UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

A Syntactic Approach to Macro-Grammars for Context-Free Languages

AUTOR: Jaime Nuche Bascón h000276

TUTOR: Pablo Nogueira Iglesias

Julio 2009

Contents

1 Introduction **1**

1.1 Contributions 2

1.2 Structure of the work 2

2 More Than Parsing **5**

2.1 Introduction 5

2.2 MTP Motivation 5

2.3 Object-Oriented Normal Form (ONF) 6

2.4 Extended ONF (EONF) 7

Theorem 2.4.1: BNF-EONF equivalence 7

2.5 Generalized ONF (GONF) 8

3 Macro Enhancement for Context-Free Grammars **11**

3.1 Introduction 11

3.2 State of the Art 11

3.3 Definitions 13

Definition 3.3.1: Macro 13

Definition 3.3.2: Macro-Set 14

Definition 3.3.3: Set of Γ-terms 14

Definition 3.3.4: Substring 15

Definition 3.3.5: Subterm 15

Definition 3.3.6: Instance 15

Definition 3.3.7: Macrogrammar 15

Example 3.3.1: Example macrogrammars 16

Definition 3.3.8: Derivation relation 16

Example 3.3.2: Example derivations 17

Definition 3.3.9: OI and IO derivations 17

Example 3.3.3: Example nested macrogrammar 18

3.4 Issues present in Thiemann's and Neubauer's work 19

4 A sufficient condition for context independence 23

4.1 Introduction 23
4.2 Definitions & Theorems 23
Definition 4.2.1: Labeled directed graph 23
Definition 4.2.2: Call graph 23
Example 4.2.1: Example call graphs 24
Definition 4.2.3: Instance derivation 25
Example 4.2.2: Example instance derivation 26
Definition 4.2.4: Instance generated by a macrogrammar 26
Lemma 4.2.1: OI-Instance derivation equivalence 26
Definition 4.2.5: Length function 28
Definition 4.2.6: Unlimited macro 29
Theorem 4.2.1: Finiteness of the generated instance set 29
Theorem 4.2.2: Context-free macrogrammar 30
Example 4.2.3: Macrogrammar specialization 32
Corollary 4.2.1: Sufficient condition for context independence 33
4.3 Uniform grammar condition 33
Definition 4.3.1: Uniform macro 33
Definition 4.3.2: Uniform macrogrammar 33
Lemma 4.3.1: Uniform macrogrammars contain no unlimited macros 34
Example 4.3.1: Non-Uniform macrogrammar 35

5 Strict Syntactic Restriction 37

5.1 Introduction 37
5.2 Definitions & Theorems 37
Definition 5.2.1: Restricted production 37
Example 5.2.1: Restricted & non-restricted productions 37
Definition 5.2.2: Strictly restricted macrogrammar 38
Theorem 5.2.1: Context-free strict macrogrammar 38
5.3 Real-world patterns 40
5.3.1 Patterns 40
Example 5.3.1: Operator precedence 40
Example 5.3.2: Similar structures in different contexts 42
Example 5.3.3: Permutation phrases 44

5.3.2 Conclusion 46

6 Lenient Syntactic Restriction **47**

6.1 Introduction 47

6.2 Definitions & Theorems 47

Definition 6.2.1: Presence graph 47

Example 6.2.1: Example presence graphs 48

Definition 6.2.2: Presence derivation 50

Example 6.2.2: Example presence derivations 50

Lemma 6.2.1: Presence derivation supersedes instance derivation . . . 51

Definition 6.2.3: Cycle 53

Example 6.2.3: Example cycles 53

Definition 6.2.4: Restricted cycle 54

Definition 6.2.5: Leniently restricted macrogrammar 54

Lemma 6.2.2: Parameter limit for derivations of a given length 55

Theorem 6.2.1: Context-free lenient macrogrammar 56

6.3 Real-world patterns 58

6.3.1 Patterns 59

Example 6.3.1: Operator precedence 59

Example 6.3.2: Similar structures in different contexts 59

Example 6.3.3: Permutation Phrases 61

6.3.2 Conclusion 63

6.4 Comparative with Thiemann's and Neubauer's work 63

7 Conclusions and future work **67**

7.1 Conclusions 67

7.2 Future lines of work 68

Bibliography **69**

Chapter 1
Introduction

We commence this thesis by setting a backdrop for the work.

In [HN05] Herranz and Nogueira introduced the MTP (More Than Parsing) tool. They designed MTP to be an automatic parser generator. Their main contribution consisted in a syntax formalism which avoids the burden of annotated grammars, an inconvenience present in most modern automatic parser generators, while at the same time supplying the parser with quality data structures.

The syntax formalism they introduced, GONF(Generalized Object Normal Form), is similar to the well-known BNF formalism for describing context-free grammars; the grammars used at present to build parsers. GONF allows for the use of parameterized non-terminals in the description of grammars. However, it was necessary to prove that this extension did not cause the formalism to generate grammars not in the context-free class.

GONF's parameterized non-terminals are simply macros, like those in regular programming languages. Grammars with macros have been studied in [Fis68, TN04, TN08]. It was proved in these works that macro-grammars can actually generate context-sensitive languages. They also introduce some attempts at limiting macro-grammars to generate only context-free languages, though results are not completely satisfactory and present some issues.

Based on these works, I present a new formal framework for macro-grammars. I also provide a new characterization for macro-grammars and two different practical restrictions which ensure a given macro-grammar remains within the context-free class boundaries. We can apply these restrictions to GONF or any other notation for macro-grammars.

1.1 Contributions

- A theoretical frame for macro-grammars based on previous research made in [Fis68, TN04, TN08].
- A sufficient condition which ensures a macro-grammar generates a context-free language.
- Two syntactic restrictions which ensure the previous condition is fulfilled. A syntactic condition entails benefits over a semantic one:
  - Less computing-intensive in general.
  - It is constructive. The designer applies the condition when describing the grammar. Grammars are written correctly from start.
  - Mistakes can be reported in place to the designer. Easy to integrate into an editor.
- Raw algorithm (mathematical induction) to transform a macro-grammar into a BNF grammar.
- Corroboration of previous contributions made in [Pér08].

1.2 Structure of the work

In chapter 2 we discuss MTP and GONF in further detail. We provide some foundation on the assumptions regarding the formalisms involved. Finally, we see how GONF calls for a macro-grammar approach.

Chapter 3 contains all the necessary background and formal definitions to work with macro-grammars. It also discusses some of the issues in [TN08].

The sufficient condition for a macro-grammar to generate a context-free language is given and proved correct in chapter 4. The chapter also contains a proof that Pérez's condition from [Pér08] is a particular case.

In chapter 5 a strict syntactic restriction is given. We show in this chapter that a macro-grammar in strict restricted form generates a context-free language. Three

examples from Java and JavaScript grammars are expressed using macros in strict restricted form.

In chapter 6 a lenient syntactic restriction is given. We show that a macro-grammar in lenient restricted form generates a context-free language. We express the Java and JavaScript examples using leniently restricted macros. Finally, we compare these results with the strict restriction and with Thiemann's and Neubauer's results in [TN08].

Finally, in chapter 7 we summarize our results and outline some possible future applications worth investigating.

Chapter 2
More Than Parsing

2.1 Introduction

In this chapter we consider the motivations and grammar-related theoretical aspects of the MTP (More Than Parsing) tool introduced by Herranz and Nogueira in [HN05]. MTP's goal is to become an automatic generator of parsers. We will make precise the original motivation for the rest of this thesis and analyze the formalisms in [HN05].

In this chapter we assume the reader is familiarized with BNF notation for grammars [ASU86]. A grammar is a tuple (Σ, N, P, S) with Σ the set of terminals, N the set of non-terminals, P the set of productions (or rules) and S the axiom. We write λ for the empty string.

2.2 MTP Motivation

Compilers are an essential tool to transform code from a high-level language into machine code. In a compiler, the syntax of the language to be compiled is described using a context-free grammar. Writing a compiler from scratch is a complex and difficult task. The development of tools to automatically generate portions of it seems natural.

Abstract syntax trees (AST) are a widespread mechanism used by such tools. Abstract syntax allows for a structural description of the underlying grammar. ASTs represent a connection point, a transition point where it is possible to establish the separation between independent modules in the compiler. Unfortunately, most modern tools require the designer to label the grammar with semantic actions to build the AST. For example, ANTLR [PQ94] builds ASTs automatically, but it requires that grammar productions be annotated to indicate the desired AST constructions. In general, the quality of the AST is conditioned by the underlying syntax formal-

ism used by the tool. The limitations of such formalism in expressing the AST data structure are overcome by the labeling.

Herranz's and Nogueira's aim was to provide a syntax formalism powerful enough to render labeling unnecessary, that is, to provide quality types for the AST. The MTP tool is supported on a new syntax formalism GONF [HN05] which allows for a simultaneous description of both the language syntax and the AST data structure without the necessity of production annotation. However they do not provide an analysis on the expressiveness of GONF. It could be that GONF grammars generate a language class wider than the context-free languages, which is not a desirable property.

It is now that we can put this thesis in the right frame. The original objective of this work was to determine the expressiveness of GONF. That is, to determine whether GONF grammars are contained within the context-free class of languages. Let us start by peering into the origins of GONF.

2.3 Object-Oriented Normal Form (ONF)

Based on object-oriented design, ONF [Kos91] is a syntax formalism that restricts BNF productions to either classifications:

$$A \rightarrow A_1 \mid A_2 \mid \ \dots \ \mid A_n$$

with $n > 1$, A_i and A non-terminals; or structures:

$$B \rightarrow x_1 \ x_2 \ \dots \ x_m$$

with $m > 0$, x_i either terminal or non-terminal and B a non-terminal with only one such production.

ONF narrows the gap between syntax description and AST data structures. It is easy to assimilate classifications to *is-a* relationships (inheritance) and structures to *has-a* relationships (composition). The grammar symbols provide names for the types/classes of the AST.

ONF is just a restriction over BNF notation. Therefore, under the derivation relation

defined for BNF, the set of languages generated by ONF cannot be larger than the context-free set.

2.4 Extended ONF (EONF)

Much like BNF, ONF can be extended with regular expressions on the right-hand sides of productions. This extension equips grammars with the ability to directly represent iteratives (lists) and optionals. In other words, regular expressions account for container types in the AST.

EONF adds three regular constructs. For an $\alpha \in (\Sigma \cup N)^*$:

- $(\alpha)^*$. That is, a list of zero or more α's. The same language is generated by $A \rightarrow \lambda \mid \alpha\ A$.
- $(\alpha)^+$. That is, a list of one or more α's. The same language is generated by $A \rightarrow \alpha \mid \alpha\ A$.
- $(\alpha)^?$. That is, an optional. The same language is generated by $A \rightarrow \lambda \mid \alpha$.

Regular expressions generate regular languages and regular languages are context-free languages. It intuitively follows regular constructs do not allow EONF to generate more languages than ONF (or BNF, for that matter). At any rate, let us show that BNF and EONF are equally expressive, that is, they can generate exactly the same languages.

THEOREM 2.4.1 A BNF grammar can always be translated into an EONF grammar generating the same language, and vice versa.

PROOF: **BNF can always be translated into EONF**

The only BNF productions not allowed in EONF are optionals (classifications) not formed solely of non-terminals. It should suffice to wrap up such productions to translate them into EONF. Therefore, a BNF-like production such as:

$$A \rightarrow \alpha_1 \mid \alpha_2\ \ldots \mid \alpha_n \text{ with } \alpha_i \in (N \cup \Sigma)^*$$

is wrapped into the EONF set:

$$A \rightarrow A_1 \mid A_2 \ \ldots \mid A_n$$
$$A_1 \rightarrow \alpha_1$$
$$A_2 \rightarrow \alpha_2$$
$$\vdots$$
$$A_n \rightarrow \alpha_n$$

It immediately follows that if $\alpha_i \overset{*}{\Rightarrow} w$, then $A_i \overset{*}{\Rightarrow} w$. We make sure that we are not using the new symbols anywhere else in the grammar, so that they derive only these strings.

EONF can always be translated into BNF

Regular expressions are not part of the BNF syntax, but any other EONF construction is valid BNF. All we have to do to obtain a pure BNF grammar is to replace all appearances of regular expressions with their equivalent definition in BNF. For this, a new non-terminal A is added to the set of non-terminals for each string α appearing on the right-hand side of a rule. Then all appearances of such α are replaced by its corresponding A and productions for that A are added according to the definitions for the regular expressions above.

It immediately follows by the way we have defined the regular constructs, that both grammars generate the same language.

□

2.5 Generalized ONF (GONF)

ONF had the advantage of forcing the naming of types, but EONF allows expressing nameless containers in the form of regular expressions. In principle, this setback can be worked around if the AST target language has nameless composite types. Yet, even in that case, nameless containers can get out of hand by means of nesting constructs such as $F \rightarrow (x\ (y\ z)^*)^+$.

To solve this, GONF restricts regular expressions to have only one element with

information, that is, only one type should be generated for the contents of a given regular construct. This is achieved by simply forcing the naming of the regular expressions. For example, the above example of nested construct can be translated into:

$$F \rightarrow T^{+}$$
$$T \rightarrow x\ R^{*}$$
$$R \rightarrow y\ z$$

Naming containers contents gives raise naturally to the idea of supporting designer-defined containers in the form of parameterized non-terminals. For example:

$$list(x, t) \rightarrow x\ (t\ x)^{*}$$
$$argList \rightarrow list(arg, ',')$$
$$stmtList \rightarrow list(stmt, ';')$$

GONF includes such generalized constructs where names are made explicit, thus allowing automatic type generation for the AST.

Herranz and Nogueira do not state the derivation relation they use for parameterized non-terminals. But, by their examples and assumptions, it is easy to infer they are using a kind of unrestricted derivation. We will explain this in more detail in the next chapter.

So, as we can see, GONF permits the presence of structures in the grammar which are basically macros. Fortunately, grammars with macros have been already studied in previous works [Fis68, TN04, TN08]. In the next chapter we will study macro-grammars in detail and see how they generate new sets of languages more powerful than context-free languages. Such discovery presents us with the necessity of finding some restriction for macro-grammars (hence for GONF) to ensure we stay within the context-free frame.

Chapter 3
Macro Enhancement for Context-Free Grammars

3.1 Introduction

In this chapter we give the necessary background about macro-grammars as well as present the formal (and intuitive) definitions we will need. As we saw in the previous chapter, GONF extends EONF to contain macros, thus our theoretical results on macro-grammars will be appliable to GONF.

The last section constitutes a little survey on some issues present in the last paper published by Thiemann and Neubauer [TN08].

3.2 State of the Art

Context-free grammars have been the most successful and widespread formalism used to describe programming languages. They allow to parse a program and decide its syntactical correctness in cubic time in the worst case and in linear time for certain restricted subsets, such as LL and LR grammars [ASU86].

