NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Transforming Boolean grammars to Datalog

Dimitrios T. Konstantinidis

Supervisors: **Panagiotis Rondogiannis,** Professor
Angelos Charalambidis, Assistant Professor

ATHENS

January 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετατροπή Boolean γραμματικών σε Datalog

Δημήτριος Θ. Κωνσταντινίδης

Επιβλέποντες: **Παναγιώτης Ροντογιάννης,** Καθηγητής
Άγγελος Χαραλαμπίδης, Επίκουρος Καθηγητής

ΑΘΗΝΑ

Ιανουάριος 2024

BSc THESIS

Transforming Boolean grammars to Datalog

Dimitrios T. Konstantinidis
S.N.: 1115201700065

SUPERVISORS: **Panagiotis Rondogiannis,** Professor
Angelos Charalambidis, Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετατροπή Boolean γραμματικών σε Datalog

Δημήτριος Θ. Κωνσταντινίδης
Α.Μ.: 1115201700065

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Παναγιώτης Ροντογιάννης,** Καθηγητής
Άγγελος Χαραλαμπίδης, Επίκουρος Καθηγητής

ABSTRACT

Context-free grammars (CFGs) are a major field in programming, and their use cases, especially in compilers, are a cornerstone of information technology. In contrast, logic programming, considering its equivalence to any programming language, is an underrated and generally under-represented area of programming. Grammars and logic programming are two very different constructs but, in retrospect, they seem similar in their syntax. Currently in the scientific world, a transformation of context-free grammars to logic programs has been introduced that examines the nuanced relationship between these two. The extensions of CFGs proposed by Okhotin, conjunctive grammars which add the conjunction operation to CFGs, and Boolean grammars which extend conjunctive grammars with negation, are a significant step in formal language theory. In this thesis, a natural extension of the aforementioned transformation is proposed so as to handle conjunctive and Boolean grammars. A clear mathematical definition of the transformation is introduced that is implemented through a simple CFG parser which produces an Abstract Syntax Tree. In the nodes of the tree which correspond to grammar rules, the transformation is applied and a Datalog clause is produced. The end product is a Datalog program which possibly contains negation. For that reason, the well-founded semantics for negation are used to determine the existence of a string in the language created by the provided grammar. The transformation that we propose enhances the bridging between grammars and logic programming, and is an interesting extension which adds to our intuition into the similarities of these two worlds.

SUBJECT AREA: Formal Language Theory

KEYWORDS: Context-free grammars, Conjunctive grammars, Boolean grammars, Datalog

ΠΕΡΙΛΗΨΗ

Οι γραμματικές χωρίς συμφραζόμενα (CFGs) αποτελούν σημαντικό πεδίο στην πληροφορική και η χρήση τους, ειδικά σε μεταγλωττιστές, αποτελεί ακρογωνιαίο λίθο των τεχνολογιών πληροφορίας. Αντίθετα, ο λογικός προγραμματισμός, λαμβάνοντας υπόψη την ισοδυναμία του με οποιαδήποτε γλώσσα προγραμματισμού, είναι ένας υποτιμημένος και γενικά υποεκπροσωπούμενος τομέας προγραμματισμού. Οι γραμματικές και ο λογικός προγραμματισμός είναι δύο πολύ διαφορετικές κατασκευές, όμως, φαίνονται όμοιες στο συντακτικό τους. Στον επιστημονικό κόσμο, έχει εισαχθεί μια μετατροπή των CFGs σε ένα λογικό πρόγραμμα, η οποία εξετάζει την περίπλοκη σχέση μεταξύ αυτών των δύο. Οι επεκτάσεις των CFGs που προτείνονται από τον Okhotin, οι συζευκτικές γραμματικές που προσθέτουν την πράξη της σύζευξης στις CFGs και οι γραμματικές Boolean που επεκτείνουν τις συζευκτικές γραμματικές με την πράξη της άρνησης, είναι ένα σημαντικό βήμα στην θεωρία τυπικών γλωσσών. Σε αυτή τη πτυχιακή, προτείνεται μια επέκταση της προαναφερθείσας μετατροπής ώστε να χειρίζεται συζευκτικές και Boolean γραμματικές. Εισάγεται ένας σαφής μαθηματικός ορισμός της μετατροπής, ο οποίος υλοποιείται μέσω ενός απλού CFG parser που παράγει ένα Αφηρημένο Συντακτικό Δέντρο. Στους κόμβους του δέντρου που αντιστοιχούν σε κανόνες γραμματικής, εφαρμόζεται η μετατροπή και παράγεται ένας κανόνας Datalog. Το τελικό προϊόν είναι ένα πρόγραμμα Datalog που πιθανώς περιέχει άρνηση. Για το λόγο αυτό, η καλώς θεμελιωμένη σημασιολογία για την άρνηση χρησιμοποιούνται για να προσδιοριστεί, εν τέλει, η ύπαρξη μιας συμβολοσειράς στη γλώσσα που δημιουργήθηκε από την παρεχόμενη γραμματική. Η μετατροπή που προτείνουμε ενισχύει τη γεφύρωση μεταξύ γραμματικών και λογικού προγραμματισμού και είναι μια ενδιαφέρουσα επέκταση που ενισχύει τη διαίσθησή μας για τις ομοιότητες αυτών των δύο κόσμων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Θεωρία Τυπικών Γλωσσών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Γραμματικές χωρίς συμφραζόμενα, Συζευκτικές γραμματικές, Boolean γραμματικές, Datalog

Στην μητέρα μου

ACKNOWLEDGEMENTS

I would like to thank deeply my supervisors, Professor Panagiotis Rondogiannis and Assistant Professor Angelos Charalambidis for their guidance and assistance. In the past year they have shown me a hidden brilliant world and I am forever indebted to them.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis was implemented during my bachelor studies in the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens.

1. INTRODUCTION

1.1 Formal Grammars

Formal grammars are fundamental constructs within the realm of formal language theory, providing a systematic and mathematical framework for describing the syntax of languages. Developed by linguist Noam Chomsky in the 1950s, formal grammars are rule-based systems that define the structure and composition of strings in a language. They consist of a set of production rules that dictate how symbols, both terminal and non-terminal, can be combined to generate valid strings conforming to the language's syntax. These grammars are classified into various types, such as regular grammars, context-free grammars, and context-sensitive grammars, each possessing different expressive power. Formal grammars find extensive applications in computer science, especially in the design and analysis of programming languages, compilers, and natural language processing systems. They serve as the backbone for understanding the syntactic structures of languages and play a pivotal role in parsing and language recognition algorithms.

1.2 Logic Programming

Logic programming is a paradigm in computer science that revolves around the use of mathematical logic for expressing and executing programs. At its core is the programming language Prolog (and its subset Datalog), where computation is framed as a set of logical relationships and rules. In logic programming, a set of facts and rules is defined by using a formal logic known as Horn clauses. The program's execution involves the automated resolution of queries against these rules and facts, allowing for a declarative style of programming where the emphasis is on specifying what needs to be achieved rather than how to achieve it. Prolog and Datalog, with their inference engines, excel in tasks that involve symbolic reasoning, knowledge representation, and rule-based systems. Logic programming has found applications in various domains, including artificial intelligence, natural language processing, and expert systems, showcasing its versatility in tackling problems characterized by complex relationships and inferential reasoning.

