SIGMA - Normalization

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 challenges for code analysis tools. One such difficulty is building parsers that allow these tools to understand multilingual systems [1]. Current multilingual-capable tools use a separate parser or grammar for each supported language, or they use a limited intermediate representation (IR). Tools using multiple parsers are time-consuming to maintain [2], while those using an IR are usually constrained to an ecosystem such as .NET or the JVM [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. To reduce the effort of creating these island grammars, we've developed a method for automatically merging grammars. This method demands input grammars be normalized. This paper explains the requirements of the normalization and presents an algorithm to perform it.

The organization of this paper is as follows. Sec. II discusses the theoretical foundations related to this work. Sec. III details the properties of normalization and the meta-model and algorithms used during the normalization. Sec. IV selects three grammars to demonstrate the normalization process. Sec. V shows the results of normalizing these grammars. Finally, sec. VIII concludes this paper with a summary and description of future work.

II. Background and Related Work

Context-free grammars are defined by $G = (V, \Sigma, P, S)$. V is the set of non-terminal symbols, Σ is the set of terminal symbols, $P \subseteq V \times (V \cup \Sigma)^*$ is the set of productions, and $S \in V$ is the start symbol [5]. In examples of parts of grammars mentioned in this paper, upper case letters enclosed within angle brackets denote non-terminal symbols while lowercase symbols in `monospace font` denote terminal symbols. A subset of Extended Backus-Naur Form (EBNF) is used to represent productions [6]. Each production is written as $\Phi \rightarrow R$ where Φ is a non-terminal symbol and R is an expression/rule. Each expression can be either a symbol, the empty string (ϵ), or expressions combined with an operator. Operators include concatenation ($\langle$A$\rangle$ `a`), alternatives ($\langle$A$\rangle$ | `a`), and repetition ($\langle$A$\rangle^*$). In this paper, we restrict ourselves to the concatenation and alternative operators. We also use parenthesis to delimit expressions when doing so prevents ambiguity (e.g. $\langle$A$\rangle$ (`a` | ϵ)).

III. Approach

A. Design

To design the normalization, we needed a normalization process that simplifies merging grammars. This led us to our first design goal: that it is easy to compare productions of normalized grammars. To do this, we constrained the productions of normalized grammars to each be one of two forms:

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

Form$_2$ The rule of the production only uses the alternatives (|) operator. E.g. $\langle$A$\rangle \rightarrow \langle$B$\rangle$ | *b* | ....

However, restricting each production to one of these two forms can still introduce problems. Productions defining a portion of a language can be written in several different ways. This can introduce undesired ambiguity. To reduce this ambiguity, we identified and removed five sources of ambiguity. The first is ambiguity introduced by spurious usage of the empty string. The second is ambiguity introduced by the associative properties of operators. The third is ambiguity introduced by the commutative property of the alternatives (|) operator. The fourth is ambiguity introduced by unit productions. The fifth source is ambiguity introduced by duplicate productions.

The first ambiguity just mentioned is spurious usage of the empty string. An example of this would be the rule $\langle$A$\rangle \rightarrow \langle$B$\rangle$ ϵ `b`. The empty string in the middle of the expression is unnecessary. These unnecessary empty strings are eliminated in Algorithm 3.

The second source of ambiguity arises from the associative property of the concatenation and alternative 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 remove this source of ambiguity, we can further qualify *Form*$_1$ and *Form*$_2$. Namely, we add the constraint that a production cannot use non-terminal symbols that are the same form. Algorithm 7 of the normalization process details how this is accomplished.

The commutative property of the alternative operator causes the third source of ambiguity. 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 source of ambiguity 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 simple and is performed in Algorithm 5.

The fifth and last source of ambiguity is ambiguity introduced by duplicate productions. Here is an example production that can be simplified significantly by removing duplicate productions:

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

Detection and removal of duplicate productions is detailed in algorithm 4 of the normalization process.

Aggregating all chosen design choices, the properties of the normalized grammar are as follows. First, each production is one of the two following forms:

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

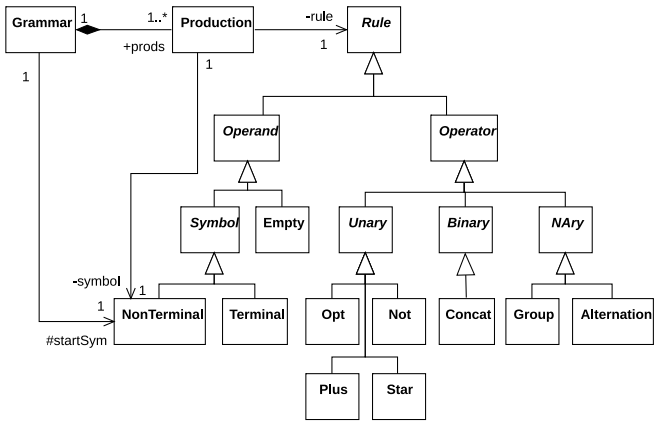

Fig. 1. Class diagram of grammar meta-model.

