SIGMA - Normalization

Isaac Griffith and Rosetta Roberts
Empirical Software Engineering Laboratory
Informatics and Computer Science
Idaho State University
Pocatello, Idaho, 83208
Email: {grifisaa@isu.edu, roberose@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, while those using an IR are usually constrained to an ecosystem such as .NET or the JVM [2], [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. Section II discusses the theoretical foundations related to this work. Section III details the properties of normalization and the meta-model and algorithms used during the normalization. Section V selects three grammars to demonstrate the normalization process. Section VI shows the results of normalizing these grammars. Finally, Section IX 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 this paper, 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. Each expression can be either a symbol, the empty string (ϵ), or expressions combined with an operator. Operators include concatenation ($\langle \text{A} \rangle$ a), alternatives ($\langle \text{A} \rangle$ | a), and repetition ($\langle \text{A} \rangle^*$). 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. $\langle \text{A} \rangle$ (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:

1. $\langle \textit{Form}_1 \rangle \rightarrow \langle \text{A} \rangle \; a \; \ldots$, where each term is a terminal symbol or a non-terminal symbol with an F_2 production and there are at least two terms in the rule.
2. $\langle \textit{Form}_2 \rangle \rightarrow \langle \text{A} \rangle \; | \; a \; | \; \ldots$, where each term is a terminal symbol, the empty string, or a non-terminal symbol with an F_1 production and there are at least two terms in the rule except for the special case when there is only production.

To design the normalization, we decided that it should have the following properties. Our domain object model represents each rule as a tree of operators and operands. Searching for similar rules and productions is simplified if each rule is composed of a single operator and its operands because this allows set and list based comparisons, which are simpler, to be performed rather than tree based comparisons. Because of this, we require that the normal form has at most single operator for each rule of the normalized grammar. 2. The size increase induced by normalization is minimal. Larger grammars are more difficult to process. 3. The normalized grammar is unambiguous given a grammar. I.e. there is exactly one normalized grammar for each grammar. 4. One anticipated difficult for searching similar portions of grammars is that refactoring that individual developer introduce that don't change the meaning of the grammar will result in portions of grammars being less similar. To mitigate this, we desire that certain transformations on the input grammar before normalization do not change the normalization result. The transformations we chose are the following 1. Refactoring common terms of rules into a separate rule. 2. Duplicating a rule. 3. Introducing an unused non-terminal symbol and its production. 4. Replacing a non-terminal symbol with a non-terminal symbol that produces that non-terminal symbol. 4. Replacing all usages of a non-terminal symbol with its rule.

To meet these requirements, we decided that the normal form would have each production as one of the following forms.

1. $\langle \mathrm{Form}_1 \rangle \rightarrow \langle \mathrm{A} \rangle \; a \; \ldots$, where each term is a terminal symbol or a non-terminal symbol with an F_2 production and there are at least two terms in the rule.
2. $\langle \mathrm{Form}_2 \rangle \rightarrow \langle \mathrm{A} \rangle \mid a \mid \ldots$, where each term is a terminal symbol, the empty string, or a non-terminal symbol with an F_1 production and there are at least two terms in the rule except for the special case when there is only production.

The reason we chose these two forms is that because rules of these forms are relatively easy to compare. This allows rules to easily be compared and merged. The restriction that non-terminal symbols referenced in each rule must have productions of the opposite form is so that examples

To meet requirement III-A, our normalization process is performed using only transformation that are the inverse of transformations we do not want to affect our normalization process. To ensure that the size increase is minimal, we do not attempt to reverse transformations that rely on the distributive property between the concatenation and union | operators. To reverse transformations of this type, it is required to distribute productions. For example,

$$\begin{aligned} \langle \mathrm{A} \rangle &\rightarrow \mathtt{a} \, \langle \mathrm{B} \rangle \\ \langle \mathrm{B} \rangle &\rightarrow \mathtt{b} \mid \mathtt{c} \end{aligned}$$

would have to be transformed to

$$\langle \mathrm{A} \rangle \rightarrow \mathtt{a\,b} \mid \mathtt{a\,c}.$$

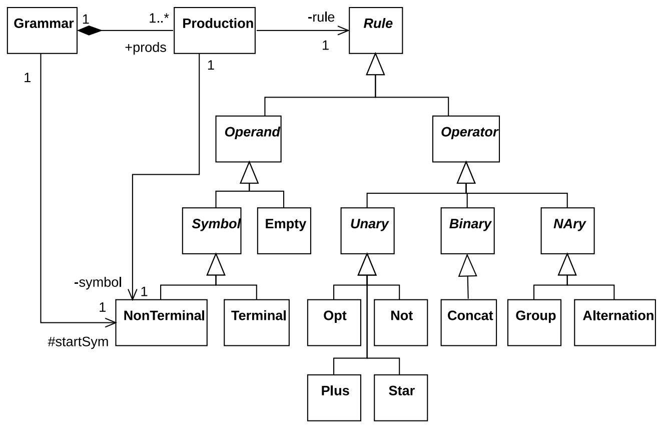
Performing transformations of this kind repeatedly would result in an unreasonable increase in the number of productions. In addition, it would be unable to handle the case when a production indirectly references itself.

