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: Current solutions to multilingual parsing, for programming languages, are flawed. Current implementations are either limited in scope or difficult to develop and maintain. The development of multilingual parsers requires the combination of multiple base grammars, leading to a maintenance headache as these grammars evolve. Such a repetitive process should be automated. Objective: Develop an approach to normalize a grammar in such a way that the grammar is equivalent to the original, but in a state which reduces the effort to merge grammars by reducing ambiguity in automated merge decision making. Methods: This normalization procedure transforms grammars such that each production is one of two forms. Additionally, the normalized grammars maintain a set of additional constraints we identified as useful. A pilot study demonstrating this approach was conducted on three existing grammars. Results: The normalization algorithm was shown to correctly normalize the three grammars. Conclusions: This work presents a normalization method towards easing the development of automatically merging programming language grammars.**

***Index Terms*—Grammar Normalization, Software Language Engineering**

I. Introduction

It has become more common in the software industry to write applications in multiple programming languages. Such multilingual software systems present tool developers with additional, difficult challenges. One such challenge concerns the development of parsers supporting these tools [1]. Currently, for each supported language, tools use either an ecosystem restricted intermediate representation (IR) based approach or a multiple parser approach.

Though the multiple parser approach is the least limited, it does incur a high maintenance cost [2]. To alleviate this, researchers have turned to the development of techniques for multilingual parsing. One such method uses specially constructed Island Grammars [3]. These Island Grammars identify only the portions of documents that are of interest to the application, while able to function in the presence of multiple programming languages. However, this requires manually combining portions of grammars.

The manual combination of grammars necessary for the creation of a multilingual Island Grammar is time-consuming and prone to a maintenance headache whenever the base language grammars evolve. The obvious answer to such a problem is the development of an automated technique to merge grammars. Unfortunately, this leads to another problem, which is the focus of this research. *Prior to merging, a grammar must be constrained such that its contained productions and each production's contained symbols are arranged in a consistent manner equivalent to the original grammar, allowing for unambiguous merging decisions*. This problem has led to the formation of the following research goal (RG):

RG Develop an approach to normalize a grammar in such a way that the grammar is equivalent to the original, but in a state which reduces the effort to merge grammars by reducing ambiguity in merge decision making.

Apropos this goal, we have 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 the design of the normalization procedure, the meta-model used to represent grammars, and the algorithms used to perform the normalization. A pilot study of the normalization process across three grammars is described in Sec. IV. Sec. V describes the results of this pilot study. Sec. VI describes the limitations of this discussion and the threats to the validity of the pilot study. Finally, Sec. VII concludes this paper with a summary and description of future work.

II. Background

Context-free grammars (CFG) are defined by $G = (V, \Sigma, P, S)$ [4]. 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 [4]. CFGs can be represented using Backus Naur Form (BNF) grammars [5]. Each BNF grammar is composed of a set of productions and a start production. 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. In BNF, there are two basic operators concatenation (' ') and alternation (|) [5].

The concatenation operator is represented by concatenating the operands (e.g. $\langle$A$\rangle$ a). The alternation operator is represented by $\mid$ separating its operands (e.g. $\langle$A$\rangle$ $\mid$ a). Parentheses are used to delimit expressions to reduce ambiguity and improve readability (e.g. $\langle$A$\rangle$ (a | ϵ)). Uppercase letters enclosed within angle brackets denote non-terminal symbols while lowercase symbols in `monospace font` denote terminal symbols.

BNF grammars, though expressive, tend to be cumbersome [6]. Thus, others have added various operators to BNF, forming variants collectively known as Extended Backus Naur Form (EBNF) [7]. EBNF both includes additional operators while redefining existing operators. Specifically, EBNF shows concatenation using the comma (,) and includes the following: the optional operator ([...]), which indicates that surrounded symbols are required 0 or 1 times, the repetition operator ({...}), which indicates that surrounded symbols are required 0 or more times. Finally, parenthesis surrounding a set of symbols denotes a group. Though EBNF is a more concise representation of a grammar than BNF, neither are typically used in practice [6].

In practice, grammars are typically represented by a modified form of EBNF in a language-specific to a parser generator tool such as ANTLR [8], yacc [9], bison [10], JavaCC [1], and many others. Beyond these forms of grammars, there are tree-based grammar description languages such as SDF [11] and TXL [12], as well as methods to describe grammars using languages such as XML [13].