1.3 Converting Grammars to Logic Programs

At a first glance, the aforementioned areas seem very distant from each other. Prolog and logic programming in general have a large amount of use cases as they are equivalent to a Turing machine. In essence, Prolog has the same abilities as any imperative programming language like C and Java. On the other hand, formal grammars are tools for expressing languages and their usage is small in other fields of computer science. Their limited expressibility hinders them from solving a large amount of problems and makes them inherently ”weaker” than programming languages. However, intuitively the syntax of both seems very similar. A simple example rule of a formal grammar is the following statement:

$$S \to A$$

Whereas a simple logic clause is:

$$p \leftarrow q$$

This similarity can be further explored by defining a conversion between these two. Although converting logic programs into formal grammars is impossible as there are problems which cannot be solved by any kind of grammar (with the exception of general grammars which are out of the scope of this thesis), transforming a formal grammar into a logic program is something that has been introduced in the scientific world [7]. In this thesis we enhance this transformation to accept two supersets of CFGs, conjunctive and Boolean grammars. These grammars serve as natural extensions of CFGs as they add the conjunction and negation operations respectively to the grammar rules.

2. BACKGROUND AND RELATED WORK

2.1 Context-Free Grammars

2.1.1 Introduction

Context-free grammars (CFGs) are a fundamental concept in formal language theory and computer science. They provide a robust mathematical framework for precisely describing the syntax of programming languages, natural languages, and many other formal languages. At the core of their significance lies the ability to reliably define the hierarchical structure of languages. Unlike regular expressions, context-free grammars introduce non-terminals, symbols that serve as placeholders to be replaced by sequences of symbols, providing a level of abstraction essential for representing complex syntactic structures. This inherent flexibility allows CFGs to encapsulate the syntax of a broad spectrum of languages, ranging from the rigid rules found in programming languages to the nuanced intricacies of natural languages.

2.1.2 Definition

As defined in [6], a context-free grammar is a quadruple $G = (\Sigma, N, P, S)$ where:

- Σ is a finite non-empty set of symbols that form the strings of the language produced by the grammar, we call this alphabet “terminals”.
- N is a finite non-empty set of variables or “non-terminals”, each of these represents a language (i.e. a set of strings).
- P is a finite set of rules, each defined as an ordered pair (A, B), where $A \in N$ and $B \in (\Sigma \cup N)^*$, that represent the definition of the language produced by G. Each rule is written as:

$$A \to B$$

- $S \in N$ is the starting symbol that defines the language produced by G.

2.1.3 Examples

Consider a context-free grammar for the language $L = \{wcw^R : w \in \{a, b\}^*\}$. This language consists of palindrome strings with c as the middle point surrounded by the characters a and b.

$$S \to aSa$$
$$S \to bSb$$
$$S \to c$$

In the above example: $\Sigma = \{a, b, c\}$, $N = \{S\}$, $P = \{(S, aSa), (S, bSb), (S, c)\}$ and S is the starting symbol.

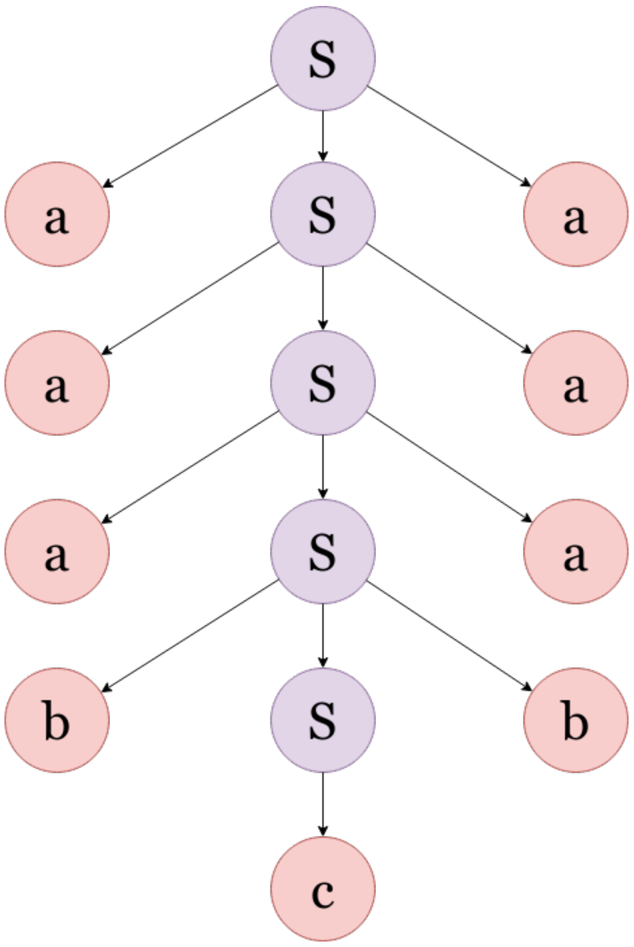

Figure 2.1: CFG Syntax Tree

2.1.4 Derivations and Syntax Trees

The elegance of context-free grammars reveals itself in their ability to articulate the step-by-step process of string generation through derivations. Each derivation, an application of the production rules P, unveils the evolution of a string from its non-terminal form to the final composition of terminals. Complementing derivations, syntax trees (often mentioned as ”parse trees”) provide a visual representation of the syntactic hierarchy encoded within a string, offering insights into the structural relationships between its constituent elements.

For instance, in the aforementioned grammar, the string $aabcbaa$ can be derived from the starting symbol S with the following derivation:

1. aSa (from rule 1.$S \to aSa)$
2. $aaSaa$ (from rule 1.$S \to aSa)$
3. $aabSbaa$ (from rule 2.$S \to bSb)$
4. $aabcbaa$ (from rule 1.$S \to c)$

The corresponding syntax tree is shown in Figure 2.1:

2.1.5 Chomsky Normal Form

In formal language theory, a context-free grammar, G, is said to be in Chomsky normal form (first described by Noam Chomsky)[3] if all of its production rules are of the form:

$$A \to BC$$
$$A \to a$$
$$S \to \epsilon$$

where A, B are non-terminals, a is a terminal symbol, S is the starting symbol (thus a non-terminal as well) and ϵ denotes the empty string.

2.1.6 Parsing

The parsing problem, checking whether a given word belongs to the language provided by a context-free grammar, is decidable. The process of parsing involves navigating through the grammar's production rules, making decisions at each step based on the current input and the rules of the grammar. Successful parsing results in the construction of the syntax tree. There are different parsing techniques employed for context-free grammars, with the two primary methods being top-down parsing and bottom-up parsing.

In **top-down** parsing, the process starts from the top of the parse tree (the start symbol) and proceeds downward, recursively expanding non-terminals to match the input string. One of the most common top-down parsing methods is recursive descent parsing, where each non-terminal in the grammar corresponds to a parsing function. These functions are invoked recursively, and each function attempts to match a portion of the input string with the production rules associated with the corresponding non-terminal. The choice of which production to apply is typically determined by examining the next k tokens of input. However, not all grammars are eligible for this method as the problems of ambiguity and left recursion raise difficulties in the parsing process. The grammars without these issues belong in the $LL(k)$ class where k is the amount of tokens after the input that will be consumed. LL stands for ”Left-to-right, Leftmost derivation”. Recursive descent parsers are relatively easy to implement and understand, providing a clear correspondence between grammar rules and parsing functions. The grammar provided in the above examples is an $LL(1)$ grammar.