B. Domain Object Model

- Use sets for children of union operators.
- Union and concatenation operators are n-ary operators.
- Epsilon/empty string

Algorithm 1 Normalization Algorithm

```
 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: end procedure
```



e

IV. Normalization Algorithm

The following algorithm defines the approach for normalizing a given grammar. The normalization process defined here facilitates the ability to merge productions, in pursuit of the overarching goal of automated generation of Island [X], Tolerant [X], Bridge [X], and Bounded Seas [X] grammars.

This algorithm assumes that the source grammar, G, was initially in some defined formalism such as ANTLR [X], EBNF [X], BNF [X], SDF [X], TXL [X], etc. The grammar was then read in and processed to conform to the metamodel depicted in Figure **??**. Assuming that the grammar meets this condition, the goal of this algorithm is then to reformat the grammar such that each production is of one of Form_1 or Form_2

The normalization process, as defined in Algorithm 1, repeatedly executes six processes 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.

A. Eliminating Unused Rules

This process removes all productions that are not produced, directly or indirectly, from the start production. This is accomplished by enumerating all symbols producuable from the start symbol via a depth first search (see Algorithm 3) and then creating a new grammar using only the enumerated symbols, as shown in Algorithm 2.

Algorithm 2 Eliminate Unused Productions

```
1: function EliminateUnusedProductions($\mathcal{G}$)
2:     $H \leftarrow (V, E)$
   ▷ Create empty graph
3:     for all $v \in \mathcal{G}.V$ do
4:         $\mathcal{H}.V \leftarrow \mathcal{H}.V \cup \{v\}$
5:         AddRuleToGraph($\mathcal{G}, \mathcal{H}, \mathcal{G}.P(v)$)
6:         $\mathcal{H}.E \leftarrow \mathcal{H}.E \cup \{(v, \mathcal{G}.P(v))\}$
7:     end for
8:     DFSMark($\mathcal{G}.S$)
9:     $\mathcal{G}.V \leftarrow \{\, v \in \mathcal{G}.V \mid \text{MARKED}(v) \,\}$
10:    $\mathcal{G}.P \leftarrow \{\, (v, \mathcal{G}.P(v)) \mid v \in \mathcal{G}.V \,\}$
11: end function
12: function AddRuleToGraph($\mathcal{G}, \mathcal{H}, r$)
13:    $\mathcal{H}.V \leftarrow \mathcal{H}.V \cup \{r\}$
14:    if IsOperator($r$) then
15:        for all $c \in \text{OPERANDS}(r)$ do
16:            AddRuleToGraph($\mathcal{G}, \mathcal{H}, c$)
17:            $\mathcal{H}.E \leftarrow \mathcal{H}.E \cup \{(r, c)\}$
18:        end for
19:    end if
20: end function
```

Algorithm 3 Depth First Marking

```
1: function DFSMark($start$)
2:     $\mathcal{S} \leftarrow [start]$
3:     while $\mathcal{S} \neq \varnothing$ do
4:         $p \leftarrow \text{POP}(\mathcal{S})$
5:         Mark($p$)
6:         for all $s \in$SUCC($p$) do
7:             if !IsMarked($s$) then
8:                 PUSH($\mathcal{S}, s$)
9:             end if
10:        end for
11:    end while
12: end function
```

B. Simplifying Productions

This process aims to simplify productions. This is achieved by removing unnecessary ε's concatenated with other rules and replacing operators with only one operand with their operator. This process is embodied in Algorithm 4.

C. Merging Equivalent Productions

Productions that have identical rules are replaced by a single production. This new production is given a name derived from the productions that were merged to create it. The algorithm for this is shown in Alg. 5.

D. Eliminating Unit Productions

All non-terminals with productions of one of the following two forms will have their non-terminal symbols replaced by their rules, and their productions eliminated.