As noted in Sec. I, Island Grammars are one approach to the development of multilingual grammars. Island Grammars extend the definition of a CFG. Thus, an Island Grammar is defined as $\mathcal{G} = (V, \Sigma, P, S, I)$ [14]. An Island Grammar modifies a base grammar using the set I. This set contains productions of interest (also known as islands) used to extract knowledge from source files. All other productions reduce to what is colloquially known as water. This reduction occurs by having a "catch-all" production to match any non-interest production [14].

Normalization of grammars is necessary for efficient processing [15]. To the best of the authors' knowledge there are currently no known normalization procedures to facilitate grammar merging. Though, a well known approach to normalization is the Chomsky Normal Form [16]. In this form, all productions are one of the following:

$$\begin{array}{rcl} \langle \mathrm{A} \rangle & \rightarrow & \langle \mathrm{B} \rangle \langle \mathrm{C} \rangle \\ \langle \mathrm{A} \rangle & \rightarrow & \mathrm{a} \\ \langle \mathrm{S} \rangle & \rightarrow & \epsilon \end{array}$$

That is, each production's rule is either two non-terminal symbols, a terminal symbol, or the empty string (only for the start production). Furthermore, a production may have multiple definitions rather than be written in a more concise form using the alternation operator. Although this form provides a useful simplification [15], it is not appropriate for merging as it does not account for several issues that can lead to ambiguous results when merging.

[1]https://javacc.github.io/javacc/

III. Approach

This section details the design, meta-model, and implementation of a normalization algorithm to simplify the automated merging of grammars. We begin by laying out the foundations of the normalization.

A. Normalization Foundations

The automated merging of grammars requires the grammar to be normalized to an equivalent but constrained form. The primary constraint restricts each of the productions of a grammar to one the following two forms:

Form$_1$ The rule of the production only uses the concatenation operator to concatenate symbol, e.g. $\langle$A$\rangle \rightarrow \langle$B$\rangle$ b . . . .

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$\rangle$ | b | ϵ | . . . .

In addition to the primary constraint, there are three secondary constraints, as follows. (i) Each symbol in a rule cannot have the same form as its containing rule. (ii) Unit rules are not permitted (except for the start symbol producing a single terminal symbol). (iii) No pair of productions may have identical right-hand sides. The constraints both identify and resolve five separate issues. These issues revolve 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 the spurious usage of the empty string. An example of this would be the production $\langle$A$\rangle \rightarrow \langle$B$\rangle$ ϵ b. The empty string in the middle of the expression is unnecessary. 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.

$$\begin{array}{rcl} \langle S_1 \rangle & \rightarrow & \mathrm{a} \, \langle \mathrm{B} \rangle \\ \langle \mathrm{B} \rangle & \rightarrow & \mathrm{b \, c} \\ \langle S_2 \rangle & \rightarrow & \langle \mathrm{A} \rangle \, \mathrm{c} \\ \langle \mathrm{A} \rangle & \rightarrow & \mathrm{a \, b} \end{array}$$

This is mitigated by constraint (i) above. The commutative property of the alternation operator causes the third problem. For example, the following two productions are identical, but are represented differently:

$$\begin{array}{rcl} \langle \mathrm{A} \rangle & \rightarrow & \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \\ \langle \mathrm{A} \rangle & \rightarrow & \mathrm{c} \mid \mathrm{b} \mid \mathrm{a} \end{array}$$

To remove this ambiguity, the domain model uses a set container for the terms, which eliminates the problem by removing the ordering from our representation. The fourth issue is the use of unit productions. A unit production is one in which the right-hand side is a single symbol. Here is an example:

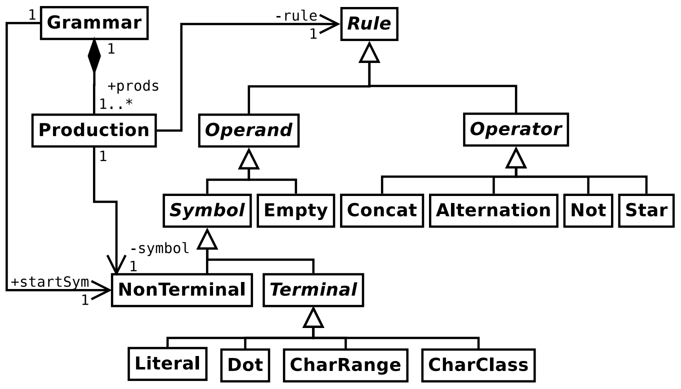

Fig. 1. Class diagram of grammar meta-model.