Bottom-up parsing is a technique employed to analyze the syntactic structure of an input string according to a context-free grammar. Unlike top-down parsing, bottom-up parsing starts from the input string and works upward, aiming to reduce portions of the input to non-terminals until the start symbol is reached. LR parsing (”Left-to-right, Rightmost derivation”), a popular variant of bottom-up parsing, employs a deterministic finite automaton (DFA) and a stack to guide the parsing process. The parser shifts input symbols onto the stack until a viable right-hand side of a grammar rule is identified, at which point a reduction operation is performed. This reduction replaces the right-hand side with the corresponding non-terminal, mimicking the construction of the parse tree in a bottom-up fashion. LR parsers are efficient and capable of handling a broad class of grammars, with $LR(1)$ and $LALR(1)$ (”Look-ahead LR”) being widely used variants. Although bottom-up parsers are not limited by the problems of ambiguity and left recursion, they are considerably harder to implement than top-down parsers.

Other common parsing algorithms for CFGs include:

- **Earley Parser**: Introduced in [5] and named after its creator Jay Earley, is a general parsing algorithm known for its versatility and ability to handle a broad class of grammars, including those with left recursion and ambiguity. Operating on the principles of dynamic programming, the Earley parser builds a chart or table to store partial parsing results. It processes the input string incrementally, maintaining a set of states that represent possible positions within the grammar. The algorithm progresses by predicting, scanning, completing, and combining states, ultimately producing a set of valid parse items. The Earley parser's strength lies in its flexibility, making it particularly suitable for natural language processing applications where grammars can be intricate and ambiguous. Despite its generality, its worst-case time complexity is $O(n^3)$, limiting its efficiency for extremely large inputs or grammars. Nonetheless, the Earley parser remains a valuable tool in the parsing toolkit, especially when dealing with grammars that other parsing algorithms may find challenging.
- **CYK**: Published in [13] and re-introduced in [4], operates by constructing a table to store partial parsing results. The table is filled by considering all possible combinations of non-terminals that can generate substrings of the input. This process of bottom-up table filling allows the CYK algorithm to efficiently identify valid parse trees for the input string. One notable feature is its versatility in handling ambiguous grammars. At this point, we must note that the grammar must be in CNF. The CYK algorithm has found applications in various fields, including natural language processing and computational linguistics, showcasing its significance in parsing tasks where the underlying grammatical structure is well-defined and suitable for CNF representation. The worst case performance of the algorithm is $O(n^3 \cdot |G|)$ where $|G|$ is the size of the CNF-grammar G.

2.1.7 CFGs in the Chomsky Hierarchy

The Chomsky Hierarchy [3], [2], proposed by linguist and cognitive scientist Noam Chomsky, is a classification system that categorizes formal grammars into distinct classes based on their generative power. This hierarchy comprises four levels: Type 3 (Regular), Type 2 (Context-Free), Type 1 (Context-Sensitive), and Type 0 (Unrestricted). At the bottom of the hierarchy are Type 3 grammars, or regular grammars, which are equivalent to finite automata. Moving up, Type 2 grammars, or context-free grammars, find extensive use in defining the syntax of programming languages. Type 1 grammars, or context-sensitive grammars, have rules that are sensitive to the context of the symbols, allowing for greater expressive power. At the pinnacle of the hierarchy are Type 0 grammars, or unrestricted grammars, which have no restrictions on their production rules. The Chomsky Hierarchy provides a theoretical framework to understand the relationships and limitations among different classes of formal languages, offering insights into the computational complexities inherent in language recognition and generation.

2.1.8 Pumping Lemma for Context-Free Languages

Introduced in [9], the Pumping Lemma for Context-Free Languages is a fundamental concept in formal language theory that provides a tool for demonstrating the non-context-

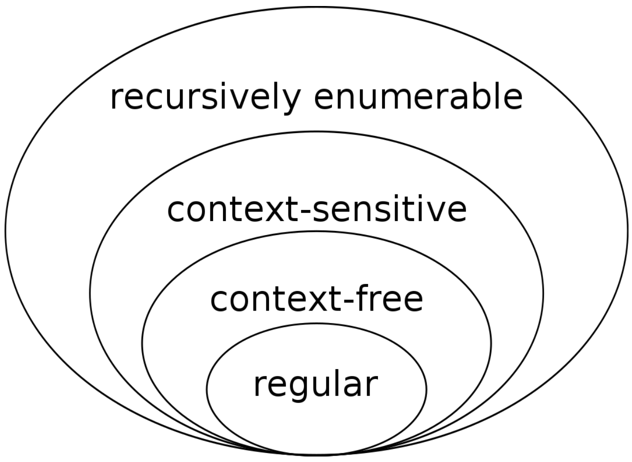

Figure 2.2: Chomsky Hierarchy

freeness of certain languages. It serves as a proof by contradiction: if a language violates the conditions, it cannot be context-free. This lemma provides a powerful tool for establishing the limitations of context-free grammars and identifying languages that fall outside this class. The lemma states that if a language L is context-free, then there exists some integer $p \geq 1$ such that every string $s \in L$ (where $|s| \geq p$) can be written as:

$$s = uvwxy$$

Where:

- $|vx| \geq 1$
- $|vwx| \leq p$
- $uv^n wx^n y \in L, \forall n \geq 0$

The integer p is called the “pumping length” of the language L.

Example: Consider the language $L = \{a^n b^n c^n : n \geq 0\}$. First we assume that L is context-free. The string $s = a^p b^p c^p$ can be written as $uvwxy$ such that $|vx| \geq 1$ and $|vwx| \leq p$ where $p \geq 1$ and $p \in \mathbb{N}$.

For vwx there are 5 possibilities:

- $vwx = a^i, i \leq p$
- $vwx = b^i, i \leq p$
- $vwx = c^i, i \leq p$
- $vwx = a^i b^j, i + j \leq p$
- $vwx = b^i c^j, i + j \leq p$

Note that $vwx = a^i b^j c^k, i + j + k \leq p$ is impossible as we have defined s as $a^p b^p c^p$ and $|vwx| \leq p$. For all the aforementioned possibilities, the string $uv^n wx^n y \notin L$, therefore -through proof by contradiction- the language L is not context-free.

In a similar way the language $L = \{wcw : w \in \{a, b\}^*\}$ can be proven to be not context-free.

2.1.9 Closure Properties

The closure properties of context-free grammars refer to the preservation of certain language properties under various operations. One fundamental closure property is closure under union, meaning that if two languages can be generated by CFGs, their union can also be generated by a CFG. This property highlights the composability of context-free languages. Another crucial closure property is closure under concatenation, asserting that the concatenation of two languages generated by CFGs is itself generated by a CFG. Additionally, context-free grammars exhibit closure under the Kleene star operation, implying that the repetition of a language generated by a CFG can also be generated by a CFG.