However, context-free grammars are not able to abstract common patterns present in programming languages. That forces the parser developer to tediously repeat patterns throughout the structure of her grammar description. This limitation also keeps the grammar from being modular and reusable.

For instance, consider the following piece of BNF grammar:

$$\begin{aligned} \mathit{VarList} \rightarrow\ & \mathit{Var} \\ \mid\ & \mathit{VarList}\ \mathit{Var} \\ \mathit{SchemeList} \rightarrow\ & \mathit{Scheme} \\ \mid\ & \mathit{SchemeList}\ \mathit{Scheme} \end{aligned}$$

We have a list of variables and a list of schemes. As we can see their structure is exactly the same, they both correspond to the abstract pattern of a list. If we could capture the list pattern then we could use it by means of specialization with the concrete elements of the list. The pattern could then be part of a module. We could import the pattern and use it without having to reproduce it every time.

As of today, the only supported mechanism for pattern capture is regular expressions on the right-hand side of productions [Cha84, WM95]. Unfortunately regular forms can only express the general pattern of a list. Thereby we are interested in finding a more powerful mechanism.

Fischer first studied macro-grammars [Fis68], aiming at providing grammar description tools with more powerful features. Macro-grammars extend normal context-free grammars by adding macro symbols very much in the fashion of programming macros. This allows for a pattern to be captured in the definition rule of a macro symbol, including regular expressions. He showed macro-grammars produce two new, not comparable language classes depending on the order on which nested macros are rewritten. These two classes are: IO (inside-outside expansion) and OI (outside-inside expansion).

Fischer showed too that these two classes are properly contained in the class of context-sensitive languages and that, trivially, properly contain the class of context-free languages. Consequently, despite that macro-grammars retain many of the decidability properties of context-free grammars, we do not have an efficient algorithm to decide such properties, most notably, the word problem.

Finally, Fischer proved OI macro-grammars were the same language class as Aho's indexed grammars class. Aho, in turn, showed it is recursively unsolvable to determine whether an indexed grammar generates a context-free language [Aho68]. Therefore there is no Turing decidable, sufficient and necessary condition for when an OI macro-grammar is in fact a context-free grammar. It is, however, possible to find decidable sufficient conditions.

Thiemann and Neubauer have recently tried to tackle the issue of keeping macros' extra expressiveness while remaining within the context-free class of languages. They have focused their developments on the OI class. They first provided a production-wise syntactical restriction which ensures an OI macro-grammar is limited to a

context-free language [TN04], but this condition proves to be too limiting. Then they devised a specialization method to map an OI macro-grammar into a context-free grammar and a static analysis capable of deciding whether such specialization terminates [TN08]. Unfortunately, this last paper is not as rigorous as it is desirable. Some key concepts are not defined and proofs are more of sketches than actual proofs.

3.3 Definitions

From this point on, we assume that for any given set of symbols T, the free monoid T^* is defined for T. That is, for any given set, there is a binary concatenation operation defined associative and the empty word λ is its neutral element.

Definition 3.3.1 (Macro) We can consider macros as extended non-terminals. Unlike normal non-terminals, a macro can be fed with parameters, the number of which we say it is the macro's arity and it is fixed for a given macro. Parameters are enclosed in parentheses and separated by commas. A macro takes always strings as parameters and yields the string on the right-hand side of any production defining the macro. As we can see, macros are basically functions whose type is: $f : string^{arity(f)} \rightarrow string$.

Normal non-terminals are nullary macros, that is, macros with arity 0 which take no parameters. In that case, we can drop the parentheses and write them in the usual way.

Macro parameters can be referred to on the right-hand side of any production defining the macro symbol. These references are simply copies of the parameter strings a grammar is fed with. A macro parameter is nameless, we use natural numbers to refer them. The number $i - 1$ stands for the ith argument. 0 stands for the first argument from left to right, 1 for the second, etc. We say these are formal parameters.

Example of a macro with arity 2 and a defining rule for it with formal parameters:

$$MacroSymbol(string1,\ string2)$$
$$MacroSymbol \rightarrow a\ 0\ \ bb\ 1\ \ NonTermSymbol\ b\ 1$$

To actually use a macro, we can invoke/instantiate it on the right-hand side of a production of any macro. We say the parameters a macro is instantiated with are

actual parameters. For example:

$$\textit{MacroSymbol2} \rightarrow c \ \textit{MacroSymbol}(b, 0)$$

Note that we can pass a *MacroSymbol2* formal parameter as *MacroSymbol* actual parameter. □

DEFINITION 3.3.2 (MACRO-SET) Thiemann and Neubauer define this same concept in their last paper [TN08], calling it a “signature”. However this name is already in use for a similar, yet slightly different, construction in standard theories such as universal algebra and term rewriting systems [Grä79, BN98]. Hence the decision to change its name.

A macro-set is a pair $\Gamma = (N, \mathsf{a})$ where N is a finite set of macros and $\mathsf{a} : N \rightarrow \mathbb{N}$ is an arity function from N to the set of natural numbers. For a given element in N, a returns its arity. □

DEFINITION 3.3.3 (SET OF Γ-TERMS) Let Γ be a macro-set. Let X, be a set of symbols disjoint from N. The set $T_\Gamma(X)$ of Γ-terms with symbols X is defined inductively by:

$$\frac{x \in X}{x \in T_\Gamma(X)} \qquad \frac{A \in N \qquad t_i \in T_\Gamma(X) \qquad 1 \leq i \leq \mathsf{a}(A)}{A(t_1, \ldots, t_{\mathsf{a}(A)}) \in T_\Gamma(X)}$$

$$\frac{A \in T_\Gamma(X) \qquad B \in T_\Gamma(X)}{A.B \in T_\Gamma(X)}$$

The set of Γ-terms will be used to define different sets of terms by instantiating parameter X and extending the set . Since in the definition of Γ-terms we have included a concatenation operator, we define Γ-terms a premonoid so that we can use the free monoid concatenation operator every time we employ Γ-terms to define a set of terms.

We write AB instead of $A.B$ and drop the concatenation operator. The same convention is applied to any free monoid.

We write $\bar{t}$ for $t_1, \ldots, t_{\mathsf{a}(A)}$ and continue to use t_i for the ith element. We use

subscripts within the range.

We also write T_i as a shorthand for $T_\Gamma(\Sigma \cup \{0, 1 \ldots i-1\})$. □

Definition 3.3.4 (Substring) Given a set of terms T. A term $t_1 \in T$ is a substring of another term $t_2 \in T$ if there exist $\alpha, \beta \in T^*$ such that $t_2 = \alpha t_1 \beta$. □

Definition 3.3.5 (Subterm) Given a set of terms T. A term $t_1 \in T$ is a subterm of another term $t_2 \in T$ if:

- t_1 is a substring of t_2.
- $t_3(\overline{v})$ is a subterm of t_2 and t_1 is a substring of some v_i.

That is, a term is a subterm of another if it appears somewhere in it. We note it $t_2[t_1]$.

For instance: b is a subterm of $A(gm, a)db$ and of $FP(f, abd)g$.

□

Definition 3.3.6 (Instance) A term $A(\overline{t})$ is an instance if either $\mathsf{a}(A) = 0$ or $\mathsf{a}(A) > 0$ and $\overline{t} \in T_\Gamma(\Sigma)$. □

Definition 3.3.7 (Macrogrammar) A macrogrammar is a tuple (Σ, Γ, P, S) where:

- Σ the set of terminals.
- $\Gamma = (N, \mathsf{a})$ a macro-set, with N the set of macros and a an arity function defined for all elements in N.
- $P \subseteq \{(A, w) \mid A \in N,\ w \in T_{\mathsf{a}(A)}\}$ is a finite set of macro productions.

  That is, productions' right-hand side terms are made up of concatenated terminals, non-terminals (nullary macros), macros and natural numbers referring formal parameters. We do not allow empty rules, nor feeding λ as a parameter.
- $S \in N$ with $\mathsf{a}(S) = 0$.

□

Example 3.3.1 (Example macrogrammars)

$$
\begin{aligned}
G_{abc}: S &\rightarrow F(a,\ b,\ c)\\
F &\rightarrow 012\\
F &\rightarrow F(a0,\ b1,\ c2)
\end{aligned}
$$

$$
\begin{aligned}
G_{list}: S &\rightarrow L(a)\\
S &\rightarrow L(b)\\
L &\rightarrow 0\\
L &\rightarrow 0\ L(0)
\end{aligned}
$$

□

Definition 3.3.8 (Derivation relation for macrogrammars) The notation $w[i \mapsto t]$ denotes that all occurrences of parameter i in w are replaced with t. A group of substitutions can be grouped together $w[i \mapsto t_i, i+1 \mapsto t_{i+1}, \ldots]$.

A macrogrammar generates strings over Σ^* using the following derivation relation $\Rightarrow$:

- If $A \rightarrow w \in P$ and $t_1, \ldots, t_{\mathsf{a}(A)} \in T_\Gamma(\Sigma)$ then $A(\bar{t}) \Rightarrow w[0 \rightarrow t_1, \ldots, \mathsf{a}(A)-1 \rightarrow t_{\mathsf{a}(A)}]$.

  This means that whenever we replace a macro with its definition, we replace formal parameters with the actual ones.

- If $f \in (N \cup \{.\})$, $t_1, \ldots, t_{\mathsf{a}(f)} \in T_\Gamma(\Sigma)$ and $t_i \Rightarrow t_i'$ then $f(t_1, \ldots, t_i, \ldots, t_{\mathsf{a}(f)}) \Rightarrow f(t_1, \ldots, t_i', \ldots, t_{\mathsf{a}(f)})$.

  That is, if the function we consider is a macro, we can always derive parameters before deriving the macro itself.

  Besides, if the function is the concatenation operator, the result is the usual derivation relation defined for normal context-free grammars. Note that since concatenation is defined associative we can just as well assume it n-ary instead

of binary for the purpose of giving it an arity. That is, we can regard n concatenated symbols as an n-ary concatenation function applied to those symbols.

Before reaching a string in Σ^*, a derivation will generate sentential forms in $T_\Gamma(\Sigma)$. As usual, $\stackrel{*}{\Rightarrow}$, denotes the reflexive transitive closure of the derivation relation.

□

EXAMPLE 3.3.2 (EXAMPLE DERIVATIONS)

For macrogrammar G_{abc} :

$S \Rightarrow F(a, b, c) \Rightarrow abc$

$S \Rightarrow F(a, b, c) \Rightarrow F(aa, bb, cc) \Rightarrow F(aaa, bbb, ccc) \Rightarrow aaabbbccc$

It is easy to see that $L(G_{abc}) = \{a^n b^n c^n | n \geq 1\}$, which is a context-sensitive language.

For macrogrammar G_{list} :

$S \Rightarrow L(a) \Rightarrow a$

$S \Rightarrow L(a) \Rightarrow aL(a) \Rightarrow aaL(a) \Rightarrow aaa$

$S \Rightarrow L(b) \Rightarrow bL(b) \Rightarrow bb$

$L(G_{list}) = \{a^n | n \geq 1\} \cup \{b^n | n \geq 1\}$. This language is regular, therefore context-free.

□

DEFINITION 3.3.9 (OI AND IO DERIVATIONS) As we mentioned before, macrogrammars generate two classes of languages depending on how we replace nested macros. These two classes correspond precisely to the following restrictions on the derivation relation:

- IO derivation. $A(\bar{t}) \Rightarrow_{IO} w[0 \rightarrow t_1, \ldots, \mathsf{a}(A) - 1 \rightarrow t_{\mathsf{a}(A)}]$ only if $t_1, \ldots, t_{\mathsf{a}(A)} \in \Sigma^*$.

  So, we can only derive a macro using its definition if all its instantiation parameters are formed of terminals only. Considering the previous parallelism with functions, this restriction is similar to call-by-value.

- OI derivation. The derivation $\Rightarrow_{OI}$ is the same as $\Rightarrow$ but with the following restriction: we restrict the second derivation rule to $(f = .)$. That is, derivation does not proceed into parameter positions and macros must be derived using its definition before allowing parameters to be derived. In this case the calling method is similar to call-by-name.

At this point, it is possible to wonder what class of languages the unrestricted derivation $\Rightarrow$ falls in. Fischer showed it is the same class of OI macrogrammars [Fis68].

Both derivations have their corresponding reflexive transitive closure, just like the unrestricted derivation.

Note that example grammars G_{abc} and G_{list} have no nested macros. Therefore they generate the same language under OI and IO derivation.

□

EXAMPLE 3.3.3 (EXAMPLE NESTED MACROGRAMMAR)

$S \rightarrow F(G)$

$G \rightarrow a$

$G \rightarrow b$

$F \rightarrow 0F(0)$

Under IO derivation this grammar generates the language $\{a^n | n \geq 1\} \cup \{b^n | n \geq 1\}$. However, we can see under OI derivation it generates the language $\{x^n | x \in \{a, b\} \wedge n \geq 1\}$. We can express these languages using regular expressions as: $aa^* \mid bb^*$ and $(a \mid b)(a \mid b)^*$.

3.4 Issues present in Thiemann's and Neubauer's work

Here we address some of the problems present in [TN08], not all, only an overview, since most are similar in nature. Our special interest in this paper comes from the approach it provides on macros. At the moment of writing these lines, it is the only work I am aware of (other than this very thesis) which has tried to accomplish a characterization (in the sufficient sense) of context-independence in macrogrammars.

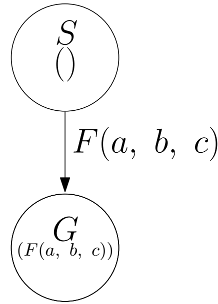
However, the work presents quite a few issues. Some are minor, solvable details, some other affect coherence and formality, hence not allowing for a rigorous evaluation of their results.

As a first example of this, let us consider their usage of the word "subterm" in their instantiation graph definition (Definition 4.4 in the paper). They do not formally define the word in the paper, however they mention Fischer's work and rely on his notions of macrogrammars for their own developments. In Fischer's work a subterm is a substring [Fis68], whereas in usual standard theories, such as term-rewriting systems, the definition of subterm is equivalent to our definition in this thesis [BN98]. Yet if we were to use Fischer's definition of subterm, a fatal flaw would arise.