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

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. The final issue concerns 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, ⟨A⟩'s production can be reduced to a unit rule by replacing the symbols ⟨B⟩ and ⟨C⟩ with a single symbol:

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

This is mitigated by constraint (iii) above.

B. Domain Model

The meta-model, depicted in Fig. 1, used to represent grammars, mirrors the structure of a combination of BNF and EBNF. As depicted, a *Grammar* is composed of one or more productions and a non-terminal start symbol. Each production contains both a non-terminal symbol (the left-hand side) and a rule (the right-hand side).

A rule can be one of two primary types, an *Operand* or an *Operator*. The *Operand* subtree further divides into either the *Empty* string operand or the *Symbol*. Symbols further refine into *NonTerminal*s and *Terminal*s. Terminals further refine into string *Literal*s, *CharRange*s which represent a range of characters, *CharClass* which specifies a character class similar to regular expressions, and finally the dot (.) which can represent any character. The refinements to the Terminal subtree are designed to support the inclusion of ANTLR [8] features.

In addition to the operands, the meta-model includes several operators. Operators supporting BNF are the *Concat* and *Alternation* operators. the alternation and concatenation operators are not binary operators, but rather n-ary operators. The alternation operator aggregates its operands in a set object. Additionally, the meta-model includes the *Star* ($\star$) operator for repetition of 0 or more and the *Not* (~) operator to represent all but a set of characters. These latter two operators are borrowed from ANTLR and help support both the use of ANTLR and EBNF grammars.

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?	optional
'.'	'.'	any character
$\ldots$	$a \ldots z$	a character range
$\backslash p\{\}$	$\backslash p\{Symbol\}$	a character class

ANTLR also includes the one or more repetition (+) and optional (?) operators, as shown in Table I. When an ANTLR grammar is parsed, these operators are substituted with equivalent expressions. 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. After parsing, every application of the $\star$ operator is replaced with a new production. Expressions of the form $\mathcal{A}^*$ are replaced with the production $\langle \text{A} \rangle \rightarrow \mathcal{A}\ \langle \text{A} \rangle \mid \epsilon$.

C. Normalization Algorithm

The approach for normalizing grammars is depicted in Algorithm 1. The Algorithm's primary component is the procedure *Normalize* (lines 1-10). This procedure takes as input, an unnormalized grammar represented as an instance of the meta-model, and produces a normalized grammar equivalent to the input grammar as output. Furthermore, the output grammar also maintains the constraints outlined in Sec. III-A. This procedure repeatedly executes six steps until the grammar stabilizes (the point at which the grammar no longer changes). 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 remainder of this subsection is devoted to describing these steps.

The first step, embodied in function *ElminateUnusedProductions* (lines 11-15), removes all productions not enumerable from the start production via a depth-first traversal of the grammar. The second step, embodied in function *SimplifyProductions* (lines 16-21), simplifies productions by removing unnecessary empty strings (ε) that are operands of the concatenation operator. The third step, embodied in function *MergeEquivProductions* (lines 22-28), replaces productions that have identical rules with a single production. This function replaces symbols by scanning the entire grammar and then replacing each old symbol with the new symbol. The new symbol's name