On the other hand, CFGs are not closed under conjunction and negation. Consider the languages $L_1 = \{a^n b^n c^m : n \geq 0, m \geq 0\}$ and $L_2 = \{a^m b^n c^n : n \geq 0, m \geq 0\}$. Both of these languages are context-free. Despite that, their intersection is the language $L = \{a^n b^n c^n : n \geq 0\}$ for which we proved its non-context-freeness. Hence the intersection of two or more context-free languages need not necessarily belong to the context-free family. The case is similar for the complement operation.

For that reason, we will be looking into two extensions of CFGs, conjunctive grammars and Boolean grammars. The first allows the conjunction/intersection operation (hence closed under intersection) and the latter allows the negation/complement operation (closed under complement).

2.1.10 Limitations and Extensions

While context-free grammars offer a potent framework for language description, they are not without limitations, as shown by the two languages provided above. Certain language constructs elude concise representation within traditional context-free grammars. Conjunctive grammars and Boolean grammars (both of which will be explored further in this thesis) emerge as extensions, overcoming some of these limitations and expanding the expressive power of formal language representation.

2.2 Conjunctive Grammars

2.2.1 Introduction

Conjunctive grammars are a type of formal grammar introduced by Alexander Okhotin in [10], primarily in the context of parsing and recognizing languages. Unlike traditional context-free grammars, which use rules to replace non-terminal symbols with strings of terminals and non-terminals, conjunctive grammars employ a more intricate set of rules. In a conjunctive grammar, productions involve the intersection of languages, allowing the combination of multiple languages simultaneously. This intersection operation introduces a new layer of expressiveness and complexity, making conjunctive grammars a powerful tool for describing a wide range of languages and structures.

The intersection operation enables the specification of conditions that involve multiple aspects of a string, allowing for more nuanced language definitions. This expressive capability makes conjunctive grammars particularly well-suited for applications where complex relationships between components of a language need to be articulated.

2.2.2 Definition

As defined in [10], a conjunctive grammar is a quadruple $G = (\Sigma, N, P, S)$ where:

- Σ is a finite non-empty set of symbols called “terminals”.
- N is a finite non-empty set called “non-terminals”.
- P is a finite set of rules, each defined as an ordered pair $(A, \{a_1, a_2, ..., a_n\})$, where $A \in N$, $a_i \in (\Sigma \cup N)^*$ and $1 \leq i \leq n$. Each rule is written as:

$$A \to a_1 \,\&\, a_2 \,\&\, ... \,\&\, a_n$$

- $S \in N$ is the starting symbol.

The following notation will be used for the rules of a single non-terminal:

$$A \to a_{11} \,\&\, ... \,\&\, a_{1n} \mid ... \mid a_{m1} \,\&\, ... \,\&\, a_{mk}$$

2.2.3 Examples

Consider the language $L = \{a^n b^n c^n : n \geq 0$. Earlier, we proved that this language is not context-free. However, the addition of the intersection operation allows us to express this language easily through the following conjunctive grammar.

$$S \to AB\&DC$$
$$A \to aA \mid \epsilon$$
$$B \to bBc \mid \epsilon$$
$$C \to cC \mid \epsilon$$
$$D \to aDb \mid \epsilon$$

$\Sigma = \{a, b, c\}$, $N = \{S, A, B, C, D\}$, $P = \{(S, \{AB, DC\}), (A, \{aA\}), (A, \{\epsilon\}), ...\}$ and S is the starting symbol.

Also, for another language that is not context-free, $L = \{wcw : w \in \{a, b\}^*\}$, a conjunctive grammar that produces it is the following.

$$S \to C\&D$$
$$C \to aCa \mid aCb \mid bCa \mid bCb \mid c$$
$$D \to aA\&aD \mid bB\&bD \mid cE$$
$$A \to aAa \mid aAb \mid bAa \mid bAb \mid cEa$$
$$B \to aBa \mid aBb \mid bBa \mid bBb \mid cEb$$
$$E \to aE \mid bE \mid \epsilon$$

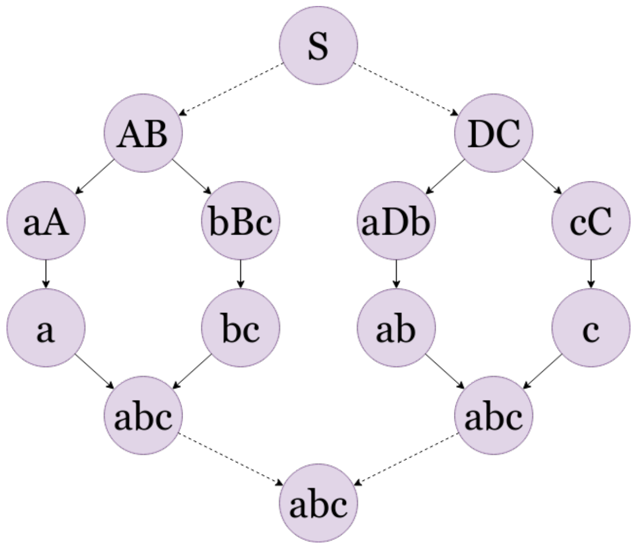

Figure 2.3: Conjunctive Grammar Syntax Tree

2.2.4 Derivations and Syntax Trees

The process of string generation through derivations is similar to CFGs. The only difference is that whenever an intersection is found, the derivation process splits into two parts. Assuming both reach the same result, then the process is a success. It is important to acknowledge that the branching parts can be computed in parallel.

For instance, for the grammar $L = \{a^n b^n c^n : n \geq 0\}$, the string abc can be derived from the starting symbol S with the following process:

1. $AB \,\&\, DC$
2. $(aA)B \,\&\, (aDb)C$
3. $(a())(bBc) \,\&\, (a()b)(cC)$
4. $(a)(b()c) \,\&\, (ab)(c())$
5. $a(bc) \,\&\, ab(c)$
6. $abc \,\&\, abc$
7. abc

The corresponding branching syntax tree can be seen in Figure 2.3

2.2.5 Binary Normal Form

Okhotin in [10] proposes a normal form that naturally extends Chomsky Normal Form [3] for the case of conjunctive grammars.

A conjunctive grammar $G = (\Sigma, N, P, S)$ is said to be in binary normal form, if each rule in P is in the form:

- $A \to B_1C_1 \,\&\, ... \,\&\, B_mC_m$, where $m \geq 1$ and $A, B_i, C_i \in N$.

- $A \to a$, where $A \in N$ and $a \in \Sigma$.
- $S \to \epsilon$ only if S does not appear in right parts (after the $\to$ symbol) of rules.

2.2.6 Parsing

In the same paper that they were introduced [10], Okhotin also proposes an algorithm for parsing conjunctive grammars. Assuming the grammar is in BNF, they can be parsed by a variation of the CYK algorithm. Like in the context-free case, a so-called recognition matrix is defined, an upper-triangular matrix of sets of nonterminals that derive the substrings of the input string. Once the string is recognized, all of its derivation trees may be constructed by analyzing the recognition matrix in a way similar to the context-free case.

The difference between the proposed algorithm and the original CYK algorithm is that in the case of conjunctive grammars one has to accumulate all the pairs from the different factorizations of the current substring, and only the full set of such pairs can be used to determine the membership of nonterminals.