Consider the following grammar:

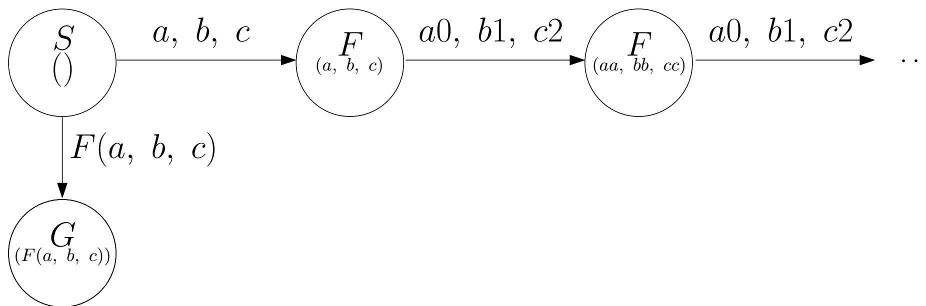
$$\begin{aligned} G_{CntExa} : S &\rightarrow G(F(a,\ b,\ c)) \\ G &\rightarrow 0 \\ F &\rightarrow 012 \\ F &\rightarrow F(a0,\ b0,\ c0) \end{aligned}$$

Assume the definition of subterm they use is the one introduced by Fischer. Then the instantiation graph associated to this grammar is:



The graph is finite, it contains only two nodes and one edge. But the language generated by G_{CntExa} under OI derivation is our usual example $L(G_{CntExa}) = \{a^n b^n c^n | n \geq 1\}$. This is a contradiction with the assertion that a finite instantiation graph implies a context-free language, as stated in their Lemma 4.7.

Now suppose the definition of subterm is the one we utilize in this thesis. In such case, the instantiation graph is:



An infinite graph, as it was originally supposed to be. This leads us to believe it is not Fischer's definition that they wanted to use, but one equivalent to ours. Still, this is not stated anywhere in the paper. Note that none of their examples contains nested macros, therefore it is not possible to infer which notion they wished to use.

Another issue with the instantiation graph is that they do not show that the graph contains exactly the same instances the macrogrammar can derive under OI. As reasonable as it may look, they do not prove it. However it is necessary for the consequent results to be correct. The only reference to the matter is in the proof of Lemma 4.7: "First, we show by induction on the length of the derivation sequence: if $S \Rightarrow_{M,OI} v_1 A(\overline{r}) v_2$, then $(A, \overline{r}) \in V$." This sounds as if they intended to prove such implication, but they never do so; right after the quoted text, they move on to the main body of the proof.

This broken link between concepts they use, shows up again later. We can see it again in their Lemma 5.2 concerning the relation between the instance graph and the closure of the transition graph (Definition 5.1): "There is a path in IG(M) that visits infinitely many different nodes of IG(M) iff there is an infinite number of edges in CTG(M)." They do not supply a proof for this lemma.

Let us consider their definition for the transition graph. The transition edges are defined as: "$E = \{A \xrightarrow{\overline{s}} B | A \to w \in P, B(\overline{s}) \in w\}$". As it happened with their use of "subterm", the expression $B(\overline{s}) \in w$ is ambiguous. It is never made precise what they mean by "$B(s)$ is in w".

This lack of precision is a constant issue throughout the work. Definitions are not sufficiently rigorous and some proofs are actually more like sketches, to varying degrees.

Nevertheless, I believe their results are essentially correct and useful from a theoretical point; unfortunately they do not supply a direct translation from their mathematical definitions to an implementation, just some example glimpses.

□

Chapter 4

A sufficient condition for context independence

4.1 Introduction

In this chapter we present the necessary concepts and proofs to state and prove a sufficient condition for when an OI macrogrammar generates a context-free language.

We introduce too an alternative condition first introduced by Iván Pérez [Pér08]. We will see how his condition is a particular case of the condition presented in this chapter.

4.2 Definitions & Theorems

Definition 4.2.1 (Labeled Directed Graph) A labeled directed graph is a tuple (V, E) where V is the set of vertices and E is the set of edges. E contains tuples of the form (src, trg, lbl) where src is the source vertex of the edge, trg is the target vertex and lbl is the label.

We write $\textcircled{v_1} \xrightarrow{l} \textcircled{v_2}$ to state that $v_1 \in V$, $v_2 \in V$ and $(v_1, v_2, l) \in E$. □

Definition 4.2.2 (Call Graph) The call graph is a simplified representation of the production set, focused on the macros. We will be using it to define a new type of derivation.

Given a macro grammar $MG = (\Sigma, \Gamma, P, S)$ and a new symbol $Z \notin (N \cup \Sigma)$ we build the call graph $CG = (V, E)$ of MG in the following way:

$$Z \in V \qquad \frac{A \in N}{A \in V}$$

$$\frac{v_1, v_2 \in V, v_1 \to \omega \in P \qquad v_2(\bar{t}) \text{ is a substring of } \omega \text{ for some } t_1, \ldots, t_{\mathsf{a}(v_2)} \in T_{\mathsf{a}(v_1)}}{(v_1, v_2, (t_1, \ldots, t_{\mathsf{a}(v_2)})) \in E}$$

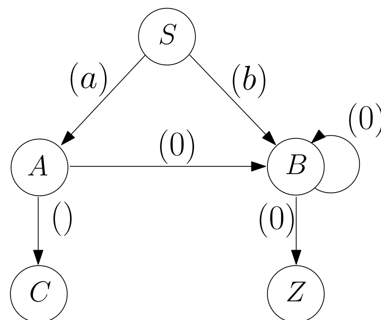
$$\frac{v_1 \in V, v_1 \to \omega \in P \qquad 0 \leq i < \mathsf{a}(v_1) \text{ is a substring of } \omega}{(v_1, Z, (i)) \in E}$$

Note that the graph captures explicit calls to macros on the right-hand sides of productions. However, it is possible to indirectly invoke a macro via a formal parameter. Such special cases are captured in the edges going to Z.

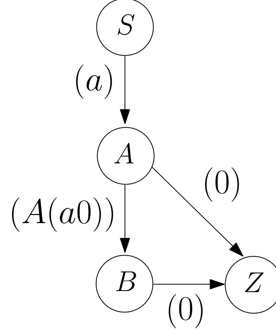
□

Example 4.2.1 (Example call graphs) We show the call graph for the following grammars:

$$\begin{aligned} GrGrammar1 : S &\to A(a) \mid B(b) \\ A &\to B(0)C \\ C &\to c \\ B &\to 0B(0) \mid 0 \end{aligned}$$



$$
\begin{aligned}
GrGrammar2: S &\rightarrow A(a) \\
A &\rightarrow B(A(a0)) \mid 0 \\
B &\rightarrow 0
\end{aligned}
$$



Note that in these two examples macros are unary so as to not clutter up the example graphs. In general that is not the case. Edges' labels are tuples, not single elements as it might seem from the examples.

□

DEFINITION 4.2.3 (INSTANCE DERIVATION) The instance derivation is a relation similar to OI derivation but restricted to instances instead of full sentential forms. We note it $\Rightarrow_I$.

For a given call graph $CG = (V, E)$:

$$
\frac{(A, B, (tb_1, \ldots, tb_{\mathsf{a}(B)})) \in E \qquad A(\overline{ta}) \text{ is an instance} \qquad B \neq Z}{A(\overline{ta}) \Rightarrow_I B(\overline{tb'}) \qquad tb_j^{'} = tb_j[0 \rightarrow ta_1, \ldots, \mathsf{a}(A) - 1 \rightarrow ta_{\mathsf{a}(A)}]}
$$

$$
\frac{(A, Z, (k)) \in E \qquad A(\overline{ta}) \text{ is an instance} \qquad ins \text{ is an instance substring of } ta_k}{A(\overline{ta}) \Rightarrow_I ins}
$$

We can also note $\Rightarrow_{CG,I}$ but the call graph is dropped if understood.

The usual reflexive transitive closure is written $\overset{*}{\Rightarrow}_I$. □

Example 4.2.2 (Example instance derivations)

For macrogrammar $GrGrammar1$:

$S \Rightarrow_I A(a) \Rightarrow_I C$

$S \Rightarrow_I B(b) \Rightarrow_I B(b)$

$S \Rightarrow_I A(a) \Rightarrow_I B(a) \Rightarrow_I B(a)$

For macrogrammar $GrGrammar2$:

$S \Rightarrow_I A(a) \Rightarrow_I B(A(aa))$

$S \Rightarrow_I A(a) \Rightarrow_I B(A(aa)) \Rightarrow_I A(aa)$

$S \Rightarrow_I A(a) \Rightarrow_I B(A(aa)) \Rightarrow_I A(aa) \Rightarrow_I B(A(aaa)) \Rightarrow_I A(aaa)$

□

We are interested in the instances a macrogrammar generates at some point in the derivation process. It will help us determine whether the grammar is in fact context-free.

Definition 4.2.4 (Instance generated by a macrogrammar) Let $MG = (\Sigma, \Gamma, P, S)$ be a macrogrammar and $A(\bar{t})$ an instance. Let rel be a derivation relation. We say MG generates $A(\bar{t})$ under rel if $S \overset{*}{\Rightarrow}_{rel} \alpha A(\bar{t})\beta$ with α, $\beta \in T_\Gamma(\Sigma)^*$.

That is, if $A(\bar{t})$ is a substring of some string generated by MG under rel. □

We show that instance derivation and OI derivation generate exactly the same instances. This will allow us to use instance derivation instead of OI derivation when studying the grammar.

Lemma 4.2.1 Let MG be a macrogrammar and CG its corresponding call graph, then CG under instance derivation generates exactly the same instances as MG does under OI derivation.

More formally, if $\alpha \in T_\Gamma(\Sigma)$ and p is an instance substring of α, then $S \overset{*}{\Rightarrow}_{OI} \alpha$ iff $S \overset{*}{\Rightarrow}_I p$.

PROOF:

We use induction on the number of derivation steps in both $S \overset{*}{\Rightarrow}_{OI} \alpha$ and $S \overset{*}{\Rightarrow}_I p$ to prove the 'only if' and 'if' directions respectively at the same time.

BASIS:

If $S \overset{*}{\Rightarrow}_I S$ then $S \in V$. But $S \in V$ only if $S \in N$. If $S \in N$ then $S \overset{*}{\Rightarrow}_{OI} S$.

Analogously, if $S \overset{*}{\Rightarrow}_{OI} S$ then $S \in N$. But if $S \in N$ then $S \in V$. Therefore $S \overset{*}{\Rightarrow}_I S$.

Thus $S \overset{*}{\Rightarrow}_{OI} S$ iff $S \overset{*}{\Rightarrow}_I S$.

INDUCTIVE STEP:

If: We assume that for a given instance $A(\bar{t})$ such that $S \overset{*}{\Rightarrow}_I A(\bar{t})$, there is a term such that $S \overset{*}{\Rightarrow}_{OI} \alpha A(\bar{t})\beta$ where $\alpha,\ \beta \in T_\Gamma(\Sigma)^*$.

Now, given a certain instance $B(\overline{tb'})$ such that $A(\bar{t}) \Rightarrow_I \ B(\overline{tb'})$, there has to exist an $e \in E$ which either:

- $e = (A, B, (tb_1, \ldots, tb_{\mathsf{a}(B)}))$ and $tb_i' = tb_i[0 \to t_1, \ldots, \mathsf{a}(A) - 1 \to t_{\mathsf{a}(A)}]$
- $e = (A, Z, (j))$ for some $0 \leq j < \mathsf{a}(A)$ and $B(\overline{tb'})$ is a substring of t_j. If $B(\overline{tb'})$ is a substring of t_j then $t_j = \varepsilon\ B(\overline{tb'})\ \zeta$ for some $\varepsilon,\ \zeta \in T_\Gamma(\Sigma)^*$.

But e exists only if there is an $r \in P$ such that either:

- $r = A \to \gamma\ B(\overline{tb})\ \delta$.
- $r = A \to \gamma\ j\ \delta$.

Where $\gamma,\ \delta \in T_\Gamma(\Sigma)^*$ in both cases.

Therefore either $\alpha A(\bar{t})\beta \Rightarrow_{OI} \alpha\gamma B(\overline{tb'})\delta\beta$ or $\alpha A(\bar{t})\beta \Rightarrow_{OI} \alpha\gamma t_j \delta\beta = \alpha\gamma\varepsilon B(\overline{tb'})\zeta\delta\beta$. Hence $S \overset{*}{\Rightarrow}_{OI} \ \ldots B(\overline{tb'}) \ldots$ as we wanted to show.

Only if: Analogously to the 'if' case, we assume that if $A(\bar{t})$ is an instance and $S \overset{*}{\Rightarrow}_{OI} \alpha A(\bar{t})\beta$ with $\alpha,\ \beta \in T_\Gamma(\Sigma)^*$ then $S \overset{*}{\Rightarrow}_I A(\bar{t})$.

Suppose $B(\overline{tb'})$ is an instance such that $\alpha A(\bar{t})\beta \Rightarrow_{OI} \alpha\gamma B(\overline{tb'})\delta\beta$ where $\gamma,\ \delta \in T_\Gamma(\Sigma)^*$. Then there is an $r \in P$ such that either:

- $r = A \rightarrow \gamma\ B(\overline{tb})\ \delta$ and $tb'_i = tb_i[0 \rightarrow t_1, \ldots, \mathsf{a}(A) - 1 \rightarrow t_{\mathsf{a}(A)}]$.
- $r = A \rightarrow \varepsilon j \zeta$ for some $0 \leq j < \mathsf{a}(A)$ and $t_j = \eta B(\overline{tb'})\theta$ such that $\varepsilon\eta = \gamma$ and $\zeta\theta = \delta$.

But if such r exists, then there is an $e \in E$ which either:

- $e = (A, B, (tb_1, \ldots, tb_{\mathsf{a}(B)}))$.
- $e = (A, Z, (j))$.

Therefore $A(\bar{t}) \Rightarrow_I\ B(\overline{tb'})$. Finally $S \overset{*}{\Rightarrow}_I B(\overline{tb'})$ as we wanted to show.

We can conclude that $S \overset{*}{\Rightarrow}_{OI} \alpha$ iff $S \overset{*}{\Rightarrow}_I p$.

□

Definition 4.2.5 (Length Function) Given a set of terms Y. For a term $t \in (T_\Gamma(X) \cup Y)^*$ the function $l : (T_\Gamma(X) \cup Y)^* \rightarrow \mathbb{N}$ yields the number of symbols from N, X and Y a term t is composed of. So if t is made of k symbols, not necessarily different, then $l(t) = k$.

More formally:

- $l(\lambda) = 0$.
- If $t \in \{X \cup Y \cup N\}$ and $t \neq \lambda$ then $l(t) = 1$.
- If $t = rs$ then $l(t) = l(r) + l(s)$.
- If $t = A(\bar{t})$ then $l(t) = 1 + l(t_1) + \ldots + l(t_{\mathsf{a}(A)})$.

For instance: $1(a\ A\ B(a,\ 0,\ d)\ G(2\ b\ B(a))) = 11$ since there are 11 symbols including terminals, macros and formal parameters. □

Definition 4.2.6 (Unlimited macro) An unlimited macro is a macro such that at least one of its parameters has no length limit. That is, for any length, we can use the macro to generate an instance with at least one parameter longer.