Algorithm 1 Normalization

```
 1: procedure NORMALIZE($\mathcal{G}$)
 2:     repeat
 3:         $\mathcal{G} \leftarrow$ ELIMINATEUNUSEDPRODUCTIONS($\mathcal{G}$)
 4:         $\mathcal{G} \leftarrow$ SIMPLIFYPRODUCTIONS($\mathcal{G}$)
 5:         $\mathcal{G} \leftarrow$ MERGEEQUIVPRODUCTIONS($\mathcal{G}$)
 6:         $\mathcal{G} \leftarrow$ ELIMINATEUNITPRODUCTIONS($\mathcal{G}$)
 7:         $\mathcal{G} \leftarrow$ EXPANDPRODUCTIONS($\mathcal{G}$)
 8:         $\mathcal{G} \leftarrow$ COLLAPSEPRODUCTIONS($\mathcal{G}$)
 9:     until UNCHANGED($\mathcal{G}$)
10:     return $\mathcal{G}$
11: function ELIMINATEUNUSEDPRODUCTIONS($G$)
12:     $W \leftarrow G.V \cap$ DEPTHFIRSTSEARCHFROM($G.S$)
13:     $Q \leftarrow \{(w, G.P(w)) \mid w \in W\}$
14:     $H \leftarrow (W, G.\Sigma, Q, G.S)$
15:     return $H$
16: function SIMPLIFYPRODUCTIONS($G$)
17:     for all $\wr \in$ OPERATORNODES($G$) do
18:         if ISCONCATOPERATOR($\wr$) then
19:             for all $\{p \in$ OPERANDS($\wr$) $\mid p = \epsilon\}$ do
20:                 REMOVEOPERAND($p$)
21:     return $G$
22: function MERGEEQUIVPRODUCTIONS($G$)
23:     for all $\{p_1, p_2\} \in$ UNORDEREDPAIRS($G.P$) do
24:         if RULE($p_1$) = RULE($p_2$) then
25:             $\rho \leftarrow$ COMBINESYMBOLS($p_1, p_2$)
26:             $G$.REPLACEUSES($p_1, \rho$)
27:             $G$.REPLACEUSES($p_2, \rho$)
28:     return $G$
29: function ELIMINATEUNITPRODUCTIONS($G$)
30:     for all $p \in \mathcal{G}.V \setminus \{G.S\}$ do
31:         if ISSYMBOL(RULE($p$)) then
32:             REPLACEUSES($p$, RULE($p$))
33:     if ISNONTERMINALSYMBOL(RULE($G.S$)) then
34:         REPLACEUSES($G.S$, RULE($G.S$))
35:     return $G$
36: function EXPANDPRODUCTIONS($G$)
37:     for all $p \in G.P$ do
38:         for all $\wr \in$ NONROOTOPNODES(RULE($p$)) do
39:             $G$.REPLACEWITHNEWRULE($\wr$)
40:     return $G$
41: function COLLAPSEPRODUCTIONS($G$)
42:     for all $(p_1, p_2) \in$ ORDEREDPAIRS($G.P$) do
43:         $\wr_1 \leftarrow$ ROOTOPERATOR($p_1$)
44:         $\phi_2 \leftarrow$ SYMBOL($p_2$)
45:         if $\phi_2 \in$ CHILDREN($\wr_1$) then
46:             $\wr_2 \leftarrow$ ROOTOPERATOR($p_2$)
47:             if ASSOCIATIVE($\wr_1, \wr_2$) then
48:                 $\wr_1$.REPLACECHILD($\phi_2$, CHILDREN($\wr_2$))
49:     return $G$
```

combines the names of the old productions. The fourth step, embodied in function *EliminateUnitProductions* (lines 29-35), removes all unit productions (excluding the start production). Unit productions are first identified, and then symbols on the left-hand side of each production are replaced with their right-hand side symbols. The fifth step, embodied in function *ExpandProductions* (lines 36-40), converts each production to either *Form*$_1$ or *Form*$_2$. Each non-root operator node of the expression tree of the rule is pulled into a distinct production. The final step, embodied in *CollapseProductions* (lines 41-49), combines associative operators. In the case of BNF grammars, only the concatenation and alternation operators are associative.

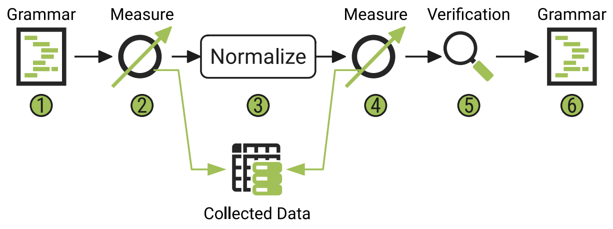

Fig. 2. Pilot study process.

IV. Pilot Study

To evaluate the above approach, we performed a small pilot study on three grammars selected from the ANTLR [8] grammar repository[2]. The grammar selection criteria were as follows: (i) selected grammars should be of varying sizes with at least one grammar small enough to be easily inspected, and (ii) grammars should be of different 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 eight commands [17]. We chose this language because its grammar is minimal and easily inspected. For similar reasons, we chose the more complicated XML grammar. XML is commonly used for sending information between applications [18] and for specifying configuration files [19]. Finally, Java™ was selected for the high likelihood it would need to be included as a base language [20]–[22]. This language is significantly more complicated than either Brainfuck or XML, as it is a widely used [23] general-purpose programming language. This selection of languages meets our original criteria.