Algorithm 4 Simplify Productions

```
1: function SimplifyProductions($\mathcal{G}$)
2:     for all $v \in \mathcal{G}.V$ do
3:         $\mathcal{G}.P(v) \leftarrow$ SimplifyRule($\mathcal{G}.P(v)$)
4:     end for
5: end function
6: function SimplifyRule($r$)
   ▷ Replace empty terminal string with $\epsilon$
7:     if IsTerminal($r$) $\wedge$ IsEmpty($r$) then
8:         return $\epsilon$
9:     end if
10:    if IsOperator($r$) then
11:        let $C$ be Children($r$)
12:        $C \leftarrow \{$ SimplifyRule($c$) $\mid c \in C \}$
13:        if IsConcatenate($r$) then
14:            $C \leftarrow \{\, c \in C \mid c \neq \epsilon \,\}$
15:            if $|C| = 0$ then
16:                return $\epsilon$
17:            end if
18:        end if
19:        if IsConcatenate($r$) $\vee$ IsUnion($r$) then
   ▷ Replace operators with single operand with operand
20:            if $|C| = 1$ then
21:                let $\{c\}$ be $C$
22:                return $c$
23:            else
24:                return $r$
25:            end if
26:        end if
27:    end if
28: end function
```

Algorithm 5 Merge Equivalent Productions

```
1: function MergeEquivProductions($\mathcal{G}$)
2:     $pairs \leftarrow \varnothing$
3:     for $i \in [0, |\mathcal{G}.\Sigma|)$ do
4:         for $j \in (i, |\mathcal{G}.\Sigma|)$ do
5:             if $i \neq j$ then
6:                 $pairs \leftarrow pairs \cup (\mathcal{G}.\Sigma[i], \mathcal{G}.\Sigma[j])$
7:             end if
8:         end for
9:     end for
10:    for all $p \in pairs$ do
11:        if $p.left.rule = p.right.rule$ then
12:            CombineAndReplace($p.left, p.right$)
13:        end if
14:    end for
15: end function
```

$$\begin{aligned} \langle \mathsf{a} \rangle &\rightarrow \langle \mathsf{b} \rangle \\ \langle \mathsf{a} \rangle &\rightarrow \mathtt{a} \end{aligned}$$

Elimination of productions of the first form, is derived from

Algorithm 6 Eliminate Unit Productions

```
1: function ELIMINATEUNITPRODUCTIONS($\mathcal{G}$)
2:     for all $p \in \mathcal{G}.\Sigma$ do
3:         if $|p.rule| = 1$ then
4:             REPLACE($uses(p), p.rule$)
5:         end if
6:     end for
7: end function
```

Algorithm 7 Expand Productions

```
1: function EXPANDPRODUCTIONS($\mathcal{G}$)
2:     repeat
3:         $changed \leftarrow \bot$
4:         for all $p \in \mathcal{G}.\Sigma$ do
5:             if ISCONCAT($p.rule$) then
6:                 for all $g \in p.rule$ do
7:                     if ISGROUP($g$) then
8:                         CREATEANDREPLACEWITH-PROD($g$)
9:                         $changed \leftarrow \top$
10:                    end if
11:                end for
12:            else if ISALT($p.rule$) then
13:                for all $a \in p.rule$ do
14:                    CREATEANDREPLACEWITHPROD($a$)
15:                    $changed \leftarrow \top$
16:                end for
17:            end if
18:        end for
19:    until $changed = \bot$
20: end function
```

Chomsky Normal Form (CNF) [X]. Eliminations of productions of the second form, a derivation from CNF, allows the simplification process to simplify rules of the following form:

$$\begin{aligned}\langle \text{a} \rangle &\rightarrow \langle \text{b} \rangle \ \texttt{a b} \\ \langle \text{b} \rangle &\rightarrow \epsilon\end{aligned}$$

E. Expanding Productions

Productions that have nested rules have all nested content replaced by with a non-terminal. The new non-terminal defines a production pointing to their content.

F. Collapsing Compatible Productions

The final step of the normalization process combines productions that are associative with each other. This ensures that any non-terminal symbols referenced by a rule will not define a duplicate production. The following provides an example:

Algorithm 8 Collapse Productions

```
1: function COLLAPSEPRODUCTIONS($\mathcal{G}$)
   ▷ Split productions into form1 and form2
2:     $f_1 \leftarrow$ COLLECT("form1")
3:     $f_2 \leftarrow$ COLLECT("form2")
4:     for all $p \in f_1$ do
5:         if ONLYTERMINALS($p.rule$) then
6:             REPLACEF1USESWITHRULE($p$)
7:         end if
8:     end for
9:     for all $p \in f_2$ do
10:        if ONLYTERMINALS($p.rule$) then
11:            REPLACEF2USEWITHRULE($p$)
12:        end if
13:    end for
14: end function
```

Language
Java™8
Brainfuck
XML

TABLE I
LANGUAGES USED IN PILOT STUDY.

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

would then collapse to form:

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

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

Brainfuck is an esoteric language notable for its extreme simplicity [CITE]. 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 [CITE something about SOAP] and configuration files [CITE]. Java™ is a general purpose programming language used all over the world [CITE]. 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 [CITE something about JSP, something about embedded languages in strings, something about mixed language systems (e.g. migration to SCALA or Kotlin)].

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.

VI. Results

VII. Interpretation

A. Evaluation of Results

B. Limits of the Study

C. Inferences

D. Lessons Learned

VIII. Threats to Validity

IX. 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.