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

Second, each symbol in a rule cannot have the same form as that rule. Third, unit rules are not permitted except for the case of the start symbol producing a single terminal symbol. Fourth, no pair of productions can have identical right-hand sides.

B. Domain Object Model

The meta-model representing our grammars mirrors the structure of BNF grammars. However, we also implemented it so that it's easily extensible to other grammar formats. We did this by representing the right-hand side of each production with an expression tree of operands and operators. The operands and operators are open to extension. For example, we also defined operators and operands specific to EBNF and ANTLR. Note that the empty string is a unique operand in our meta-model. Namely, it the only operand that is not a symbol. A class diagram of the meta-model is in fig. 1.

Worthy of mention is how the alternation ($\mid$) operator stores its operands. The alternation operator aggregates its operands in a set object. This removes the ordering information per the second ambiguity identified in our design. Finally, the alternation and concatenation operators are not binary operators, but rather n-ary operators.

C. Normalization Algorithm

The following algorithm 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 1, 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.

Algorithm 1 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 2 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
```

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 2.

The second process simplifies productions by removing unnecessary empty strings (ε). These are those that are operands of the concatenation operator. This process is incarnate in Algorithm 3.

The third process removes 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.

Algorithm 3 Simplify Productions

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

Algorithm 5 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
```

To do this, it first identifies unit productions. It then replaces symbols on the left of each production with their right-hand symbols. Algorithm 5 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 6.

The sixth and final step of the normalization process combines associative operators. For BNF grammars, only the concatenation and alternation operators are associative. Algorithm 7 details this step.

IV. Experimental Design

A. Pilot Study

To evaluate the above approach, we performed a small pilot study on three grammars. We selected these grammars from the ANTLR grammar repository. To select these three grammars, we looked for three grammars of varying sizes and

Algorithm 6 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
```

Algorithm 7 Collapse Productions

```
function CollapseProductions(G)
    for all (p1, p2) ∈ OrderedPairs(G.P) do
        ι1 ← RootOperator(p1)
        φ2 ← Symbol(p2)
        if φ2 ∈ Children(ι1) then
            ι2 ← RootOperator(p2)
            if Associative(ι1, ι2) then
                ι1.ReplaceChild(φ2, Children(ι2))
            end if
        end if
    end for
    return G
end function
```

applications. The three grammars chosen were the Brainfuck, XML, Java grammars.

Brainfuck is an esoteric language notable for its extreme simplicity [7]. It is Turing-complete despite only having 8 commands. We chose this language because its grammar is extremely small which allows it to be easily inspected. XML was also chosen because of its grammar's small size. However, it is still significantly more complex than Brainfuck. XML is commonly used for sending information between applications [8] and configuration files [9]. Java is a general purpose programming language used all over the world [10]. Its grammar is significantly more complicated than either of the two previously mentioned grammars. We chose to include this grammar because applications with a need for multilingual parsing would likely include Java as one of their languages [11]–[13].

To evaluate each grammar, we normalize each grammar. Before and after normalization, we measure the number of productions. 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.

One difficulty with the grammars we chose is that they are written in ANTLR and not BNF. To resolve this, we introduced additional operators and terminal symbols for constructs not easily representable in BNF. The constructs allowed in ANTLR but not in BNF are +—one or more repetitions, $\star$—zero or more repetitions, ~—not one of a set of characters, ?—zero or one repetition, `.`—any character, character ranges (e.g. a...z), and character classes (e.g. word characters).

We augmented our meta-model with special terminal symbols representing `.`, character ranges, and character classes. We also added operators for $\star$ and ~.

The + and ? operators were substituted with equivalent expressions while being parsed. Expressions of the form $\mathcal{A}+$ were replaced with $\mathcal{A}\ \mathcal{A}^*$ while expressions of the form $\mathcal{A}?$ were replaced with $(\epsilon|\mathcal{A})$.

Finally, after parsing, every application of the $\star$ operator was replaced with a new production. Given an expression of the form $\mathcal{A}^*$, it was replaced with a rule with the following production

$$\langle \mathrm{A} \rangle \quad \rightarrow \quad \mathcal{A}\ \langle \mathrm{A} \rangle \ \mid \ \epsilon$$

V. Results

VI. Interpretation

A. Evaluation of Results

B. Limits of the Study

C. Inferences

D. Lessons Learned

VII. Threats to Validity

VIII. Conclusions and Future Work

A. Summary of Findings

B. Relation to Existing Evidence

C. Impact

D. Limitations

Acknowledgements

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] 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.

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

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

[9] C. Jacquemot, L. Latil, and V. Abrossimov, "Preparation of a software configuration using an XML type programming language," Patent, Aug., 2007.

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

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

[12] 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.

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