Formally, given an $MG = (\Sigma, \Gamma, P, S)$, we say a macro $A \in N$ is unlimited if for any $p \in \mathbb{N}$ there is an instance $A(\bar{t})$ such that $S \overset{*}{\Rightarrow}_I A(\bar{t})$ and $l(t_i) > p$ for some t_i. $\square$

Theorem 4.2.1 The set of instances generated by a macrogrammar MG under OI is infinite iff there is at least one unlimited macro in MG.

Proof:

We say a term $t \in T_\Gamma(\Sigma)$ is used by a macro $A \in N$ if $S \overset{*}{\Rightarrow}_I A(\ldots\ t\ \ldots)$. We say t is used by MG if it is used by some $A \in N$. Note that we are considering only those terms used by MG to generate instances.

The set N is finite and every macro in N has a finite and fixed number of parameters. Thus, if the set of used terms is finite there is a maximum number of possible different instances we can generate. That is, the grammar generates combinations of symbols from finite sets in a finite number of positions, therefore, there is a finite number of combinations, that is, instances.

So if the set of terms used by MG is finite, then the set of instances generated by MG is finite as well.

Reasoning along the same lines, if the set of used terms is infinite, then there has to be some parameter position k for some macro A such that $S \overset{*}{\Rightarrow}_I A(\ldots\ t_k\ \ldots)$ for an infinite number of different terms t_k. But if $t_k^{'} \neq t_k^{''}$ then $A(\ldots\ t_k^{'}\ \ldots) \neq A(\ldots\ t_k^{''}\ \ldots)$, so MG generates an infinite set of instances.

We can claim that MG generates an infinite set of instances if and only if MG uses an infinite set of terms.

If: Let A be an unlimited macro, then for any $p \in \mathbb{N}$ we can find a term t such that $S \overset{*}{\Rightarrow}_I A(\ldots\ t\ \ldots)$ and $l(t) > p$. But $\mathbb{N}$ is infinite, therefore the set of terms used by A is infinite.

If the set of terms used by A is infinite, then the set of terms used by MG is infinite. As we saw above, this means MG generates an infinite set of instances, as we wanted to show.

Only if: Terms in $T_\Gamma(\Sigma)$ are combinations of symbols from finite sets N and Σ, very much like instances. In fact instances are terms in $T_\Gamma(\Sigma)$.

Suppose we can find a $p \in \mathbb{N}$ such that $l(t) \leq p$ for all t in the set of used terms. Then there is a maximum length for such terms and, since N and Σ are finite, the set of used terms is finite.

Now let us suppose there are no unlimited macros in MG. Then for all macro A there is a $p_A \in \mathbb{N}$ for which there is no instance $A(\bar{t})$ fulfilling that $S \overset{*}{\Rightarrow}_I A(\bar{t})$ and $l(t_i) > p_A$ for some t_i. That is, if $S \overset{*}{\Rightarrow}_I A(\bar{t})$ then for all t_i, $l(t_i) \leq p_A$.

There is a finite number of macros in N, so there is a finite number of such $p_A s$. Let us take one of the maximum $p_A s$ and call it p. Then for any instance $F(\bar{t})$ it holds that if $S \overset{*}{\Rightarrow}_I F(\bar{t})$ then $l(t_i) \leq p$ for all t_i. Hence, for all terms t used by MG, $l(t) \leq p$. Therefore, as we saw previously, the set of used terms is finite.

But as we saw at the beginning of the proof, if the set of used terms is finite, then the set of generated instances is finite too, as we wanted to show.

□

Theorem 4.2.2 If the set of instances generated by a macrogrammar MG under OI derivation is finite, then the language generated by MG under OI derivation is context-free.

Proof:

We will prove the theorem by constructing an equivalent BNF grammar. The idea is that we can regard an instance of a macro generated by a macrogrammar as a single non-terminal in the BNF grammar and that we can use that instance to specialize the rules defining the macro to generate a set of equivalent BNF productions. If the set of instances generated is finite, we can achieve this.

Let $MG = (\Sigma, \Gamma, P, S)$ be a macrogrammar which generates a finite set of instances. We define a new grammar $CFG = (\Sigma, N', P', S)$ where $N' \subset T_\Gamma(\Sigma)$. We define N' and P' as follows:

$$S \in N^{'} \qquad \frac{S \stackrel{*}{\Rightarrow}_{MG,OI} t_1 t_2 \ldots t_n \qquad t_j = A(\overline{u}) \text{ is an instance} \qquad A \to \omega \in P}{A(\overline{u}) \to \omega[0 \to u_1, \ldots, \mathsf{a}(A) - 1 \to u_{\mathsf{a}(A)}] \in P^{'}}$$

$$\frac{S \stackrel{*}{\Rightarrow}_{MG,OI} t_1 t_2 \ldots t_n \qquad t_j \text{ is an instance}}{t_j \in N^{'}}$$

Note that an instance added to $N^{'}$ becomes a single symbol, a non-terminal.

The number of different instances t_j MG can derive from S is finite, therefore $P^{'}$ and $N^{'}$ are finite too. Thus CFG is a BNF grammar and the language generated by it is context-free.

Now it remains to see if MG and CFG generate the same language. We use induction to show that $S \stackrel{*}{\Rightarrow}_{MG,OI} \omega$ iff $S \stackrel{*}{\Rightarrow}_{CFG} \omega$, with $\omega \in T_\Gamma(\Sigma)$.

BASIS:

By construction of CFG, if S is the axiom of MG then S is the axiom of CFG and $S \in N^{'}$. This is so only if S is the axiom of MG. Hence $S \stackrel{*}{\Rightarrow}_{MG,OI} S$ iff $S \stackrel{*}{\Rightarrow}_{CFG} S$.

INDUCTIVE STEP:

We assume that $S \stackrel{*}{\Rightarrow}_{MG,OI} \alpha$ iff $S \stackrel{*}{\Rightarrow}_{CFG} \alpha$ with $\alpha = t_1 t_2 \ \ldots \ t_n$, $t_j \in T_\Gamma(\Sigma)$. Suppose some $t_j = A(\overline{u})$ is an instance, then for any derivation $\alpha = t_1 t_2 \ \ldots \ tj \ \ldots \ t_n \Rightarrow_{MG,OI}$ $t_1 t_2 \ \ldots \ t_j^{'} \ \ldots \ t_n$, there has to exist a rule RM in P such that $RM = A \to \omega$ and $t_j^{'} = \omega[0 \to u_1, \ldots, \mathsf{a}(A) - 1 \to u_{\mathsf{a}(A)}]$.

But by construction of CFG, $t_j \in N^{'}$ and there is a rule R in $P^{'}$ of the form $R : A(\overline{u}) \to \omega[0 \to u_1, \ldots, \mathsf{a}(A) - 1 \to u_{\mathsf{a}(A)}]$. Therefore $\alpha \Rightarrow_{CFG} t_1 t_2 \ \ldots \ tj^{'} \ \ldots \ t_n$. Moreover, by construction of CFG, R only exists if RM exists and if RM exists then $\alpha \Rightarrow_{MG,OI} t_1 t_2 \ \ldots \ tj^{'} \ \ldots \ t_n$.

We conclude that $S \stackrel{*}{\Rightarrow}_{MG,OI} \omega$ iff $S \stackrel{*}{\Rightarrow}_{CFG} \omega$. In particular, if $\omega \in \Sigma^*$, ω is a string of the languages generated by MG and CFG. Thus, such languages are the same, as we wanted to show.

$\square$

So if a macrogrammar generates a finite set of instances, we can specialize it. Let us give an example:

Example 4.2.3 (Macrogrammar Specialization) Consider the following macrogrammar:

$$S \to D(a)$$
$$S \to D(b)$$

$$D \to E\ 0\ D(0)$$
$$D \to E\ 0\ E$$

$$E \to a\ E\ b\ E$$
$$E \to b\ E\ a\ E$$
$$E \to aa$$
$$E \to bb$$
$$E \to ab$$
$$E \to ba$$

We have built the macrogrammar to make it look obvious D is only invoked with parameters a and b. So we get instances $D(a)$ and $D(b)$. From them, we specialize two sets of BNF productions:

$$D(a) \to E\ a\ D(a)$$
$$D(a) \to E\ a\ E$$

$$D(b) \to E\ b\ D(b)$$
$$D(b) \to E\ b\ E$$

To make it clear we are now considering a BNF grammar, we can rename the nonterminals $D(a) = F$ and $D(b) = G$. We finally get the following context-free grammar:

$$S \to F \mid G$$

$$F \to E\ a\ F \mid E\ a\ E$$

$$G \to E\ b\ G \mid E\ b\ E$$

$$E \to a\ E\ b\ E \mid\ b\ E\ a\ E \mid\ aa \mid\ bb \mid\ ab \mid\ ba$$

This grammar generates strings with a different number of a's and b's, which is a context-free language. □

We have now all the pieces we need. We put them together and state our sufficient condition in the following corollary.

COROLLARY 4.2.1 (SUFFICIENT CONDITION FOR CONTEXT INDEPENDENCE) If a macrogrammar has no unlimited macros, then it generates a context-free language.

PROOF:

It follows immediately from theorems 4.2.1 and 4.2.2.

□

4.3 Uniform grammar condition

In [Pér08], Iván Pérez introduced the notion of uniform macros and macrogrammars, based on the notion of uniform recursive data types [Bla00].

Pérez uses a different formalization for macrogrammars. We do not intend here to transcribe his entire work, so we will simply use our notation to transcribe the notion of uniform macrogrammar and then show how it is a particular case of corollary 4.2.1. The derivation definition he provides is equivalent to OI, so we can use our instance derivation to derive instances in the same way he does.

DEFINITION 4.3.1 (UNIFORM MACRO) Given a macrogrammar $MG = (\Sigma, \Gamma, P, S)$. For any terms $\bar{t} \in T_\Gamma(\Sigma)$, if a macro $A \in N$ is uniform then $A(\bar{t}) \overset{*}{\Rightarrow}_I A(\overline{u})$ implies $\bar{t} = \overline{u}$.

That is, if we instantiate a macro with certain terms and then derive from it new instances of the same macro, the parameters of the derived instances have to be the same original terms of the first instance. This has to be true for any tuple of terms we decide to initially instantiate the macro with. □

Definition 4.3.2 (Uniform macrogrammar) A macrogrammar is uniform if all its macros are uniform. □

To prove uniform macrogrammars are a particular case of our condition, let us first show all uniform macrogrammars are free of unlimited macros.

Lemma 4.3.1 If a macrogrammar is uniform then it has no unlimited macros.

Proof:

Take a macrogrammar MG. Let $n = |N|$. We define the following set $Instances = \{ins_k \mid S \Rightarrow_I ins_1 \ldots \Rightarrow_I ins_k \ldots \Rightarrow_i ins_m \wedge m \leq n\}$. Note this set is finite, since there are only so many different derivations of length equal or smaller than n.

If MG is uniform then all instances generated by it are in $Instances$, let us see why. Suppose there is an instance $A(\bar{t})$ derivable from S which is not in $Instances$. There has to be a derivation $S \Rightarrow_I A_1(\bar{t}_1) \ldots \Rightarrow_I A_m(\bar{t}_m) = A(\bar{t})$. If $m \leq n$ then $A(\bar{t}) \in Instances$, therefore it has to be that $m > n$.

The set N of macros has n different macros. Thus, by the pigeon principle, if $m > n$ then there is some macro repeated in the derivation. That is, there are $i \neq j$ such that $A_i = A_j$. Assume $i < j$, then $A_i(\bar{t}_i) \overset{*}{\Rightarrow}_I A_j(\bar{t}_j)$. But MG is uniform, so $A_i(\bar{t}_i) = A_j(\bar{t}_j)$.

If there are two equal instances in the derivation, then there has to exist a shorter derivation reaching $A(\bar{t})$, a derivation $S \Rightarrow_I A_1(\bar{t}_1) \ldots \Rightarrow_I A_l(\bar{t}_l) = A(\bar{t})$ with $l = m - (j - i)$.

As long as the length of the new found derivation is greater than n, we can always find yet another shorter derivation reaching $A(\bar{t})$. The length of such derivation will eventually become equal or lower than n, in which case $A(\bar{t}) \in Instances$.

We conclude all instances generated by MG are in $Instances$. $Instances$ is finite, so the number of instances generated by MG is finite. Then, by theorem 4.2.1, MG has no unlimited macros, as we wanted to show. □

We have stated that uniform macrogrammars are a special case of macrogrammars with no unlimited macros. It could be, however, the case that all macrogrammars with no unlimited macros were uniform as well. The following counterexample

disproves such possibility.

EXAMPLE 4.3.1 (NON-UNIFORM MACROGRAMMAR)

$$S \rightarrow F(a,\ b)$$
$$F \rightarrow 01$$
$$F \rightarrow 0\ F(1,0)\ 1$$

There is only one macro F in this macrogrammar. The length of its parameters can never increase since there are no new symbols appended onto them. Hence F is not unlimited. However, by performing a simple derivation, we can see it is not uniform:

$$S \Rightarrow_I F(a,\ b) \Rightarrow_I F(b,\ a)$$

In case you are wondering, the language generated is $ab(ab)^*$.

□

Chapter 5
Strict Syntactic Restriction

5.1 Introduction

In this chapter we present a syntactic restriction for the productions of a macrogrammar. We show that if all productions of an OI macrogrammar satisfy this restriction, then the macrogrammar satisfies corollary 4.2.1 from the previous chapter. That is, the macrogrammar generates a context-free language under OI derivation.

We also use restricted productions to describe some example patterns. These examples are taken from actual JavaScript and Java language grammars [Int99, GJSB05]. By applying the restriction to these cases, we can get an idea of how concise and abstract the restriction allows us to be.

5.2 Definitions & Theorems

Definition 5.2.1 (Restricted production) Given a macrogrammar $MG = (\Sigma, \Gamma, P, S)$ and a rule $p : A \rightarrow \omega \in P$, p is in restricted form if for all substring $B(\bar{t})$ of ω, $t_i \in (T_\Gamma(\Sigma) \cup \{0, 1, \ \dots \ \mathsf{a}(A) - 1\})$.

Hence, a production is in restricted form if all macros on its right-hand side are instantiated using single formal parameters or terms with no parameters. Note that the latter may contain nested macros so long as they do not use formal parameters.

