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

Multilingual parsing is an open problem that is currently being worked on. One method of multilingual parsing that has been developed is by creating island grammars for the combined grammars [1]. An automated method for doing this is currently being developed [SIGMA REFERENCE HERE]. One challenge to automating the creation of island grammar based multilingual parsers is detecting similar parts of grammars and combining them. This process becomes easier with an appropriate normalization process and normal form.

In this project, we propose to design and verify a procedure for appropriate normalization.

RG Design a normal form and normalization process for grammars that is conducive to identifying and merging similar parts across grammars to be used for merging grammars for the automated creation of multilingual parserss.

Organization

The remainder of this paper is organized as follows. Section II discusses the theoretical foundations of this work while also discussing other related studies. Section V details the design of experiments which evaluate the normalization steps of SIGMA. Section VI details the threats to the validity of this study. Finally, this paper is concluded in Section VII.

II. Background and Related Work

A. Theoretical Foundations

«««< Updated upstream Context-free grammars are defined as $G = (V, \Sigma, P, S)$, where V is the set of non-terminal symbols, Σ is the set of terminal symbols, P is the set of productions, and S is the start production [2]. In this paper, we use a modified syntax for productions $\Phi \rightarrow R$. Φ is any non-terminal symbol and R is a rule. Rules can be either a symbol, ϵ (the empty string), rules concatenated together, or rules unioned together with the $|$ operator. Rather than using multiple productions when a symbol can produce multiple rules, the rules are combined with the $|$ operator. For example productions normally written as

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

are written as the following instead

$$\langle\text{A}\rangle \rightarrow \texttt{a} \mid \texttt{b}.$$

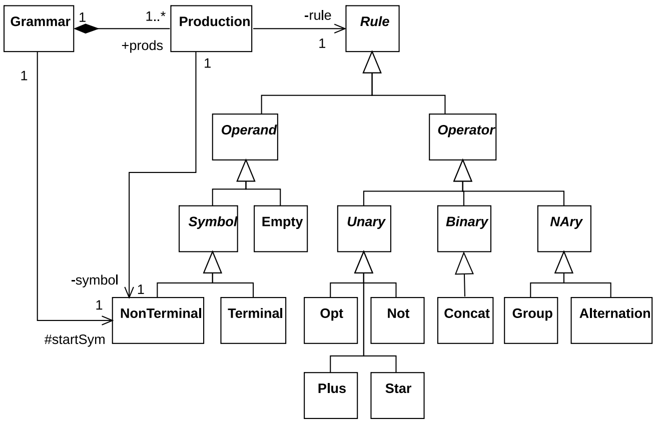
======= Context-free grammars are defined as $G = (V, \Sigma, P, S)$, where V is the set of non-terminal symbols, Σ is the set of terminal symbols, P is the set of productions, and S is the start production [2]. In this paper, a subset of Extended Backus-Naur Form (EBNF) to represent productions [**?**]. In each production $\Phi \rightarrow R$, Φ is a non-terminal symbol and R is an expression representing what the non-terminal symbol can be expanded into. An expression can be either a non-terminal symbol, a terminal symbol or string, the empty string (ϵ), expressions concatenated together, expressions separated with the $|$ operator, or an expression surrounded by parenthesis. The $|$ operator represents the union of sentences that can be expressed by each expression. The concatenation of expressions represents the sentences that can be formed by concatenating the sentences that each expression can form. »»»> Stashed changes

Various normal forms of grammars have been proposed by people. These normal forms have mainly been introduced to make parsing and manipulating grammars easier. The most used normal form is the Chomsky Normal Form (CNF) [3]. Grammars are transformed into CNF and other normal forms through simple transformations which preserve the language of the grammar. Normal forms for finding and merging similar parts across grammars have not been designed.

III. Approach

A. Domain Object Model

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



B. *Design*

To design our normal form, we decided that it should have the following properties.

1. 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.
   5. 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. $\langle\text{Form}_1\rangle \rightarrow \langle\text{A}\rangle\ a\ \ldots$, 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. $\langle\text{Form}_2\rangle \rightarrow \langle\text{A}\rangle \ \mid\ a\ \mid\ \ldots$, 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.

e

Algorithm 1 Normalization Algorithm

```
procedure NORMALIZE($\mathcal{G}$)
    repeat
        $\mathcal{G} \leftarrow$ ELIMINATEUNUSEDPRODUCTIONS($\mathcal{G}$)
        $\mathcal{G} \leftarrow$ SIMPLIFYPRODUCTIONS($\mathcal{G}$)
        $\mathcal{G} \leftarrow$ MERGEEQUIVPRODUCTIONS($\mathcal{G}$)
        $\mathcal{G} \leftarrow$ ELIMINATEUNITPRODUCTIONS($\mathcal{G}$)
        $\mathcal{G} \leftarrow$ EXPANDPRODUCTIONS($\mathcal{G}$)
        $\mathcal{G} \leftarrow$ COLLAPSEPRODUCTIONS($\mathcal{G}$)
    until UNCHANGED($\mathcal{G}$)
end procedure
```

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 4, 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,

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

would have to be transformed to

$$\langle\text{A}\rangle \rightarrow \texttt{a b} \mid \texttt{a c}.$$

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.

