, and Java™ grammars.

Brainfuck is an esoteric Turing-complete language notable for its extreme simplicity, having only 8 commands [8]. We chose this language because its grammar is extremely small which allows it to be easily inspected. For similar reasons we chose the more complicated XML grammar. XML is commonly used for sending information between applications [9] and configuration files [**?**]. Finally, Java™ was selected for the high likelihood it would need to be included as a base language [10]–[12]. This language is significantly more complicated than either Brainfuck or XML, as it is a widely used [13] general purpose programming language.

[1]https://github.com/antlr/grammars-v4

$$
\begin{aligned}
\langle \text{file} \rangle &\rightarrow \langle \text{A} \rangle \mid \epsilon \\
\langle \text{statement} \rangle &\rightarrow \texttt{>} \mid \texttt{<} \mid \texttt{+} \mid \texttt{-} \mid \texttt{.} \mid \texttt{,} \mid \langle \text{B} \rangle \\
\langle \text{B} \rangle &\rightarrow \texttt{[} \ \langle \text{file} \rangle \ \texttt{]} \\
\langle \text{A} \rangle &\rightarrow \langle \text{statement} \rangle \ \langle \text{file} \rangle
\end{aligned}
$$

Fig. 2. Brainfuck grammar after normalization. Productions ⟨A⟩ and ⟨B⟩ are productions with auto-generated names.

$$
\begin{aligned}
\langle \text{file} \rangle &\rightarrow \langle \text{statement} \rangle^* \\
\langle \text{statement} \rangle &\rightarrow \langle \text{opcode} \rangle \mid \\
&\quad \langle \text{LPAREN} \rangle \ \langle \text{statement} \rangle^* \ \langle \text{RPAREN} \rangle \\
\langle \text{opcode} \rangle &\rightarrow \langle \text{GT} \rangle \mid \langle \text{LT} \rangle \mid \langle \text{PLUS} \rangle \mid \\
&\quad \langle \text{MINUS} \rangle \mid \langle \text{DOT} \rangle \mid \langle \text{COMMA} \rangle \\
\langle \text{GT} \rangle &\rightarrow \texttt{>} \\
&\ \ \vdots
\end{aligned}
$$

Fig. 3. Brainfuck grammar before normalization. Definitions for the other ops are omitted for brevity.

To evaluate each grammar, we normalize each grammar. Before and after normalization, we measure the number of productions [14]. After normalization, each grammar is checked manually. In this checking process, each rule is verified to be of either $Form_1$ or $Form_2$. In addition, we examine each normalized grammar for unexpected rules.

V. Results

The following paragraphs detail the results of the normalization procedure for each grammar. Presented first is Brainfuck with its full grammar displayed before and after normalization. Next, select portions from the normalized XML grammar are detailed. Finally, the results of transforming Java™'s grammar are shown.

Fig. 3 and Fig. 2 show Brainfuck's grammar before and after normalization. As can be seen, the normalized grammar meets all the conditions required for the normal form. In addition, both $*$ expressions have been replaced. The normalization procedure was able to recognize that the allowed syntax for the inside of square brackets is the same as that of the entire file. Finally, all of the unit rules for the operators have been replaced directly with their text.

The exact results of XML's normalization are not shown because of their length. Table II shows that there was a net increase in the number of productions. This is because productions in the original grammar were broken apart into simpler productions and each use of the $*$ ANTLR feature introduced two rules.

As follows is an example portion of the XML grammar before and after normalization. The production representing XML elements in the pre-normalized grammar is:

TABLE II
THE NUMBER OF PRODUCTIONS IN EACH GRAMMAR BEFORE AND AFTER NORMALIZATION.

Language	Before	After
Brainfuck	12	4
XML	32	52
Java™	222	372

$$\langle \text{element} \rangle \quad \rightarrow \quad \texttt{<} \ldots \texttt{>} \ldots \texttt{</} \ldots \texttt{>} \mid \texttt{<} \ldots \texttt{/>}$$

As can be seen, there are two different variants. The first variant represents XML elements with a closing and opening tag. The second variant represents XML elements with a single self closing tag. In the normalized grammar, each of these two variants were extracted into their own rules:

$$\begin{aligned} \langle \text{element} \rangle &\rightarrow \langle \text{A} \rangle \mid \langle \text{B} \rangle \\ \langle \text{A} \rangle &\rightarrow \texttt{<} \ldots \texttt{>} \ldots \texttt{</} \ldots \texttt{>} \\ \langle \text{B} \rangle &\rightarrow \texttt{<} \ldots \texttt{/>} \end{aligned}$$

Like the XML grammar, Java™'s grammar experienced a significant size increase from the normalization procedure. Part of the reason is because the input Java™ grammar had a significant number of optional expressions: expressions using ?. Each of these were eventually extracted out to a new production, resulting in a large number of productions of the form $\langle \text{A} \rangle \rightarrow \mathcal{A} \mid \epsilon$.