Intuitively, if all productions in a macrogrammar are in restricted form, then no new symbols are added to a parameter being carried from left to right in a derivation. This means that parameters do not grow in size and stay within a limit, which is exactly what we are chasing after to ensure the precondition of corollary 4.2.1 is satisfied. Note that this is true only if the macrogrammar is OI. □

EXAMPLE 5.2.1 (RESTRICTED & NON-RESTRICTED PRODUCTIONS)

Restricted productions:

$G \rightarrow f\ d\ G$

$A \rightarrow a\ B(b,\ C(r))$

$F \rightarrow G(0,\ 2)\ H(F(F(a)),\ 1,\ gh)\ G(f, 0)\ ff$

Non-restricted productions:

$G \rightarrow G(0,\ 1,\ g\ 2)$

$A \rightarrow A(a)\ m\ B(0\ C(n))$

$F \rightarrow H(G(0))$

□

DEFINITION 5.2.2 (STRICTLY RESTRICTED MACROGRAMMAR) A macrogrammar is in strict restricted form if all its productions are in restricted form. □

THEOREM 5.2.1 A strictly restricted macrogrammar generates a context-free language under OI derivation.

PROOF:

We will show that a restricted macrogrammar has no unlimited macros and therefore it generates a context-free language.

Let us consider a strictly restricted macrogrammar $MG = (\Sigma, \Gamma, P, S)$ and $CG = (V, E)$ its corresponding call graph.

We define the set $Terms$ as:

$$\frac{(A, B, (\ldots\ t\ \ldots)) \in E \qquad t \in T_{\Gamma}(\Sigma)}{t \in Terms}$$

That is, $Terms$ contains all the terms free of formal parameters which are used in some edge/production to instantiate a macro. The number of productions is finite, as it is the number of edges in CG, therefore $Terms$ is finite too.

Let max be the length of the term/s in $Terms$ with maximum length. That is, $max = l(t)$ for some $t \in Terms$ and $max \geq l(t)$ for all $t \in Terms$. $Terms$ is finite, so max always exists.

A derivation from the axiom is of the general form $Der : S \Rightarrow_I N_0(\overline{v}_0) \Rightarrow_I \ldots \Rightarrow_I N_m(\overline{v}_m)$ where $m \in \mathbb{N}$.

We show by induction on the instance subindex i that $l(v_{ij}) \leq max$ for any v_{ij} in Der.

BASIS:

If $i = 0$ then $v_{ij} = v_{0j}$. $\mathsf{a}(S) = 0$, therefore if $S \Rightarrow_I N_0(\overline{v}_0)$ then there has to be an edge $e \in E$ such that $e = (S, N_0, (v_{01}, \ldots, v_{0\mathsf{a}(N_0)}))$. Thus, by definition of $Terms$, $v_{0j} \in Terms$ for all v_{0j}. Therefore $l(v_{0j}) \leq max$ for any v_{0j}.

We have proved the base case.

INDUCTION STEP:

We assume $l(v_{kj}) \leq max$ for $k < m$. We know that $N_k(\overline{v}_k) \Rightarrow_I N_{k+1}(\overline{v}_{k+1})$. Therefore there is an $e \in E$, coming from a restricted production, such that either:

- $e = e_1 = (N_k, N_{k+1}, (r_1, \ldots, r_{\mathsf{a}(N_{k+1})}))$, where $r_j = v_{k+1j}$ or $0 \leq r_j < \mathsf{a}(N_k)$ and $v_{k+1j} = v_{kr_j}$.
- $e = e_2 = (N_k, Z, (d))$, where $0 \leq d < \mathsf{a}(N_k)$ and $v_{kd} = N_{k+1}(\overline{v}_{k+1})$.

If $e = e_1$. $N_{k+1}(\overline{v}_{k+1})$ is an instance, thus $v_{k+1j} \in T_\Gamma(\Sigma)$.

If $r_j = v_{k+1j}$ then by definition of $Terms$, $v_{k+1j} \in Terms$ and therefore $l(v_{k+1j}) \leq max$. Otherwise $v_{k+1j} = v_{kr_j}$, but by the induction hypothesis $l(v_{kr_j}) \leq max$, so $l(v_{k+1j}) \leq max$. We have showed $l(v_{k+1j}) \leq max$ for the $e = e_1$ case.

If $e = e_2$, then $v_{kd} = N_{k+1}(\overline{v}_{k+1})$. By definition of the length function $l(v_{k+1j}) < l(v_{kd})$ and therefore $l(v_{k+1j}) \leq l(v_{kd})$. Applying the induction hypothesis we get

$l(v_{k+1j}) \leq l(v_{kd}) \leq max$ which implies $l(v_{k+1j}) \leq max$ as we wanted to show for $e = e_2$.

We conclude that $l(v_{ij}) \leq max$ for any v_{ij} in Der. Since all derivations are of the form of Der, for any term t such that $S \overset{*}{\Rightarrow}_I F(\ldots\ t\ \ldots)$ it holds that $l(t) \leq max$.

As we saw in the proof of theorem 4.2.1, if $l(t) \leq max$ for any t such that $S \overset{*}{\Rightarrow}_I F(\ldots\ t\ \ldots)$, then MG generates a finite set of instances. Hence, it generates a context-free language under OI derivation, as we wanted to show.

□

5.3 Real-world patterns

In [TN08], Thiemann and Neubauer provide three patterns present in the JavaScript and Java grammar. They abstract the particular cases in which these patterns appear and rearrange them using macros. Since they do not impose any restriction on the shape of productions, they can fully abstract the patterns.

Thiemann's and Neubauer's notation is slightly different to our own. They use ':' instead of '→'. They also write macros as tuples where the first element is the macro name. The elements of the tuple (macro name and parameters) are separated by blank spaces, instead of commas. For example: $(Macro\ Param1\ Param2\ Param3)$

In these examples we present their rearrangement of patterns using their notation. Then we attempt to do the same restricting the rules and see how much abstraction we are forced to sacrifice.

5.3.1 Patterns

Example 5.3.1 (Operator Precedence) JavaScript uses a hierarchical structure of nonterminals to express precedence among binary operators. Namely:

$$\begin{array}{rcl} \mathit{RelationalExpNoIn} & : & \mathit{ShiftExp} \\ & | & \mathit{RelationalExpNoIn}\ \text{'}<\text{'}\ \mathit{ShiftExp} \\ \mathit{ShiftExp} & : & \mathit{AdditiveExp} \\ & | & \mathit{ShiftExp}\ \text{'}\ll\text{'}\ \mathit{AdditiveExp} \\ \mathit{AdditiveExp} & : & \mathit{MultiplicativeExp} \end{array}$$

$$| \quad \mathit{AdditiveExp}\ \text{'+'}\ \mathit{MultiplicativeExp}$$

Here each nonterminal expresses the precedence position of its corresponding operator. Thiemann and Neubauer capture this structure by defining a binary operation macro:

$$(\mathit{BOp}\ \mathit{op}\ \mathit{base})\ :\ \mathit{base} \mid (\mathit{BOp}\ \mathit{op}\ \mathit{base})\ \mathit{op}\ \mathit{base}$$

and then using it by nesting three instances of it:

$$\begin{aligned} RelationalExpNoIn\ :\ & (\mathit{BOp}\ \text{'} < \text{'} \\ & \quad (\mathit{BOp}\ \text{'} \ll \text{'} \\ & \quad\ (\mathit{BOp}\ \text{'} + \text{'}\ MultiplicativeExp))) \end{aligned}$$

This macro nesting supplies the necessary precedence in a more concise way as well as avoids the presence of new nonterminals.

Using our notation, we can write the *BOp* definition as:

$$\mathit{BOp} \to 1 \mid \mathit{BOp}(0,\ 1)\ 0\ 1$$

and the definition of $RelationalExpNoIn$ as:

$$\begin{aligned} RelationalExpNoIn \to\ & \mathit{BOp}\ (\text{'} < \text{'}, \\ & \quad \mathit{BOp}\ (\text{'} \ll \text{'}, \\ & \quad\quad \mathit{BOp}\ (\text{'} + \text{'},\ MultiplicativeExp))) \end{aligned}$$

The only limitation our strict syntactic restriction imposes is that formal parameters must appear alone when being used as actual parameters on the right-hand side of a production. As we can see, our definitions for *BOp* and $RelationalExpNoIn$ both comply with this; $RelationalExpNoIn$ does so trivially since it is a nullary macro. Note that, in this case, the restriction allows us to express nesting.

Thus, we have been able to abstract the pattern just like Thiemann and Neubauer did. We were not forced to lose abstraction. □

Example 5.3.2 (Similar structures in different contexts) It is possible, in a computer language, to find the same structures with slight variations in different contexts. For example, allowed modifier tokens for variables can be different depending on the context a variable is declared in.

Since there are small differences among these structures, using a BNF grammar to describe our language forces us to replicate, for every context, the nonterminals and rules defining the structure; applying the differences in each case.

Particularly, in JavaScript relational expressions must not use the relational operator in inside the header of a for statement. This is so because in such context the reserved word in has a different meaning. However any other relational expression not containing in is allowed. The JavaScript BNF grammar solves this by duplicating the rules for relational expressions:

$$
\begin{array}{ll}
RelationalExp & : ShiftExp \\
 & \mid RelationalExp \text{ ' } < \text{ ' } ShiftExp \\
 & \mid RelationalExp \text{ '}in\text{' } ShiftExp \\
RelationalExpNoIn & : ShiftExp \\
 & \mid RelationalExpNoIn \text{ ' } < \text{ ' } ShiftExp \\
EqualityExp & : RelationalExp \\
 & \mid EqualityExp \text{ ' } == \text{ ' } RelationalExp \\
EqualityExpNoIn & : RelationalExpNoIn \\
 & \mid EqualityExpNoIn \text{ ' } == \text{ ' } RelationalExpNoIn \\
BitANDExp & : EqualityExp \\
 & \mid BitANDExp \text{ '\&' } EqualityExp \\
BitANDExpNoIn & : EqualityExpNoIn \\
 & \mid BitANDExpNoIn \text{ '\&' } EqualityExpNoIn
\end{array}
$$

Thiemann and Neubauer abstract out the difference in both structures, in this case the appliable relational operators, and pass this information down the structure via a formal parameter:

$$(BitANDExp\ ops)\ :\ (BOp\ \text{'\&'}\ (BOp\ \text{'}==\text{'}\ (BOp\ ops\ ShiftExp)))$$

then they associate a nonterminal to each set of appliable operators and instantiate the macro twice to represent the two structures:

$$\begin{aligned}
&REOpsIn\ :\ \text{'}<\text{'}\ |\ \text{'}in\text{'}\\
&REOpsNoIn\ :\ \text{'}<\text{'}\\
&BitANDExpIn\ :\ (BitANDExp\ REOpsIn)\\
&BitANDExpNoIn\ :\ (BitANDExp\ REOpsNoIn)
\end{aligned}$$

Unfortunately, in this case the rule which defines $BitANDExp$ is not complying with the syntactic restriction. The formal parameter *ops* appears nested inside of a macro. Hence we do not have a straightforward translation from Thiemann and Neubauer notation to our own. If we want to represent this pattern and comply with the restriction, we have to sacrifice some abstraction.

Macros $BitANDExpIn$ and $BitANDExpNoIn$ state the choice of operators for each context by instantiating $BitANDExp$ with different operators $REOpsIn$ and $REOpsNoIn$. Since we have to delete the *ops* formal parameter, this information (that is, the choice) cannot be passed down along the structures. It has to be explicitly stated when the divergence occurs between the two structures. This forces us to replicate the structure up to the point of the difference. That is:

$$\begin{aligned}
&BitANDExpIn \rightarrow BOp\ (\text{'\&'}\ BOp\ (\text{'}==\text{'}\ BOp\ (REOpsIn,\ ShiftExp)))\\
&BitANDExpNoIn \rightarrow BOp\ (\text{'\&'}\ BOp\ (\text{'}==\text{'}\ BOp\ (REOpsNoIn,\ ShiftExp)))
\end{aligned}$$

Nonterminals $REOpsIn$ and $REOpsNoIn$ remain unchanged:

$$REOpsIn \rightarrow \text{ ' < ' } \mid \text{'}in\text{'}$$
$$REOpsNoIn \rightarrow \text{ ' < ' }$$

and the macro $BitANDExp$ disappears.

We have been forced to explicitly define part of both structures, despite it being common and prone to be abstracted. Nonetheless our new portion of grammar is much shorter and readable than the original JavaScript piece of grammar, thanks to the *BOp* macro.

So, applying the strict syntactic restriction, we cannot express this pattern as abstractly as we would like. But we can still generalize much of it and remove the cumbersome stage replicating rules of the JavaScript grammar. □

Example 5.3.3 (Permutation Phrases) Java grammar contains a pattern common to many programming languages: Permutation phrases. A permutation phrase consists of a finite set of alternatives, which may appear in any order, but each of which may appear at most once. The following portion of Java grammar shows this case for field modifiers:

$$\begin{aligned} FieldModifiers \quad & : FieldModifier \\ & \mid FieldModifiers\ FieldModifier \\ FieldModifier \quad & : \text{ '}static\text{'} \\ & \mid \text{'}final\text{'} \\ & \mid \text{'}public\text{'} \end{aligned}$$

However this BNF rules do not capture the condition that no modifier may appear repeated. Checking this condition is left to the semantic analyzer; though it could be syntactically captured, it would result in a cumbersome set of BNF rules.

Macros can capture this condition in the following way:

$$(PermP3\ m1\ m2\ m3) \quad : \ /*\,empty*/ \tag{5.1}$$
$$\mid m1\ (PermP3\ NUL\ m2\ m3) \tag{5.2}$$
$$\mid m2\ (PermP3\ m1\ NUL\ m3) \tag{5.3}$$
$$\mid m3\ (PermP3\ m1\ m2\ NUL) \tag{5.4}$$

$$/*\,no\ production\ for\ NUL*/$$
$$FieldModifiers\ :\ (PermP3\ 'static'\ 'final'\ 'public')$$

Once a modifier has been used, it is discarded by substituting it with the NUL nonterminal, which is not defined and therefore non-generative.

Since our objective is to obtain a context-free grammar, we must provide a means to specialize the macrogrammar into a BNF grammar at a latter stage of the parsing process. It is at this point, that rules containing NUL can be discarded. Non-terminals not defined are discarded too; from this portion of grammar, NUL and $(PermP3\ NUL\ NUL\ NUL)$ would be left with no rule defining them and therefore eliminated.

Productions 5.2, 5.3 and 5.4 satisfy the syntactic restriction but rule 5.1 is an empty rule . Our definition for macrogrammars does not accept empty productions or passing the empty word as parameter to a macro. By rule 5.1, it is possible that nonterminal $FieldModifiers$ generates the empty word, however the original Java grammar definition does not allow this for $FieldModifiers$ [GJSB05], it must always generate at least one modifier.