It is important to note that the extension for conjunctive grammars does not increase the complexity of the algorithm.

2.2.7 Limitations and Extensions

Let us consider another language $L = \{ww : w \in \{a, b\}^*\}$. At a first glance, it seems almost identical to the grammar from Example 2, however in this case the center marker c is missing. The grammar from Example 2 essentially uses the center marker, and therefore this method cannot be applied to writing a conjunctive grammar for a language without the marker. So far, no conjunctive grammar that describes the language $L = \{ww : w \in \{a, b\}^*\}$ has been discovered yet. Also, no method to prove non-conjunctiveness of a given language has been developed. Although the aforementioned language is a big problem in the area of conjunctive grammars, Boolean grammars (which will be explored next) provide a simple and elegant solution for it.

2.2.8 Closure Properties

It is easily established that the family of conjunctive languages is closed under union, intersection, concatenation and Kleene star, because each of these operations is explicitly included in the grammar formalism. However, it remains an open problem whether the family of conjunctive languages is closed under complement. In contrast, Boolean grammars that we examine in the next section are closed under complement.

2.3 Boolean Grammars

2.3.1 Introduction

Boolean grammars are another type of formal grammar introduced by Alexander Okhotin in [11], primarily in the context of parsing and recognizing languages. They are a natural extension of conjunctive grammars as they allow the usage of the complement operation.

The complement operation enables another degree of expressiveness in the language, thus allowing us to express even more intricate languages, with more nuanced definitions. Also, the complement operation is the final (and the most nuanced) logical operation that can be applied to grammars. Hence, Boolean grammars can solve the aforementioned problems that CFGs and conjunctive grammars face.

However, the addition of the complement operation, introduces risks which arise from the existence of negation. Negation-as-failure has been a very troublesome concept in the area of programming. This can be seen by the massive difference between the semantics of positive and negative logic programming, which will be explored later in this thesis.

2.3.2 Definition

As defined in [11], a Boolean grammar is a quadruple $G = (\Sigma, N, P, S)$ where:

- Σ is a finite non-empty set of symbols called "terminals".
- N is a finite non-empty set called "non-terminals".
- P is a finite set of rules, each defined as an ordered triple $(A, \{a_1, a_2, ..., a_n\}, \{b_1, b_2, ..., b_m\})$, where $A \in N$, $a_i, b_j \in (\Sigma \cup N)^*$ and $0 \le i \le n, 0 \le j \le m$.Each rule is written as:

$$A \to a_1 \,\&\, ... \,\&\, a_n \,\&\, \neg b_1 \,\&\, ... \,\&\, \neg b_m$$

- $S \in N$ is the starting symbol.

The following notation will be used for the rules of a single non-terminal:

$$A \to a_{11} \,\&\, ... \,\&\, \neg b_{1n} \mid ... \mid a_{m1} \,\&\, ... \,\&\, \neg b_{mk}$$

2.3.3 Examples

Consider the language $L = \{ww : w \in \{a, b\}^*\}$ (for which no conjunctive grammar has been found yet). A Boolean grammar that produces this language is the following.

$$\begin{gathered} S \to \neg AB \& \neg BA \& C \\ A \to XAX \mid a \\ B \to XBX \mid b \\ C \to XXC \mid \epsilon \\ X \to a \mid b \end{gathered}$$

$\Sigma = \{a, b\}$, $N = \{S, A, B, C, X\}$, $P = \{(S, \{C\}\{AB, BA\}), (A, \{XAX\}, \emptyset), (A, \{a\}, \emptyset), ...\}$ and S is the starting symbol.

Also consider the language $L = a^{2^n}, n \in \mathbb{N}$. A Boolean grammar that produces it is the following.

$$S \to A \,\&\, \neg aA \mid aB \,\&\, \neg B \mid aC \,\&\, \neg C$$

$$A \to aBB$$
$$B \to \neg CC$$
$$C \to \neg EE$$
$$E \to \neg A$$

2.3.4 Semantics

In contrast to CFGs and conjunctive grammars, whose semantics are trivial and mostly self-explanatory, the existence of the complement operation complicates semantics for Boolean grammars. Consider the following statements.

$$P \to \neg Q$$
$$Q \to \neg P$$

In this Boolean grammar the existence of any string in the language it produces is unclear.

There have been many proposals, but the following three are the most important to cover:

Basic Semantics are the most simple attempt at producing meaning from Boolean grammars. Proposed intuitively by Okhotin in [11], the semantics claim that assuming a grammar $G = (\Sigma, N, P, S)$ which consists of the following rules for the non-terminal X:

- $X_1 \to a_{11} \,\&\, ... \,\&\, \neg b_{11} \,\&\, ...$
- $X_2 \to a_{21} \,\&\, ... \,\&\, \neg b_{21} \,\&\, ...$

  ...

- $X_m \to a_{m1} \,\&\, ... \,\&\, \neg b_{m1} \,\&\, ...$

The solution for a single rule X_k is $A_k \cap B_k$ where

$$A_k = \bigcap_i a_{ki}$$

$$B_k = \bigcap_j \neg b_{kj}$$

And the generalized equation for the whole non-terminal X is

$$X = \bigcup_k X_k$$

For the simple grammar:

$$S \to A \,\&\, \neg B$$
$$A \to aaA \mid aa$$
$$B \to aaaaB \mid aaaa$$

The basic semantics for S would be:

$$S = A \cap \neg B$$

where:

$$A = aaA \cup aa$$

$$B = aaaaB \cup aaaa$$

The above grammar produces the language $L = \{a^{2n}, n > 0, n\%2 = 1\}$.

This simplistic definition faces severe problems on specific grammars which by intuition should produce the same language but their solution produced by these semantics is different.

Consider the following grammars (with $\Sigma = \{0, 1\}$) G_1 and G_2 respectively:

$$A \rightarrow \neg A \,\&\, \neg B$$

$$B \rightarrow 0 \,\&\, 1$$

$$A \rightarrow \neg A \,\&\, \neg B$$

$$B \rightarrow 0 \,\&\, 1 \mid B$$

Both of the language these grammars produce seem identical. However, when the basic semantics are applied to them, the grammar G_1 has no solution, whereas the solution of G_2 is $(A, B) = (\emptyset, \Sigma^*)$. This is only due to the addition of the rule $B \rightarrow B$.

Naturally Reachable Solution Semantics are the second proposal of Okhotin in [11] aiming to tackle the weaknesses of the previous semantics. Covering them in full is out of the scope of this thesis but, in short, consider a sequence of interpretations. Each interpretation "builds" the solution from the bottom-up from each rule. If the sequence converges in finite steps we call it a naturally reachable solution. However, the NRS semantics are also problematic, especially in the case where a negation circle exists in the grammar. Consider the following grammar G:

$$A \rightarrow \neg B \mid D$$

$$B \rightarrow \neg C \mid D$$

$$C \rightarrow \neg A \mid D$$

$$D \rightarrow aD \mid \epsilon$$

Despite the fact that the solution for G is clearly

$$(A, B, C, D) = (a^*, a^*, a^*, a^*)$$

The NRS semantics produce an *undefined* solution as they never converge.

Well-Founded Semantics for Boolean Grammars, which were proposed in [8], successfully solve these problems through the inclusion of three-valued logic.