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

Algorithm 2 Eliminate Unused Productions

```
function EliminateUnusedProductions($\mathcal{G}$)
    $H \leftarrow (V, E)$
    ▷ Create empty graph
    for all $v \in \mathcal{G}.V$ do
        $\mathcal{H}.V \leftarrow \mathcal{H}.V \cup \{v\}$
        AddRuleToGraph($\mathcal{G}, \mathcal{H}, \mathcal{G}.P(v)$)
        $\mathcal{H}.E \leftarrow \mathcal{H}.E \cup \{(v, \mathcal{G}.P(v))\}$
    end for
    DFSMark($\mathcal{G}.S$)
    $\mathcal{G}.V \leftarrow \{\, v \in \mathcal{G}.V \mid \text{MARKED}(v) \,\}$
    $\mathcal{G}.P \leftarrow \{\, (v, \mathcal{G}.P(v)) \mid v \in \mathcal{G}.V \,\}$
end function
function AddRuleToGraph($\mathcal{G}, \mathcal{H}, r$)
    $\mathcal{H}.V \leftarrow \mathcal{H}.V \cup \{r\}$
    if IsOperator($r$) then
        for all $c \in$ operands($r$) do
            AddRuleToGraph($\mathcal{G}, \mathcal{H}, c$)
            $\mathcal{H}.E \leftarrow \mathcal{H}.E \cup \{(r, c)\}$
        end for
    end if
end function
```

Algorithm 3 Depth First Marking

```
function DFSMark($start$)
    $\mathcal{S} \leftarrow [start]$
    while $\mathcal{S} \neq \varnothing$ do
        $p \leftarrow$ pop($\mathcal{S}$)
        Mark($p$)
        for all $s \in$succ($p$) do
            if !isMarked($s$) then
                push($\mathcal{S}, s$)
            end if
        end for
    end while
end function
```

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.

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.

Algorithm 4 Simplify Productions

```
function SimplifyProductions($\mathcal{G}$)
    for all $v \in \mathcal{G}.V$ do
        $\mathcal{G}.P(v) \leftarrow$ SimplifyRule($\mathcal{G}.P(v)$)
    end for
end function
function SimplifyRule($r$)
    ▷ Replace empty terminal string with $\epsilon$
    if IsTerminal($r$) $\wedge$ IsEmpty($r$) then
        return $\epsilon$
    end if
    if IsOperator($r$) then
        let $C$ be Children($r$)
        $C \leftarrow \{$ SimplifyRule($c$) $\mid c \in C \,\}$
        if IsConcatenate($r$) then
            $C \leftarrow \{\, c \in C \mid c \neq \epsilon \,\}$
            if $|C| = 0$ then
                return $\epsilon$
            end if
        end if
        if IsConcatenate($r$) $\vee$ IsUnion($r$) then
    ▷ Replace operators with single operand with operand
            if $|C| = 1$ then
                let $\{c\}$ be $C$
                return $c$
            else
                return $r$
            end if
        end if
    end if
end function
```

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.

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

Elimination of productions of the first form, is derived from 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:

Algorithm 5 Merge Equivalent Productions

```
function MergeEquivProductions($\mathcal{G}$)
    $pairs \leftarrow \varnothing$
    for $i \in [0, |\mathcal{G}.\Sigma|)$ do
        for $j \in (i, |\mathcal{G}.\Sigma|)$ do
            if $i \neq j$ then
                $pairs \leftarrow pairs \cup (\mathcal{G}.\Sigma[i], \mathcal{G}.\Sigma[j])$
            end if
        end for
    end for
    for all $p \in pairs$ do
        if $p.left.rule = p.right.rule$ then
            CombineAndReplace($p.left, p.right$)
        end if
    end for
end function
```

Algorithm 6 Eliminate Unit Productions

```
function EliminateUnitProductions($\mathcal{G}$)
    for all $p \in \mathcal{G}.\Sigma$ do
        if $|p.rule| = 1$ then
            replace($uses(p), p.rule$)
        end if
    end for
end function
```

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

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:

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

would then collapse to form:

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

Algorithm 7 Expand Productions

```
function ExpandProductions($\mathcal{G}$)
    repeat
        $changed \leftarrow \bot$
        for all $p \in \mathcal{G}.\Sigma$ do
            if isConcat($p.rule$) then
                for all $g \in p.rule$ do
                    if isGroup($g$) then
                        createAndReplaceWithProd($g$)
                        $changed \leftarrow \top$
                    end if
                end for
            else if isAlt($p.rule$) then
                for all $a \in p.rule$ do
                    createAndReplaceWithProd($a$)
                    $changed \leftarrow \top$
                end for
            end if
        end for
    until $changed = \bot$
end function
```

Algorithm 8 Collapse Productions

```
function CollapseProductions($\mathcal{G}$)
▷ Split productions into form1 and form2
    $f_1 \leftarrow$ Collect("form1")
    $f_2 \leftarrow$ Collect("form2")
    for all $p \in f_1$ do
        if OnlyTerminals($p.rule$) then
            ReplaceF1UsesWithRule($p$)
        end if
    end for
    for all $p \in f_2$ do
        if OnlyTerminals($p.rule$) then
            ReplaceF2UseWithRule($p$)
        end if
    end for
end function
```

V. Experimental Design

A. Pilot Study

«««< Updated upstream 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 that we chose were the brainfuck, Java™, and XML grammars (Table **??**). We chose the brainfuck grammar because the small size of it allows it to be easily manually inspected. Java™'s grammar was chosen because it has a relatively complex grammar and the Java™programming language is used in corporate environments. XML's grammar was chosen because

Language
Java™7
Brainfuck
XML

TABLE I
Languages used in pilot study.

it is relatively simple, but not as simple as brainfuck's, while also having significant uses.

======= 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. »»»> Stashed changes

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. Threats to Validity

VII. Conclusions and Future Work

The timeline to complete this study is as follows:

We intend to publish these results at one of the following conferences:

Acknowledgements

References

[1] 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.

[2] 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.

[3] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137–167, Jun. 1959.