We can take advantage of this and represent the pattern in our macrogrammars formalism as follows:

$$
\begin{aligned}
PermP3 \rightarrow\ & 0 \\
& |\ 1 \\
& |\ 2 \\
& |\ 0\ PermP3(NUL,\ 1,\ 2) \\
& |\ 1\ PermP3(0,\ NUL,\ 2) \\
& |\ 2\ PermP3(0,\ 1,\ NUL)
\end{aligned}
$$

$$
/ * no\ production\ for\ NUL * /
$$

$$
FieldModifiers \rightarrow PermP3(\text{'}static\text{'},\ \text{'}final\text{'},\ \text{'}public\text{'})
$$

We can express this pattern too using the strict restriction. □

5.3.2 Conclusion

Of the three patterns presented, we have been able to directly express two of them complying with our strict syntactic restriction. To capture the third pattern we have been forced to sacrifice some abstraction, but we have still been able to remove the full replication of BNF rules and symbols.

Chapter 6

Lenient Syntactic Restriction

6.1 Introduction

In the previous chapter we introduced a (strict) syntactic restriction and studied how it allowed us to abstract different patterns present in actual, used context-free grammars.

In this chapter we define a new, more lenient syntactic restriction. While the strict restriction was defined on a production basis, this new restriction applies to the whole production set. In general, single productions are not forced to fulfill a restriction of form. We will go on to prove if a macrogrammar satisfies the lenient restriction then it satisfies corollary 4.2.1.

After we have defined precisely the new restriction, we apply it to the same patterns we used in the previous chapter. We will see how now we can fully abstract all patterns.

Finally, in the last section, we compare our results in this chapter with what Thiemann and Neubauer obtained in [TN08].

6.2 Definitions & Theorems

Definition 6.2.1 (Presence Graph) The presence graph is an alternative representation of the production set, very much like the call graph. The call graph grabs in its vertices those macros which appear as substrings in productions and therefore are explicitly invoked . But macros can also be invoked using formal parameters. In such case, the call graph contains an edge ending in the special vertex Z. However no vertices come out of Z, so, by simply looking at the graph, we cannot know what macros can be invoked from that point on.

The presence graph is devised to capture this missing information. It grabs all

macros present as subterm in some production, effectually capturing those macros invoked later using formal parameters. Grabbed macros can be later invoked or not, so it actually captures more macros than the ones missing in the call graph. But that is irrelevant for our purposes, as we will see later.

Note that for a macro to be invoked it must be present as subterm (substrings are subterms as well) somewhere. Intuitively, the set of present macros is a superset of the set of invoked macros.

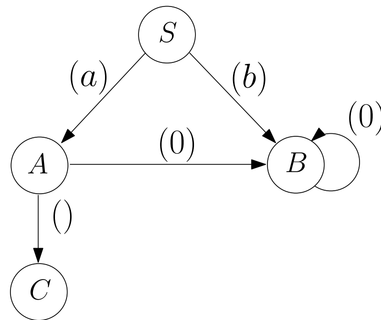
Given a macrogrammar $MG = (\Sigma, \Gamma, P, S)$, we build the presence graph $PG = (V, E)$ of MG as:

$$\frac{A \in N}{A \in V} \qquad \frac{v_1, v_2 \in V, v_1 \to \omega \in P \qquad \omega[v_2(\bar{t})] \text{ for some } t_1, \ldots, t_{\mathsf{a}(v_2)} \in T_{\mathsf{a}(V_1)}}{(v_1, v_2, (t_1, \ldots, t_{\mathsf{a}(v_2)})) \in E}$$

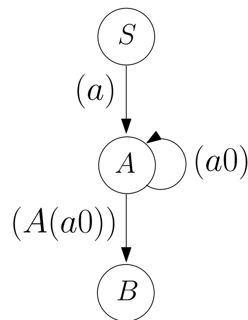
□

Example 6.2.1 (Example presence graphs) We show the presence graph for macrogrammars $GrGrammar1$ and $GrGrammar2$ from example 4.2.1:

$$\begin{aligned} GrGrammar1: S &\to A(a) \mid B(b) \\ A &\to B(0)C \\ C &\to c \\ B &\to 0B(0) \mid 0 \end{aligned}$$



$$
\begin{aligned}
GrGrammar2 : S &\rightarrow A(a) \\
A &\rightarrow B(A(a0)) \mid 0 \\
B &\rightarrow 0
\end{aligned}
$$

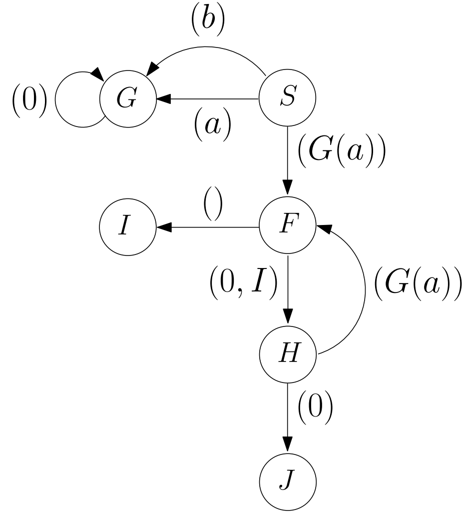


Compare this graph with the call graph for this same macrogrammar in example 4.2.1. If we want to know which macros are invoked after having reached Z we need to make derivations or use some other similar mechanism.

As opposed to it, the presence graph has captured the information of all macros being possibly invoked at some point. Note it is irrelevant that there are no edges coming out of B, this just means B is not instantiating macros of its own. The macros B may invoke have been captured somewhere else in the graph, in this case in A's self-referring edge.

We show the presence graph for the following macrogrammar too:

$$
\begin{aligned}
GrGrammar3 : S &\rightarrow F(G(a))\ G(b) \\
F &\rightarrow H(0,\ I) \\
G &\rightarrow 0 \mid 0\ G(0) \\
H &\rightarrow J(0)\ F(G(a)) \\
J &\rightarrow 0
\end{aligned}
$$



In this last example, the I macro is captured in the graph, despite the fact it is never invoked on a right-hand side (the defining rule for H does not use parameter 1).

□

DEFINITION 6.2.2 (PRESENCE DERIVATION) We define the presence derivation to generate all instances present in some instance derivation and those instances we can generate from these, even if the latter ones do not appear in an instance derivation. We note it $\Rightarrow_P$.

For a presence graph $PG = (V, E)$:

$$\frac{(A, B, (tb_1, \ldots, tb_{\mathsf{a}(B)})) \in E \qquad A(\overline{ta}) \text{ is an instance}}{A(\overline{ta}) \Rightarrow_P B(\overline{tb'}) \qquad tb_j^{'} = tb_j[0 \rightarrow ta_1, 1 \rightarrow ta_2 \ \ldots, \ \mathsf{a}(A) - 1 \rightarrow ta_{\mathsf{a}(A)}]}$$

We can write $\Rightarrow_{PG,P}$ to state the presence graph explicitely. We write the reflexive transitive closure as $\stackrel{*}{\Rightarrow}_P$. □

Example 6.2.2 (Example presence derivations)

For macrogrammar $GrGrammar2$:

$S \Rightarrow_P A(a) \Rightarrow_P B(A(aa))$

$S \Rightarrow_P A(a) \Rightarrow_P A(aa)$

$S \Rightarrow_P A(a) \Rightarrow_P A(aa) \Rightarrow_P B(A(aaa))$

$S \Rightarrow_P A(a) \Rightarrow_P A(aa) \Rightarrow_P A(aaa) \Rightarrow_P A(aaaa)$

As we can see, we can generate the same instances we generated using instance derivation, and, in this case, we cannot generate more. We could use presence derivation to generate more if a macro did not use all its parameters on the right-hand side of some rule defining it. We have not defined a normal form for our macrogrammars, but it is easy to see a macrogrammar in which the previous happens is not a "well formed" macrogrammar. Therefore, in general, the set of instances generated by presence derivation and the set of instances generated by instance derivation will be the same.

For macrogrammar $GrGrammar3$:

$S \Rightarrow_P G(a) \Rightarrow_P G(a)$

$S \Rightarrow_P G(b) \Rightarrow_P G(b)$

$S \Rightarrow_P F(G(a)) \Rightarrow_P H(G(A),\ I) \Rightarrow_P J(G(A))$

$S \Rightarrow_P F(G(a)) \Rightarrow_P I$

$GrGrammar$ is an example of a not "well formed" macrogrammar. Macro H does not use its second parameter. In consequence, we can generate the nullary macro I using presence derivation, but not instance derivation.

□

Let us show now that presence derivation generates, at least, the instances generated by instance derivation.

Lemma 6.2.1 Given a macrogrammar MG and its corresponding presence graph

$PG = (E_P, V_P)$ and call graph $CG = (E_C, V_C)$. The set of instances generated by PG under presence derivation is a superset of the set of instances generated by CG under instance derivation.

That is, if $S \overset{*}{\Rightarrow}_I q$ then $S \overset{*}{\Rightarrow}_P q$.

PROOF:

We use induction on the number of derivation steps in $S \overset{*}{\Rightarrow}_I q$.

BASIS:

If $S \overset{*}{\Rightarrow}_I S$ then $S \in V_C$. $S \in V_C$ only if $S \in N$. If $S \in N$ then $S \in V_P$. Finally, if $S \in V_P$ then $S \overset{*}{\Rightarrow}_P S$.

INDUCTIVE STEP:

Let us assume that if $S \overset{*}{\Rightarrow}_I A(\bar{t})$ then $S \overset{*}{\Rightarrow}_P A(\bar{t})$.

For any instance $B(\overline{tb'})$ such that $A(\bar{t}) \Rightarrow_I B(\overline{tb'})$ there has to exist an $e_I \in E_I$ of either the form:

- $e_I = (A, B, (tb_1, \ldots, tb_{\mathsf{a}(B)})$ and $tb_i' = tb_i[0 \mapsto t_1, 1 \mapsto t_2 \ \ldots \ \mathsf{a}(A) - 1 \mapsto t_{\mathsf{a}(A)}]$.
- $e_I = (A, Z, (j))$ for some $0 \leq j < \mathsf{a}(A)$ and $B(\overline{tb'})$ is a substring of t_j. If $B(\overline{tb'})$ is a substring of t_j then $t_j = \varepsilon \ B(\overline{tb'}) \ \zeta$ for some ε, $\zeta \in T_\Gamma(\Sigma)^*$.

But e_I exists only if there is an $r \in P$ such that either:

- $r = r_1 = A \rightarrow \gamma \ B(\overline{tb}) \ \delta$.
- $r = r_1 = A \rightarrow \gamma \ j \ \delta$.

Where γ, $\delta \in T_\Gamma(\Sigma)^*$ in both cases.

If $r = r_1$ then there is an $e_P = (A, B, (tb_1, \ldots, tb_{\mathsf{a}(B)})) \in E_P$. Therefore $A(\bar{t}) \Rightarrow_P B(\overline{tb'})$ and $S \overset{*}{\Rightarrow}_P B(\overline{tb'})$ as we wanted to show.

If $r = r_2$ then $t_j = \varepsilon \ B(\overline{tb'}) \ \zeta$, so $t_j[B(\overline{tb'})]$

Since $S \overset{*}{\Rightarrow}_P A(\bar{t})$, there has to exist a derivation $S = N_0(\overline{v}_0) \Rightarrow_P \ldots \Rightarrow_P N_m(\overline{v}_m) = A(\bar{t})$. Take an index $0 \leq k \leq m$. We take the lowest k such that for some parameter

v_{ki} it holds that $v_{ki}[B(\overline{tb'})]$. Let us call it l. Note that since $\mathsf{a}(S) = 0$, $0 < l \leq m$. Note too that l always exists since the condition always holds for $l = m$, that is, $v_{lj} = t_j$ and $t_j[B(\overline{tb'})]$, as we saw before.

Consider the one step derivation $N_{l-1}(\overline{v}_{l-1}) \Rightarrow_P N_l(\overline{v}_l)$. There is an edge $(N_{l-1}, N_l, (w_1, \ldots, w_{\mathsf{a}(N_l)})) \in E_P$ such that $v_{li} = w_i[0 \rightarrow v_{l-11}, \ldots, \mathsf{a}(N_{l-1}) - 1 \rightarrow v_{l-1\mathsf{a}(N_{l-1})}]$ and at least one v_{li} with $v_{li}[B(\overline{tb'})]$. But there is no v_{l_1g} such that $v_{l-1g}[B(\overline{tb'})]$, since l is the lowest k for which that holds.

Thus, $v_{li}[B(\overline{tb'})]$, $v_{li} = w_i[0 \rightarrow v_{l-11}, \ldots, \mathsf{a}(N_{l-1}) - 1 \rightarrow v_{l-1\mathsf{a}(N_{l-1})}]$ and there is no g such that $v_{l-1g}[B(\overline{tb'})]$. Therefore there must exist $x_1, \ldots, x_{\mathsf{a}(B)} \in T_{\mathsf{a}(N_{l-1})}$ such that $w_i[B(\overline{x})]$ and $tb'_h = x_h[0 \rightarrow v_{l-11}, \ldots, \mathsf{a}(N_{l-1}) - 1 \rightarrow v_{l-1\mathsf{a}(N_{l-1})}]$.

But if $w_i[B(\overline{x})]$ and $N_l(\overline{w}_l)[w_i]$ then $N_l(\overline{w}_l)[B(\overline{x})]$. Hence there is an edge $(N_{l-1}, B, (x_1, \ldots, x_{\mathsf{a}(B)})) \in E_P$ and therefore $N_{l-1}(\overline{v}_{l-1}) \Rightarrow_P B(\overline{tb'})$. Finally $S \stackrel{*}{\Rightarrow}_P B(\overline{tb'})$ as we wanted to show.

We conclude that if $S \stackrel{*}{\Rightarrow}_I q$ then $S \stackrel{*}{\Rightarrow}_P q$.

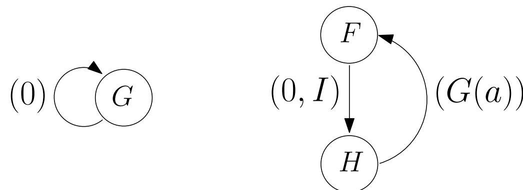
□

Definition 6.2.3 (Cycle) Given a labeled directed graph $G = (V, E)$. A set $C \subseteq E$ is a cycle if:

- If $e = (s, t, l) \in C$, then there exist $e_1 = (s_1, t_1, l_1) \in C$ and $e_2 = (s_2, t_2, l_2) \in C$ such that $s = t_1$ and $t = s_2$.
- No subset of C, except for C and $\emptyset$, fufills the previous condition.

□

Example 6.2.3 (Example Cycles) *GrGrammar*3 contains the two following cycles: $C_G = \{(G,\ G,\ (0))\}$ and $C_{FH} = \{(F,\ H,\ (0,\ I)), (H,\ F, (G(a)))\}$. Graphically:



□

Definition 6.2.4 (Restricted Cycle) Given a presence graph $PG = (V, E)$. A cycle $C \subseteq E$ is in restricted form if either:

- If $e \in C$, then e is in strict restricted form according to definition 5.2.1.
- There is an $e = (A, B, (t_1, \ldots, t_{\mathsf{a}(B)})) \in C$ such that $t_1, \ldots, t_{\mathsf{a}(B)} \in T_\Gamma(\Sigma)$.