Let Σ be an alphabet (a finite non-empty set of symbols), a three-valued language over Σ is a function from Σ^* to the set $\{0, \frac{1}{2}, 1\}$

The value $\frac{1}{2}$ indicates undecidability over the existence of a given string in a language. Consider the aforementioned example where:

$$P \to \neg Q$$
$$Q \to \neg P$$

The two rules produce two languages. The existence of any string in either P or Q is undecidable as they both exist in a negation circle. For that reason, both P and Q are evaluated to produce the language that answers $\frac{1}{2}$ over any queried string.

Compared to other proposed semantics, the well-founded semantics allow us to evaluate grammars which contain negation circles. In the previous example (where the NRS semantics failed to produce a solution):

$$A \to \neg B \mid D$$
$$B \to \neg C \mid D$$
$$C \to \neg A \mid D$$
$$D \to aD \mid \epsilon$$

Regarding the rules $A \to \neg B$, $B \to \neg C$ and $C \to \neg A$ the WF semantics decide that the language they produce is $L = \frac{1}{2}$. This allows us to focus on the rules $A \to D$, $B \to D$ and $C \to D$. The non-terminal D clearly produces the language Σ^* (where $\Sigma = \{a\}$). Thus, through the WF semantics, the evaluation of the non-terminals (A, B, C, D) is that they produce the language $L = \Sigma^*$ which is the most intuitive solution.

2.3.5 Binary Normal Form

The essence of Chomsky Normal Form for context-free grammars and Binary Normal Form for conjunctive grammars is extended for Boolean grammars as well. In the paper where WF semantics were defined [8], a definition for a Binary Normal Form for Boolean grammars is provided:

A Boolean grammar $G = (\Sigma, N \cup \{U, T\}, P, S)$ is said to be in binary normal form if P contains the rules $U \to \neg U$ and $T \to \neg\epsilon$, where U and T are two special symbols not in N, and every other rule in P is of the form:

- $A \to B_1C_1 \,\&\, ... \,\&\, B_mC_m \,\&\, \neg D_1E_1 \,\&\, ... \,\&\, \neg D_nE_n \,\&\, TT[\,\&\, U]$
- $A \to a[\,\&\, U]$
- $S \to \epsilon[\,\&\, U]$, only if S does not appear in right-hand sides of rules (after the $\to$ symbol)

2.3.6 Parsing

Regarding parsing we shall only look into algorithms for WF semantics. In the same paper that WF semantics were introduced [8], an algorithm is given for the membership of a string in the language defined by a grammar G (which is in Binary Normal Form). The inner workings of this algorithm are out of the scope of this thesis, but, in short, it is an algorithm which evaluates at first simple rules such as $A \to a$. Based on these simple evaluations, the information for the more complex rules is built in a bottom-up manner. The algorithm runs in $O(n^3)$ time and the correctness of the algorithm is also proved in the paper.

2.3.7 Closure Properties

As we have established, Boolean grammars naturally extend conjunctive grammars, for which we know that they are closed under *union*, *concatenation*, *Kleene Star* and *intersection* but not under *negation* (for which we are unaware). In this case, they inherit the closure properties of conjunctive grammars while also being closed under complement as the existence of negation exists in their core formalism. Thus, Boolean grammars are closed under *union*, *concatenation*, *Kleene Star*, *intersection* and *negation*.

2.4 Datalog

2.4.1 Introduction

Datalog is a declarative query language designed for expressing and querying deductive databases. Developed in the 1970s, Datalog has its roots in Prolog and logic programming. The language is specifically tailored for expressing rules and queries over sets of facts and rules, making it particularly suitable for knowledge representation and manipulation. Datalog is often used as a query language for deductive databases and it has been applied to problems in data integration, networking, program analysis, and more.

The fundamental building blocks of Datalog are facts, rules, and queries. Facts represent ground truths about the domain, rules define relationships and derivations based on existing facts, and queries seek information by asking questions about the data. Datalog rules adhere to Horn clause form, consisting of a head (conclusion) and a body (premise) connected by implication. The rules are used to infer new facts from existing ones, creating a transitive closure that allows for the exploration of complex relationships within the data.

2.4.2 Examples

Example of facts in Datalog (statements that are held to be true):

```
parent(josephine, eugene).
parent(eugene, carolina).
```

These two statements hold the information that Josephine is the parent of Eugene and Eugene is the parent of Carolina.

Example of rules in Datalog (constructs to deduce new facts from known facts)

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

The meaning of a program is defined to be the set of all of the facts that can be deduced using the initial facts and the rules. This program's meaning is given by the following facts:

```
parent(josephine, eugene).
parent(eugene, carolina).
ancestor(josephine, eugene).
ancestor(eugene, carolina).
ancestor(josephine, carolina).
```

When the following query is asked:

```
? - ancestor(josephine, X).
```

X will be matched with eugene and carolina.

2.4.3 Differences with Prolog

Prolog and Datalog share a common ancestry and are both declarative programming languages used for expressing and querying logical relationships. However, there are notable differences between the two. Prolog is a general-purpose programming language with a broader scope, allowing for the definition of procedures and the execution of algorithms beyond database querying. In contrast, Datalog is a specialized query language specifically designed for deductive databases. While Prolog supports backtracking and the ability to express procedural logic, Datalog focuses primarily on expressing rules and queries in a more restricted form. Datalog's syntax is often more concise and tailored for expressing relationships in databases, making it particularly suited for applications in knowledge representation and database querying. Specifically, Datalog does not allow the usage of compound terms. The following clause would be invalid in Datalog:

```
p(s(x)) :- p(s(s(x)).
```

The specialized nature of Datalog makes it more efficient for certain types of tasks related to querying and manipulating data, whereas Prolog's versatility extends to a wider range of programming applications. The benefits that Prolog offers are not needed for the task of this thesis, hence Datalog will be used.

2.4.4 Semantics of Datalog

The bottom-up evaluation, or bottom-up computation, in Datalog is a key aspect of its deductive reasoning process. In Datalog, the evaluation strategy is often referred to as ”bottom-up” because it starts with the known facts (base or ground facts) and then applies rules to iteratively derive new facts until no more derivations are possible. This approach is also known as fixpoint computation.

During bottom-up evaluation, the system begins with the initial set of facts stored in the database. It then repeatedly applies the rules of the Datalog program to derive new facts. Each iteration adds more derived facts to the set, contributing to a transitive closure of the logical relationships defined by the rules. The process continues until reaching a fixpoint, where no further derivations are possible, and the system stabilizes.

Bottom-up evaluation in Datalog takes advantage of monotonicity, a key property of the language. Monotonicity ensures that the addition of new rules or facts can only lead to the derivation of more true facts without invalidating existing ones. This property simplifies the reasoning process and contributes to the efficiency of bottom-up evaluation in Datalog.

2.4.5 Semantics for Negation

As was the case in Boolean grammars, the existence of negation complicates matters in the same way. The aforementioned 'bottom-up' evaluation lacks the ability to fully grasp the difficulties of negation-as-failure. There have been many proposals but we will delve only into the Well-Founded Semantics.