The pilot study process, as depicted in Fig. 2, follows the enumerated flow. 1.) Each grammar is read in and processed to form an instance of the meta-model depicted in Fig 1. 2.) The system measures the number of productions [24] of the instance and records this value. 3.) Once measured, the instance is normalized using the process specified in Algorithm 1. 4.) The Algorithm's output grammar is measured, and the number of productions recorded. 5.) Once the normalization process completes, the system verifies that each production is either of *Form*$_1$ or *Form*$_2$ and that it does not include unexpected rules. 6.) Finally, the normalized grammar is output.

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

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

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

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

Fig. 4. Brainfuck grammar after normalization. Symbols ⟨A⟩ and ⟨B⟩ are symbols with auto-generated names.

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, partial results of transforming Java™'s grammar are shown.

Fig. 3 and Fig. 4 show Brainfuck's grammar before and after normalization. As depicted, the normalized grammar meets all the conditions required for the normal form. The normalization procedure recognized that the allowed syntax inside square brackets is the same as that of the entire file. Also, note that the normalization replaced both $*$ expressions. 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. Rather, Table II shows the net increase in the number of productions. This change occurs because productions in the original grammar are split apart into simpler productions, and each use of the $*$ ANTLR feature introduced two rules.

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

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:

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

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 that 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:

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

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 arose because of the constraint that a production cannot reference another production that has the same form. This replacement results in the following:

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

Other substitutions resulted in a small number of productions in the normalized grammar being extremely long. Intriguingly, these large 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 that 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.

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 the number of productions for Brainfuck's grammar and an increase for the XML and Java™ grammars, as shown in Table II. The increase in the last two grammars was not unreasonable. This increase is attributable to the conversion of the following two specific ANTLR features into equivalent BNF features: (i) the optional qualifier ? and (ii) the zero or more qualifier $*$.

VI. Threats to Validity

In this work, we focused on threats to the conclusion, construct, internal, and external validity, as detailed by Wohlin et al. [25]. There is a severe threat to the conclusion validity of this study, as this was a study conducted on only three grammars. Additionally, there are two threats to the internal validity of the study. First, the selected grammars are not necessarily representative of the population of grammars. Second, there is no notion of causal influence that can be inferred by the study. There is also a threat to the construct validity of this study, in that we examined only a single method (given that multiple methods for this particular problem do not exist). Finally, threats to external validity are present due to the lack of support in the process for grammars used in practice.

VII. Conclusions and Future Work

In this paper, we developed an automatic approach to normalizing grammars. This approach reduces the effort required to merge grammars automatically. 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 multilingual Island Grammar-based parsers, thereby easing the construction of multilingual 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 [12] and SDF [11] grammars. Additionally, this process is currently being integrated as part of an overarching approach to automate the development of multilingual parsers through the automated construction of Island Grammars based on existing grammars. Finally, we intend to present the theoretical underpinnings of the presented algorithm in the context of context-free 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] 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.

[3] 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.

[4] 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.

[5] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference," *Proceedings of the International Comference on Information Processing, 1959*, 1959.

[6] V. Zaytsev, "Bnf was here: What have we done about the unnecessary diversity of notation for syntactic definitions," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1910–1915. [Online]. Available: https://doi.org/10.1145/2245276.2232090

[7] E. B. ISO, "Iso/iec 14977: 1996 (e)," *ISO: Geneva*, 1996.

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

[9] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[10] R. P. Corbett, "Static semantics and compiler error recovery," Ph.D. dissertation, California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 1985.

[11] 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.

[12] 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.

[13] R. Lämmel and V. Zaytsev, "An introduction to grammar convergence," in *Integrated Formal Methods*, M. Leuschel and H. Wehrheim, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 246–260.

[14] A. V. Deursen and T. Kuipers, "Building documentation generators," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug. 1999, pp. 40–49.

[15] M. Sipser, *Introduction to the Theory of Computation*. Cengage learning, 2012.

[16] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137 – 167, 1959. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995859903626

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

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

[19] C. Jacquemot, L. Latil, and V. Abrossimov, "Preparation of a software configuration using an XML type programming language," US Patent US20 040 003 388A1, Jan., 2004.

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

[21] 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.

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

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

[24] 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.

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.