Note that the edges are made out of productions and we can apply the strict syntactic restriction to them. More precisely, $e = (A, B, (\bar{t}))$ is in strict restricted form if $t_i \in (T_\Gamma(\Sigma) \cup \{0, 1, \ \ldots \ \mathsf{a}(A) - 1\})$.

In the original restriction, we ensured that parameters did not keep stacking up new symbols. With this new restriction, we allow them to do so, but we want to make sure that they will stop at some point.

The intuition is that for a parameter to keep adding new symbols, macros must be recursive. Recursions are captured by the presence graph in the form of cycles. Once we have detected a cycle, we want the flow of new parameters through it to come to a stop. We force this in two possible manners: either we force all edges in the cycle to be in strict syntactic form, or we force one of the edges in the cycle to not use formal parameters, thus interrupting the stacking process.

Note that if an edge does not use formal parameters, it does not imply the rule it comes from is not "well formed", because a production has more information than an edge and a single production can produce more than one edge. □

Definition 6.2.5 (Leniently restricted macrogrammar) Let MG be a macrogrammar and $PG = (V, E)$ its corresponding presence graph. MG is in lenient restricted form if all cycles in E are restricted.

Note this definition subsume that of strict restriction. All strictly restricted macrogrammars are leniently restricted too. □

Now we proceed to show that a leniently restricted macrogrammar generates a context-free grammar. First we need to state and prove the following lemma.

LEMMA 6.2.2 Given a presence graph $PG = (V, E)$. Let $Terms = \{t \mid (A, B, (\ldots t \ldots)) \in E$ for some $A, B \in N\}$. Let $a = l(t_{max})$ such that $t_{max} \in Terms$ and $l(t_{max}) \geq l(t)$ for all $t \in Terms$.

Given an instance $A(\bar{t})$ such that $l(t_j) \leq a$ for all t_j and given a presence derivation of the form $A(\bar{t}) = N_0(\overline{v}_0) \Rightarrow_P \ldots \Rightarrow_P N_n(\overline{v}_n)$.

If the derivation uses $0 \leq k \leq n$ edges from E not in restricted form, then $l(v_{ij}) \leq \frac{a*(a^{k+1})-1}{a-1}$ for all v_{ij} in the derivation.

PROOF:

We define the following recurrence:

$$lim_0 = a$$
$$lim_1 = lim_0 * a + a$$
$$\vdots$$
$$lim_M = lim_{m-1} * a + a$$

By definition $a \geq 0$, thus $lim_{r+1} \geq lim_r$.

We show by induction on the length n of the derivation that if the derivation uses k edges not in restricted form then $l(v_{ij}) \leq lim_k$.

BASIS:

$n = 0$ and no edges are used, therefore $k = 0$, $lim_k = lim_0 = a$ and $v_{0j} = t_j$. Since $l(t_j) \leq a$, it follows that $l(v_{0j}) \leq lim_k$ for all v_{0j}.

INDUCTIVE STEP:

We assume that for $i \leq n$, $l(v_{ij}) \leq lim_k$ and that the derivation has used k edges not in restricted form.

Now we derive one more step $A(\bar{t}) \overset{*}{\Rightarrow}_P N_n(\overline{v}_n) \Rightarrow_P N_{n+1}(\overline{v}_{n+1})$. Let $e = (N_n, N_{n+1}, (x_1, \ldots, x_{\mathsf{a}(N_{n+1})}))$ be the edge used.

If e is in restricted form then $x_i \in (T_\Gamma(\Sigma) \cup \{0, 1, \ldots, \mathsf{a}(N_n)\})$ and the derivation still uses k edges not in restricted form.

If $x_i \in T_\Gamma(\Sigma)$ then $v_{n+1i} = x_i$. By definition of a, $l(x_i) \leq a$, so $l(v_{n+1i}) \leq a$. Besides

$a \leq lim_k$, hence $l(v_{n+1i}) \leq lim_k$ for all v_{n+1i}.

If $x_i \in \{0, 1, \ldots, \mathsf{a}(N_n)\}$ then $v_{n+1i} = v_{nj}$ for some v_{nj} and by hypothesis, $l(v_{nj}) \leq lim_k$ for all v_{nj}. Therefore $l(v_{n+1i}) \leq lim_k$ for all v_{n+1i}.

Therefore if e is in restricted form, the derivation uses k edges not in restricted form and for $i \leq n+1$, $l(v_{ij}) \leq lim_k$. The condition holds.

If e is not in restricted form then $x_i \in T_{\mathsf{a}(N_n)}$. As previously, $l(x_i) \leq a$ for all x_i. Suppose a term x_i uses p formal parameters and q symbols from Σ or N, then $p + q = l(x_i)$ and $p + q \leq a$. Thus $p \leq a$ and $q \leq a$.

The length of v_{n+1i} is given by $l(v_{n+1i}) = l(v_{nz_1}) + l(v_{nz_2}) + \ldots + l(v_{nz_p}) + q$ where $z_1, z_2, \ldots, z_p \in \{0, 1, \ldots, \mathsf{a}(N_n)\}$. By hypothesis $l(v_{nj}) \leq lim_k$ for all v_{nj}, therefore $l(v_{nz_1}) + l(v_{nz_2}) + \ldots + l(v_{nz_p}) \leq lim_k + \overset{p-2}{\ldots} + lim_k = p * lim_k$. Hence $l(v_{n+1i}) \leq p * lim_k + q$. But p $\leq a$ and $q \leq a$, so $l(v_{n+1i}) \leq a * lim_k + a$. By definition of the recurrence $lim_{k+1} = lim_k * a + a$, therefore $l(v_{n+1i}) \leq lim_{k+1}$.

We know that $lim_{k+1} \geq lim_k$. Therefore $l(v_{ij}) \leq lim_{k+1}$ for $i \leq n+1$. Since e is not in restricted form, the derivation uses $k+1$ edges not in restricted form. We have shown the condition holds for the inductive step.

Now we can solve the recurrence and we get: $lim_m = \frac{a*(a^{m+1})-1}{a-1}$. This means that for any derivation using k edges not in restricted form, $l(v_{ij}) \leq \frac{a*(a^{k+1})-1}{a-1} = lim_k$ for all the parameter terms in the derivation, as we wanted to show.

□

THEOREM 6.2.1 A leniently restricted macrogrammar generates a context-free language under OI derivation.

PROOF:

For a given macrogrammar MG in lenient restricted form. By lemma 6.2.1 the set of instances generated by presence derivation is a superset of the set of instances generated by instance generation.

Suppose we can find a natural number q such that if $S \overset{*}{\Rightarrow}_P A(\bar{t})$ then $l(t_i) \leq q$ for all t_i. Therefore if $S \overset{*}{\Rightarrow}_I A(\bar{t})$ then $l(t_i) \leq q$ for all t_i and thus there are no unlimited macros in MG.

In such case, by corollary 4.2.1, MG generates a context-free grammar under OI derivation. We finally see why it is irrelevant that present derivation generates potentially more instances than instance derivation. A limit for the former will yield a limit for the latter.

Let us try to find such q. Let a be the value a defined in 6.2.2. Take the presence graph $PG = (V, E)$ of MG. Let $|E|$ be, as usual, the cardinality of E. We define q as $q = \frac{a*(a^{|E|+1})-1}{a-1}$, note that $\frac{a*(a^{val+1})-1}{a-1} \leq q$ for all $val \leq |E|$. We say an instance $F(\bar{t})$ is a prime if $l(t_j) \leq a$ for all $l(t_j)$.

Take a derivation $Der : N_0(\overline{v}_0) \Rightarrow_P \ldots \Rightarrow_P N_n(\overline{v}_n)$ such that $N_0(\overline{v}_0)$ is a prime. Suppose Der uses $|E|$ or fewer edges not in restricted form, then by 6.2.2, $l(v_{ij}) \leq q$ for all v_{ij} in Der. Derivation length n is the number of used edges (restricted or not), thus if $\underline{n \leq |E|}$, $l(v_{ij}) \leq q$ for all v_{ij} in Der.

Now let us assume $\underline{n > |E|}$. The set E has only $|E|$ edges, therefore there has to be at least two one-step subderivations in Der that use the same edge from E. But if an edge is repeated, it means that we can, in the derivation, reach the edge from itself and therefore the edge is part of a cycle. So Der uses at least one cycle.

MG is in lenient restricted form by assumption. Thus, for any cycle C which Der uses, either:

1. e is restricted form for all $e \in C$.
2. There is an $e = (A, B, (t_1, \ldots, t_{\mathsf{a}(B)})) \in C$ such that $t_1, \ldots, t_{\mathsf{a}(B)} \in T_\Gamma(\Sigma)$.

Suppose <u>all cycles used by Der are of type 1</u>. Then all edges in those cycles are in restricted form. Thus, if Der uses an edge which is not in restricted form, that edge is not in a cycle. Such edge can be used by Der at most in one one-step subderivation, otherwise it would be in a cycle.

PG contains $|E|$ edges, therefore it contains at most $|E|$ edges not in cycles. Hence Der can use at most $|E|$ different edges not in a cycle and thus, Der can use at most $|E|$ different edges not in restricted form, each of which can be used by Der at most once. Therefore Der uses at most $|E|$ edges not in restricted form, so $l(v_{ij}) \leq q$ for all v_{ij} in Der.

If not all cycles used by Der are of type 1 then, Der uses at least one cycle of type 2. In such case Der uses at least one edge e of the form $e = (A, B, (t_1, \ldots, t_{\mathsf{a}(B)}))$ such that $t_1, \ldots, t_{\mathsf{a}(B)} \in T_\Gamma(\Sigma)$. Suppose e is used in the r-th one-step derivation, then, by definition of the presence derivation, $N_r(\overline{v}_r)$ is a prime.

We can split Der into two non-empty subderivations $Der1 : N_0(\overline{v}_0) \Rightarrow_P \ldots \Rightarrow_P N_{r-1}(\overline{v}_{r-1})$ and $Der2 : N_r(\overline{v}_r) \Rightarrow_P \ldots \Rightarrow_P N_n(\overline{v}_n)$ such that both $N_0(\overline{v}_0)$ and $N_r(\overline{v}_r)$ are primes. Therefore, if either $Der1$ or $Der2$ is longer than $|E|$ and uses at least one cycle of type 2, then we can split it again in the same manner. Otherwise, as we saw before, $l(v) \leq q$ for all terms v in $Der1$ and $Der2$.

Since the subdividing derivations are never empty, if we keep splitting up the successive subderivations, their length will eventually drop below $|E|$. But we saw previously that if $n \leq |E|$ for a derivation which begins with a prime, then $l(v) \leq q$ for all its terms v. Thus, a presence derivation Der starting with a prime, can always be recursively split into subderivations $SDer_1, SDer_2, \ldots, SDer_z$ such that $l(v_{SDer_i}) \leq q$ for all term v_{SDer_i} in $SDer_i$, so $l(v) \leq q$ for all v in Der (in case Der does not use any cycle of type 2, the splitting is trivial, there is no splitting at all).

The grammar axiom S has no parameters, therefore it trivially satisfies the condition for prime. Hence, for all derivation of the form $S \Rightarrow_P \ldots \Rightarrow_P B(\bar{t})$, it holds that $l(t_i) \leq q$ and finally, if $S \overset{*}{\Rightarrow}_P B(\bar{t})$, then $l(t_i) \leq q$ for all t_i in both cases.

As we saw at the beginning of the proof, we can conclude that MG generates a context-free grammar under OI derivation, as we wanted to show.

□

6.3 Real-world patterns

In the previous chapter we used the strict syntactic restriction to abstract three grammar patterns present in the Java and JavaScript languages. We saw one of the patterns could not be abstracted completely.

In this section we apply the new lenient restriction to the patterns and see this time we are not restrained by any limitation.

6.3.1 Patterns

EXAMPLE 6.3.1 (OPERATOR PRECEDENCE) Let us recall the pattern, a staging set of productions capturing precedence:

$$\begin{aligned}
\textit{RelationalExpNoIn} &: \textit{ShiftExp} \\
&\mid \textit{RelationalExpNoIn}\ \text{'<'}\ \textit{ShiftExp} \\
\textit{ShiftExp} &: \textit{AdditiveExp} \\
&\mid \textit{ShiftExp}\ \text{'}\ll\text{'}\ \textit{AdditiveExp} \\
\textit{AdditiveExp} &: \textit{MultiplicativeExp} \\
&\mid \textit{AdditiveExp}\ \text{'+'}\ \textit{MultiplicativeExp}
\end{aligned}$$

and the abstraction proposed by Thiemann and Neubauer:

$$\begin{aligned}
(\textit{BOp op base}) &: \textit{base} \\
&\mid (\textit{BOp op base})\ \textit{op base}
\end{aligned}$$

$$\begin{aligned}
RelationalExpNoIn : (&BOp\ '<' \\
&(BOp\ '\ll' \\
&\ (BOp\ '+'\ MultiplicativeExp)))
\end{aligned}$$

Below is the pattern written using our notation:

$$BOp \rightarrow 1 \mid BOp(0,\ 1)\ 0\ 1$$

$$\begin{aligned}
RelationalExpNoIn \rightarrow BOp\ (&'<', \\
&BOp\ ('\ll', \\
&\ BOp\ ('+',\ MultiplicativeExp)))
\end{aligned}$$

We were not forced to lose abstraction before, and strict restriction is included in the lenient one. Naturally, the pattern results exactly with the same aspect as it did in the strict restriction case. □

EXAMPLE 6.3.2 (SIMILAR STRUCTURES IN DIFFERENT CONTEXTS) The original pat-

tern was:

$$
\begin{array}{ll}
RelationalExp & : ShiftExp \\
 & \mid RelationalExp\ '<'\ ShiftExp \\
 & \mid RelationalExp\ 'in'\ ShiftExp \\
RelationalExpNoIn & : ShiftExp \\
 & \mid RelationalExpNoIn\ '<'\ ShiftExp \\
EqualityExp & : RelationalExp \\
 & \mid EqualityExp\ '=='\ RelationalExp \\
EqualityExpNoIn & : RelationalExpNoIn \\
 & \mid EqualityExpNoIn\ '=='\ RelationalExpNoIn \\
BitANDExp & : EqualityExp \\
 & \mid BitANDExp\ '\&'\ EqualityExp \\
BitANDExpNoIn & : EqualityExpNoIn \\
 & \mid BitANDExpNoIn\ '\&'\ EqualityExpNoIn
\end{array}
$$

and the original abstraction made by the German authors:

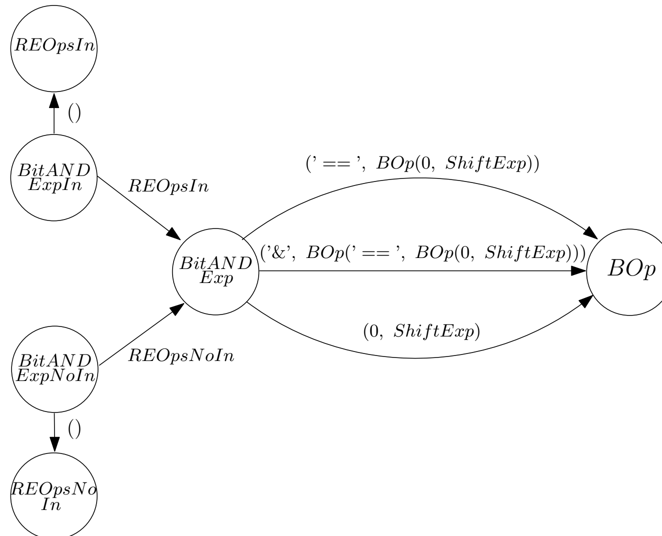
$$
\begin{array}{l}
(BitANDExp\ ops)\ :\ (BOp\ '\&'\ (BOp\ '=='\ (BOp\ ops\ ShiftExp))) \\
REOpsIn\ :\ '<' \mid 'in' \\
REOpsNoIn\ :\ '<' \\
BitANDExpIn\ :\ (BitANDExp\ REOpsIn) \\
BitANDExpNoIn\ :\ (BitANDExp\ REOpsNoIn)
\end{array}
$$

Last time we had to rearrange the pattern when we considered how to adapt it to the strict syntactic restriction. Now we are simply going to mimic the original abstraction above and see if it complies with the lenient restriction. First, let us

rewrite the rules using our notation:

$$
\begin{aligned}
&BitANDExp \;\rightarrow\; BOp(\text{'\&'},\; BOp(\text{' == '},\; BOp(0,\; ShiftExp)))\\
&REOpsIn \;\rightarrow\; \text{' < '} \mid \text{'}in\text{'}\\
&REOpsNoIn \;\rightarrow\; \text{' < '}\\
&BitANDExpIn \;\rightarrow\; BitANDExp(REOpsIn)\\
&BitANDExpNoIn \;\rightarrow\; BitANDExp(REOpsNoIn)
\end{aligned}
$$

Then we take the corresponding presence graph for this portion of grammar:



A brief glance at the graph tells us there are no cycles in it. It immediately follows that this piece of macrogrammar is in lenient restricted form.

Unlike the strict restriction, which did not allow us to fully generalize the pattern, the lenient restriction is flexible enough as to not sacrifice expressiveness.

□

Example 6.3.3 (Permutation Phrases) Once again, we recall the pattern taken

from the original grammar:

$$
\begin{aligned}
FieldModifiers \quad & : FieldModifier \\
& \mid FieldModifiers\ FieldModifier \\
FieldModifier \quad & :\ \text{'}static\text{'} \\
& \mid \text{'}final\text{'} \\
& \mid \text{'}public\text{'}
\end{aligned}
$$

the original abstraction made by Thiemann and Neubauer:

$$
\begin{aligned}
(PermP3\ m1\ m2\ m3) \quad & :\ /*\, empty */ \\
& \mid m1\ (PermP3\ NUL\ m2\ m3) \\
& \mid m2\ (PermP3\ m1\ NUL\ m3) \\
& \mid m3\ (PermP3\ m1\ m2\ NUL) \\
/*\, no\ production\ for\ NUL *\!/ & \\
FieldModifiers \quad & :\ (PermP3\ \text{'}static\text{'}\ \text{'}final\text{'}\ \text{'}public\text{'})
\end{aligned}
$$

and our reinterpretation keeping the original semantics of the Java grammar:

$$
\begin{aligned}
PermP3 \rightarrow\ & 0 \\
& \mid 1 \\
& \mid 2 \\
& \mid 0\ PermP3(NUL,\ 1,\ 2) \\
& \mid 1\ PermP3(0,\ NUL,\ 2) \\
& \mid 2\ PermP3(0,\ 1,\ NUL)
\end{aligned}
$$

$$
\begin{aligned}
& /*\, no\ production\ for\ NUL *\!/ \\
& FieldModifiers \rightarrow PermP3(\text{'}static\text{'},\ \text{'}final\text{'},\ \text{'}public\text{'})
\end{aligned}
$$

As it happened with the operator precedence pattern, the set of productions in our notation is already in lenient restricted form. □

6.3.2 Conclusion

A strictly restricted macrogrammar is leniently restricted as well. Therefore the first and third cases remain the same. The improvement comes from the hand of the second case, the lenient restriction has permitted its rewriting in the most general way possible, thus not imposing any artificial rearrangement of the structure.

All three patterns can be now expressed the way it is desirable. A grammar designer could have written these three patterns using macros in her context-free grammar with no noticeable restriction on her part.

It is reasonable to believe that the lenient syntactic restriction is powerful enough to represent most patterns (if not all) present in context-free grammars employed in the construction of compilers. This is so because when we try to abstract a pattern already present in a context-free grammar, it is not likely that ever increasing parameters arise necessarily. Sure it is always possible to come up with such problematic abstraction, but it would most probably be some contrived, artificial design.

On the other hand, a bold and imaginative designer may try to use macros to syntactically capture more than it is possible with a context-free grammar. Things that in present day compilers are left to the semantic analyzer or not implemented at all . A good example of this is the retrieval-language. Fischer showed in [Fis68] that we can use the retrieval-language to syntactically check variable declarations. That is, it allows us to capture in the syntax whether a given variable has been previously declared in a declaration block or similar. However, the retrieval-language is not context-free and thus any attempt of using macros to write it can never be complying with the lenient syntactic restriction.

6.4 Comparative with Thiemann's and Neubauer's work

As we have seen, we have aimed at providing a sufficient condition for context-independence in macrogrammars. In [TN04, TN08], Thiemann and Neubauer tried to accomplish this same objective too. However, in chapter 3, we saw how their work presented some issues. With this thesis, I have tried to provide a simpler, syntactical approach.

A syntactical approach, if possible, is always more desirable than other options.

This is so because it provides the designer with insight as to how to write her grammar. She is forced to think before implementing, thus avoiding context-sensitive macrogrammars beforehand. Furthermore, a syntactic condition can be easily implemented into an editor, so in case she makes a mistake when writing the grammar, her attention can be immediately drawn to it, thus not dragging along the mistake.

In comparison, a semantical approach is cumbersome. A semantical condition has to "interpret" the macrogrammar (hence its name). If such interpretation is departed from the original macrogrammar, then it is not possible for the designer to know at a certain point if what she has done so far is correct; this is the case with Thiemann's and Neubauer's condition. It could still be possible to implement a condition checker into an editor, but it should prove to be much more expensive and computing intensive than a syntactic condition checker. Besides, being the analysis departed from the original structure, it would be necessary to carry along the relation between the original macrogrammar and the analysis in order to provide the designer with information on how to correct her mistakes.

In this thesis, we have actually discussed not one, but two syntactic restrictions. It is possible to implement an algorithm which works in two stages, such that the first restriction is applied up to the point the designer writes a production not in strict form. From that point on, the second restriction would be applied.

The strict restriction can be checked at the same time the designer writes the production; that is the fastest a verification can get. The second restriction requires the construction of a presence graph or an equivalent data structure. Unfortunately, we have to search for cycles in such structure, but let us compare this with the following remark from Thiemann's and Neubauer's: "The situation is not as simple as just checking the instantiation graph for cycles. A cyclic instantiation graph is only a necessary condition, but it is not sufficient because partial evaluation already terminates if the underlying transition system is quasiterminating." They also use graphs to test their condition, in fact they use five different graphs, so they are bound to check for cycles too, though they do not provide in their paper the concrete specifications of their implementation. Moreover, cycles are involved in their condition, as stated by the remark, but their analysis is much more sophisticated. Once again, our approach turns out simpler and, most likely, more efficient.

But there must be a trade-off hidden somewhere. Thiemann's and Neubauer's analysis must be so complex for some reason, otherwise they could have just as well focused their efforts on a syntactical approach. Actually, their analysis can provide a more accurate characterization of the macrogrammars. Both the strict syntactic restriction and the lenient syntactic restriction ensure a macrogrammar generates a finite amount of instances and therefore remains within the context-free languages boundaries. However, a macrogrammar can generate a finite set of instances and not comply with either of the restrictions. This cannot happen in their analysis, if a macrogrammar does not fulfill their condition, then it generates an infinite amount of instances.

The following macrogrammar serves as example of a macrogrammar not in lenient restricted form and which generates a finite amount of instances:

$S \rightarrow Byblos(Timeo,\ Timeo\ hominem,\ Timeo\ hominem\ unius,$
$Timeo\ hominem\ unius\ libri)$
$Byblos \rightarrow Byblos(Timeo,\ 0\ hominem,\ 1\ unius,\ 2\ libri)\ 3\ |\ 3$

It is easy to see (we skip the presence graph for this example) that this macrogrammar is not in lenient restricted form, because the *Byblos* macro recurs over itself and the recurring rule contains terminals concatenated to formal parameters. But the instances generated are only S and $Byblos(Timeo,\ Timeo\ hominem,$ $Timeo\ hominem\ unius,\ Timeo\ hominem\ unius\ libri)$. Of course, the generated language is context-free:

$Timeo\ hominem\ unius\ libri\ (Timeo\ hominem\ unius\ libri)^*$

So Thiemann's and Neubauer's analysis is more powerful. But maybe the difference is small in practical terms. In the previous section we studied in detail three different patterns present in real grammars. We saw how we can use the lenient restriction to capture them without being forced to sacrifice abstraction. The same holds for other patterns I have considered. We have yet to find a pattern which cannot be directly expressed in lenient restriction form. Moreover, if we found such a structure, we could still generalize most of it, albeit sacrificing part of our expressiveness.

Summarizing: we have managed to obtain in this work a syntactic restriction which can be easily transformed into an implementation with a minimum overload over the

original macrogrammar specification (just the presence graph); we are not aware of any context-free pattern which needs to be expressed in a special way, losing abstraction. In contrast, Thiemann's and Neubauer's criterion is semantic and therefore its implementation should prove to be less efficient; their condition is not close to the macrogrammar specification, therefore adding extra overloading in order to trace back and give the designer information on the results; any macrogrammar with no unlimited macros can be expressed using their condition, no exceptions.

Chapter 7
Conclusions and future work

7.1 Conclusions

GONF formalism is more expressive than BNF, it allows writing grammars which generate context-sensitive languages. GONF needs a formalization using macro-grammars (macrogrammars). Macro-grammars have already been studied.

Thiemann and Neubauer tried to provide a means to check whether a macro-grammar can be transformed into an equivalent context-free grammar. Their results are not rigorous enough and the semantic analysis they present is cumbersome. They do not provide a practical means to implement their analysis.

We have supplied macro-grammars with a new set of formal definitions. From these, we have found a new sufficient condition for context independence. A macro-grammar satisfying it can always be specialized into a normal BNF grammar. Based on this condition too, two syntactic restrictions are defined, one being more lenient and including the other. These syntax restrictions keep macro-grammars within the context-free class. They are simple enough for a grammar designer to keep them in mind when writing the grammar, thus actively avoiding structures leading to context sensitivity. These restrictions can be efficiently checked too, even integrated into an editor, allowing for an early detection of errors.

We have seen how to express patterns taken from real Java and JavaScript grammars using the syntactic restrictions. In case of the lenient restriction, all patterns could be expressed using a natural abstraction, with no limitations. We are led to think it is hard to find patterns in actually used context-free grammars which need for a special rewriting under the lenient syntactic restriction. Even if so, we can always gain some abstraction. In any case, we can always reduce the size of the original grammar and increase its reusability and partitioning into modules.

We have also proved correct the result obtained in [Pér08] concerning macro-grammars. It is a particular case of the conditions obtained in this work.

7.2 Future lines of work

This thesis has introduced two practical syntactic restrictions for macro-grammars. Macros reduce the size and increase the readability of grammars. An automatic parser generator willing to support macros can implement these restrictions to achieve it. Besides, these syntactic restrictions are based on a semantic condition, this semantic condition may be used to search for new practical restrictions.

GONF needs a formalization stronger than the one in [HN05]. The formal frame in this thesis can be such formalization or at least serve as a starting point. This is necessary in order to apply this thesis' results to GONF. Such translation should prove to be simple. Once achieved, MTP can support macros.

We can regard grammars as term rewriting systems (RTS)[BN98]. It would be interesting to use RTS to study macro-grammars. Macro-grammars are simple RTS, maybe left-linear and non-overlapping, and such RTS have quite a few properties. A possible line of work would be to study whether those properties tell us something more about how to characterize macro-grammars.

The mathematical induction in the proof of theorem 4.2.2 is a rough specialization algorithm to obtain a context-free grammar from a macro-grammar. This method needs refining. As we saw previously, parsing for LR and LL grammars can be made in linear time [ASU86]. It can also be interesting to devise a specialization algorithm which directly restricts the generated grammar to one of these classes.

Bibliography

[Aho68] A. V. Aho. Indexed grammars–an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, 1968.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bla00] P. A. Blampied. *Structured recursion for non-uniform data-types*. PhD thesis, University of Nottingham, 2000.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1998.

[Cha84] N. P. Chapman. Lalr(1,1) parser generation for regular right part grammars. *Acta Informatica*, 21:29–45, 1984.

[Fis68] M. J. Fischer. Grammars with macro-like productions. In *9th Annual Symposium on Switching and Automata Theory*, pages 131–142, 1968.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[Grä79] G. Grätzer. *Universal Algebra*. Springer, New York, second edition, 1979.

[HN05] A. Herranz and P. Nogueira. More than parsing. In *Spanish Conference on Programming and Languages*, pages 193–202. Thomson Paraninfo, 2005.

[Int99] ECMA International. Ecmascript language specification http://www.ecma-international.org/publications/files/ecma-st/ecma-262.pdf, December 1999. ECMA-262, 3rd edition.

[Kos91] Kai Koskimies. Object-orientation in attribute grammars. In *Attribute Grammars, Applications and Systems*, pages 297–329, 1991.

[Pér08] Iván Pérez. Formalización de gramáticas paramétricas independientes de contexto e implementación en mtp. Graduate thesis, 2008.

[PQ94] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[TN04] P. Thiemann and M. Neubauer. Parameterized lr parsing. *Electr. Notes Theor. Comput. Sci.*, 110:115–132, 2004.

[TN08] P. Thiemann and M. Neubauer. Macros for context-free grammars. In *Principles and Practice of Declarative Programming*, pages 120–130, 2008.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.