An example of de-duplication involved the production defining a Java™ expression. In this example, the following three rules were combined:

$$\begin{aligned} \langle \text{parExpression} \rangle &\rightarrow \texttt{(} \langle \text{expression} \rangle \texttt{)} \\ \langle \text{expression} \rangle &\rightarrow \langle \text{primary} \rangle \mid \ldots \\ \langle \text{primary} \rangle &\rightarrow \texttt{(} \langle \text{expression} \rangle \texttt{)} \mid \ldots \end{aligned}$$

Note that the first part of ⟨primary⟩'s rule is the same as ⟨parExpression⟩'s rule. In the normalization process, this was detected and the equivalent portion was replaced with the ⟨parExpression⟩ symbol. Finally, ⟨primary⟩'s rule was substituted directly into ⟨expression⟩ and eliminated. This substitution happened because of the constraint that a production cannot reference another production that has the same form. This resulted in the following rules:

$$\begin{aligned} \langle \text{parExpression} \rangle &\rightarrow \texttt{(} \langle \text{expression} \rangle \texttt{)} \\ \langle \text{expression} \rangle &\rightarrow \langle \text{parExpression} \rangle \mid \ldots \mid \ldots \end{aligned}$$

Other substitutions resulted in a small number of productions in the normalized grammar being extremely long. Intriguingly, these long productions were mostly *Form*$_2$. Those of *Form*$_1$ involved either Java™'s try-catch feature or generic method declaration.

Similar to the results of normalizing Brainfuck, a large number of symbols which produced a single token were replaced with that token in Java™'s normalized form. Like as in the XML grammar, expressions involving $*$ were replaced with productions. However, these expressions happened less often than in XML, making them far less noticeable.

VI. INTERPRETATION

Our results show that the normalization process functions as designed. Each of the normalized grammars meets the conditions of the normalization. There was a decrease in number of productions for Brainfuck's grammar and an increase for the XML and Java™ grammars, as show in table II. The increase for the last two grammars was not unreasonable. Most of the increase was caused by the conversion of two specific ANTLR features into equivalent BNF features: (i) the optional qualifier ? and (ii) the zero or more qualifier $*$.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we developed an automatic approach to normalizing grammars. This approach reduces the effort required to automatically merge grammars for the purpose of automatically constructing multi-lingual island grammars. We demonstrated and evaluated this approach via a pilot study on three grammars: Brainfuck, XML, and Java™. The pilot study presents a step towards the automated construction of multi-lingual island grammar based parsers, thereby easing the construction of multi-lingual code analysis tools.

There is a significant amount of future work to be done. This normalization procedure was designed around grammars written in BNF. However, typical applications will likely use grammars written in more complicated forms. Further refinement of the normalization procedure using these features is one such avenue. Currently, our meta-model represents grammars written in BNF, EBNF, and ANTLR. We intend to extend this meta-model to allow representing TXL [15] and SDF [16] grammars. The generated names of symbols created in this normalization process lack readability. An avenue of future research would involve finding ways to generate more useful names. Additionally, this process is currently being integrated as part of an overarching approach to automate the development of multi-lingual parsers through the automated construction of island grammars based on existing grammars.

REFERENCES

[1] Z. Mushtaq, G. Rasool, and B. Shehzad, "Multilingual Source Code Analysis: A Systematic Literature Review," *IEEE Access*, vol. 5, pp. 11 307–11 336, 2017.

[2] P. Linos, W. Lucas, S. Myers, and E. Maier, "A metrics tool for multi-language software," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. ACTA Press, 2007, pp. 324–329.

[3] A. Janes, D. Piatov, A. Sillitti, and G. Succi, "How to Calculate Software Metrics for Multiple Languages Using Open Source Parsers," in *Open Source Software: Quality Verification*, E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 404, pp. 264–270.

[4] N. Synytskyy, J. R. Cordy, and T. R. Dean, "Robust multilingual parsing using island grammars," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 2003, pp. 266–278.

[5] M. Haoxiang, *Languages and Machines: An Introduction to the Theory of Computer Science*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1988.

[6] D. D. McCracken and E. D. Reilly, "Backus-Naur Form (BNF)," in *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., 2003, pp. 129–131.

[7] T. Parr, *The Definitive ANTLR 4 Reference*, ser. The Pragmatic Programmers. Dallas, Texas: The Pragmatic Bookshelf, 2012, oCLC: ocn802295434.

[8] U. Müller, "Brainfuck–an eight-instruction turing-complete programming language," *Available at the Internet address http://en. wikipedia. org/wiki/Brainfuck*, 1993.

[9] E. Cerami, *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. " O'Reilly Media, Inc.", 2002.

[10] B. Kurniawan, *Java for the Web with Servlets, JSP, and EJB*. Sams Publishing, 2002.

[11] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, "An interactive tool for analyzing embedded SQL queries," in *Asian Symposium on Programming Languages and Systems*. Springer, 2010, pp. 131–138.

[12] V. S. Getov, "A mixed-language programming methodology for high performance Java computing," in *The Architecture of Scientific Software*. Springer, 2001, pp. 333–347.

[13] "Stack Overflow Developer Survey 2019," https://insights.stackoverflow.com/survey/2019/, 2019.

[14] J. F. Power and B. A. Malloy, "A metrics suite for grammar-based software," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 405–426, Nov. 2004.

[15] T. Dean, J. Cordy, A. Malton, and K. Schneider, "Grammar programming in TXL," in *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. Montreal, Que., Canada: IEEE Comput. Soc, 2002, pp. 93–102.

[16] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF—reference manual—," *ACM SIGPLAN Notices*, vol. 24, no. 11, pp. 43–75, Nov. 1989.