Well Founded Semantics for Logic Programs were defined in [15] and [14]. In Prolog/Datalog, programs often involve rules and recursion, which can lead to cyclic dependencies or the potential for contradictory conclusions. Other semantics that have been proposed like Stratified [12] or Locally Stratified [1] succesfully handle negation but they have an inherent limit to their expressiveness. Specifically, negation circles are not evaluated at all. The well-founded semantics introduces the concept of a "well-founded model" to address these issues. The idea is to construct a consistent and minimal interpretation of the program by avoiding the introduction of contradictory loops.

One key element of the well-founded semantics is the introduction of the concept of "unfounded sets." These sets identify situations where cyclic dependencies or contradictions may arise. By carefully handling these cases, the well-founded semantics provides a more reliable and intuitive interpretation of Prolog programs, ensuring that the reasoning process remains sound and coherent even in the presence of complex rules and recursion.

Consider the following program:

```
p :- not q.
q :- not p.
```

Stratified and locally stratified semantics would fail -by design- to evaluate the values of both p and q. On the other hand, WF semantics (as is the case for WF semantics in Boolean grammars) use 3-valued logic. They introduce the $\frac{1}{2}$ value -indicating uncertainty- alongside the usual *true* and *false* values. In this case both *p* and *q* are evaluated as *unknown*, hence they are assigned the value of $\frac{1}{2}$.

In general, as logic programs are parsed and evaluated, each fact is given a value which is determined by other facts in the program. Starting off with facts such as:

```
t.
```

The term *t* would be assigned the value of $true$. Through that, more and more complex clauses are evaluated. Negation works in the same way when no circles exist. If the following clause was added to the program:

```
f :- not t.
```

The value of f would resolve to $false$.

Eventually, all terms **converge** to either $true$ or $false$ except from the ones that contain/are contained in a negation circle. The evaluation of these **diverges**, thus they are assigned the value of $\frac{1}{2}$. However, evaluation does not stop there. Terms who contain *undefined* facts still need to be assigned a value and that value is not necessarily $\frac{1}{2}$. The value for p in the following program:

```
p.
p :- not p.
```

Should be $true$. This is done in accordance with the following truth tables for 3-valued logic.

Intersection

&	$true$	$\frac{1}{2}$	$false$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$false$

Union

|	$true$	$\frac{1}{2}$	$false$
$\frac{1}{2}$	$true$	$\frac{1}{2}$	$\frac{1}{2}$

While for negation: $\neg\frac{1}{2} = \frac{1}{2}$.

As a final example, consider the following program:

```
t.
k :- not p, t.
p :- not q.
q :- not z.
z :- not p.
```

The terms p, q and z would be evaluated to $\frac{1}{2}$. The term t is *true* and $k = (\neg\frac{1}{2})\ \&\ true = \frac{1}{2}$

2.5 Related Work

In the current scientific world, a transformation of CFGs to datalog has been introduced in [7]. In this paper, an example of this is provided. The example is the following CFG:

$$S \to NP\ VP$$
$$VP \to V\ \ NP$$
$$V \to V\ \ Conj\ \ V$$
$$NP \to Det\ \ N$$
$$NP \to John$$
$$V \to found$$
$$V \to caught$$
$$Conj \to and$$

$$Det \to a$$

$$N \to unicorn$$

This CFG produces a language, this language contains the string *John found a unicorn*. In the paper it is stated that determining whether the string belongs in the language produced by the CFG is equivalent to deciding whether the following Datalog program:

```
s(i, j) :- np(i, k), vp(k, j).
vp(i, j) :- v(i, k), np(k, j).
v(i, j) :- v(i, k), conj(k, l), v(l, j).
np(i, j) :- det(i, k), n(k, j).
np(i, j) :- john(i, j).
v(i, j) :- found(i, j).
v(i, j) :- caught(i, j).
conj(i, j) :- and(i, j).
det(i, j) :- a(i, j).
n(i, j) :- unicorn(i, j).
```

Alongside the following data:

```
john(0, 1).
found(1, 2).
a(2, 3).
unicorn(3, 4).
```

Can derive the following query:

```
? - s(0, 4).
```

Something noteworthy from the above transformation is that the inherent *order* that CFGs contain through their rules is expressed in Datalog by the usage of terms which resemble indices of a string. For the rule VP, the corresponding Datalog clause contains 2 predicates that contain ordered indices in them. In this case, it is stated that the first non-terminal V has the same starting index as VP and the last non-terminal NP has the same ending index as VP. The connection between V and NP is that the first's ending index is the latter's starting index. This start-end "chain" can be further examined for the rule V where the *conj* predicate contains neither *i* nor *j*, thus being dependent on the evaluation of the two *v* predicates.

This example is not accompanied by a formal definition due to the fact that the paper focuses on showing that a similar transformation to Datalog is possible for more powerful grammar formalisms with context-free derivations, such as tree-adjoining grammars, IO macro grammars and (parallel) multiple context-free grammars. Futhermore, an augmentation of CFGs is proposed where the left-hand side of each rule is annotated with a λ-term that tells how the meaning of the left-hand side is composed from the meanings of the right-hand side nonterminals. For these CFLGs (context-free λ-term grammars) an extended transformation is given. The correction of this transformation is proven. Also, the computational complexity of the transformation alongside the Datalog evaluation is stated. Finally, it is established that a magic-set rewriting of the resulting program coincides with the deduction system for Earley parsing.

3. TRANSFORMATION OF BOOLEAN GRAMMARS TO DATALOG

3.1 Parsing of Boolean grammar into AST

As stated earlier, Boolean grammars are represented through a finite set of rules which contain terminal and non-terminal symbols. Each rule is of the following format:

$$Head \rightarrow Tail$$

$Head$ is a single non-terminal symbol and $Tail$ is a tuple of conjuncts. Conjuncts are **ordered** sets of terminal/non-terminal symbols and are separated by the $\&$ character. These symbols may possibly contain the $\neg$ character before them to denote negation. Usually non-terminal characters begin with an uppercase letter, whereas terminal characters begin with lowercase.

Grammars are most of the time written as a set of these rules instead of using their formal definitions. These rules are separated by new lines and the information regarding terminals and non-terminals exists in the rules themselves. This is done either by the uppercase/lowercase distinction, or by simply considering that all non-terminals need to be included at least once in the head of a rule. Hence, all terms not found in the head of a rule are terminals. Finally, the S symbol is traditionally used as the starting symbol.

Following this expression of grammars, it is important to note that the set of all grammars that fall under a category (CFG, conjunctive, Boolean) is essentially a language and can be described through a context-free grammar.

In the case of CFGs which are sets of rules with only one conjunct, a CFG that describes them is the following:

$$S \rightarrow RuleList$$
$$RuleList \rightarrow Rule\ RuleList$$
$$RuleList \rightarrow \epsilon$$
$$Rule \rightarrow Head\ right\_arrow\ Tail$$
$$Head \rightarrow string$$
$$Tail \rightarrow StringList$$
$$StringList \rightarrow string\ StringList$$
$$StringList \rightarrow string$$

Where $right\_arrow$ is the $\rightarrow$ character and $string = [a-zA-Z][a-zA-Z0-9]^*$ in regular expression terms.

Similarly, conjunctive grammars share almost all of the above rules with the exception of a different $Tail$ rule and the addition of a $Conjunct$ and a $Tail2$ rule:

$$\ldots$$
$$Tail \rightarrow Conjunct\ Tail2$$
$$Tail2 \rightarrow amperscand\ Conjunct\ Tail2$$

$$Tail2 \to \epsilon$$

$$Conjunct \to StringList$$

$$\ldots$$

The $amperscand$ terminal is the $\&$ character.

Finally, Boolean grammars modify the $StringList$ and add the $NegString$ rules to allow negative terminals/non-terminals:

$$\ldots$$

$$StringList \to NegString\ \ StringList$$

$$StringList \to NegString$$

$$NegString \to string$$

$$NegString \to complement\ \ string$$

$$\ldots$$

Where $complement$ is the $\neg$ character.

In the aforementioned formalism, CFGs are a proper subset of conjunctive grammars and conjunctive grammars are a proper subset of Boolean grammars. Hence, from now on we will be considering that every grammar falls under the Boolean class.

The CFG of Boolean grammars provided above allows us to create a simple and efficient parsing algorithm to reduce any grammar into an AST. Considering that the CFG is in $LL(1)$ form, the algorithm is very simple to be created but its creation is out of the scope of this thesis. Also, all aforementioned existing algorithms like CYK and Earley Parsing can be used.

3.2 Conversion of AST to Datalog

An abstract syntax tree, as mentioned earlier, is a tree-like structure used to represent the structure of a program. In our case, the AST's root node has all the grammar rules as children nodes. Each of these rule-nodes will be converted into a Datalog clause.

Grammars inherently contain the essence of order in their conjuncts, this is a key point that we need to represent in our Datalog transformation. A simple approach is to add indices as terms/variables in the Datalog predicates. Considering a simple rule $S \to a\ b\ c\ d$, the positions of a, b, c, d are essentially one after another with a as the starting point. Hence, these positions could be expressed as follows (assuming they are terminals of length 1):

```
s :- a(0), b(1), c(2), d(3).
```

a is in the 1st position, b is in the 2nd, etc.

The above program would answer $true$ in a query for S if and only if the facts $a(0)$, $b(1)$, $c(2)$, $d(3)$ would be added as ”data” in our program.

Although this apporach seems elegant, we run into a problem when facing rules containing non-terminals. Regarding non-terminals, their position is unclear and the only information we have about them is which terminals/non-terminals exist before or after them. Hence, we need to include an ending index for all predicates. Also, non-terminals cannot be expressed through simple numeric literals. For that reason we need to add variables/terms to our Datalog clause to encapsulate this uncertainty. Consider the rule $S \to a\ S\ a$. The first a starts at the beginning of the non-terminal, the second a finishes at the end of it and S is somewhere in the middle. A simple clause for this rule is the following:

```
s(start, end) :- a(start, i), s(i, j), a(j, end).
```

An important thing to note is that, by definition, $start \leq i \leq j \leq end$ (the only case where $\leq$ occurs instead of $<$ is if the empty string ϵ exists inside the rule, either directly or indirectly). The inclusion of ϵ is something we should also consider but we will do so later.

Having established the process for a rule containing both terminals and non-terminals, a definition for the transformation is the following.

Given a rule $S \to a_1\ a_2\ ...\ a_n$, we define a transformation to a Datalog clause as a set of ordered predicates with a_1 and a_n starting and ending at the $start$ and end variables respectively, with in-between indices added to accomodate for the other predicates:

```
s(Start, End) :- a1(Start, I2), a2(I2, I3), ..., aN(IN, End).
```

Where a_i (and the respective predicate) can be either referring to terminal or non-terminal nodes.

In the case where a rule is of the form $S \to \epsilon$, the corresponding transformation would result in the following fact (the ϵ character denotes the empty string):

```
s(I, I).
```

The explanation for the above clause is that S begins and ends at the same index, thus being the empty string.

When this transformation is applied to all the rules of the grammar, the query should follow similar rules: Each character/symbol of the query string needs to be stated as ”data” with its corresponding start & end indices. Finally, the actual query should be over the starting symbol S from 0 to the length of the query. Formally, given a string x of length $N+1$ where $x = \{x_0, x_1, x_2, ..., x_n\}$ (each x_i is a symbol), the data needs to be formatted in the following way:

```
x0(0, 1).
x1(1, 2).
x2(2, 3).
...
xN(N, N + 1).
```

The query will be

```
?- s(0, N + 1).
```

As an example, the corresponding CFG of the language $L = \{wcw^R : w \in \{a, b\}^*\}$ is:

$$S \to aSa$$
$$S \to bSb$$
$$S \to c$$

The CFG would be converted into the following Datalog program:

```
s(Start, End) :- a(Start, I2), s(I2, I3), a(I3, End).
s(Start, End) :- b(Start, I2), s(I2, I3), b(I3, End).
s(Start, End) :- c(Start, End).
```

Assuming we would like to query the string $bacab$, the data would have the form:

```
b(0, 1).
a(1, 2).
c(2, 3).
a(3, 4).
b(4, 5).
```

And the final query would be

```
?- s(0, 5).
```

In this case, the query would resolve to $true$.

Currently we have only seen the case where each rule has a single conjunct (i.e. CFGs) which is essentially what the Kanazawa paper [7] introduces. In order to extend this, we need to add logic for conjunction. Thankfully, a Datalog clause is evaluated to $true$ when all predicates are evaluated to $true$, thus inherently containing the conjunction operation. Simply put, for every conjunct, we just need to add its start-end chain as a list of predicates in the same clause.
Given a rule in the following form:

$$S \to a_{11} \; ... \; a_{1n} \;\&\; ... \;\&\; a_{m1} \; ... \; a_{mk}$$

The resulting clause would be:

```
s(Start, End) :- a11(Start, I12), ..., a1N(I1N, End),
                 ...,
                 aM1(Start, IM2), ..., aMN(IMK, End).
```

It is important to note that the in-between indices (I_{xy}) are -by necessity- different between conjuncts so as they are evaluated independently. The query/data clause-generation part remains the same.

As an example, the corresponding conjunctive grammar of the language $L = \{a^n b^n c^n : n \geq 0\}$ is:

$$S \to XY \& WZ$$
$$X \to aX \,|\, \epsilon$$
$$Y \to bYc \,|\, \epsilon$$
$$Z \to cZ \,|\, \epsilon$$
$$W \to aWb \,|\, \epsilon$$

The resulting Datalog program after the transformation is:

```
s(Start, End) :- x(Start, I12), y(I12, End), w(Start, I22), z(I22, End).

x(Start, End) :- a(Start, I12), x(I12, End).
x(I, I).

y(Start, End) :- b(Start, I12), y(I12, I13), c(I13, End).
y(I, I).

z(Start, End) :- c(Start, I12), z(I12, End).
z(I, I).

w(Start, End) :- a(Start, I12), w(I12, I13), b(I13, End).
w(I, I).
```

Finally, we are ready to complete the final step: adding negation. Up until now we were dealing with positive Datalog whose evaluation is simple. Now the usage of negation is necessary. Syntax-wise we will use the **not** predicate for this thesis but most Datalog engines have a different syntax for negation.

Simply, the **not** predicate can be added for every negative terminal/non-terminal in the AST. The only condition where problems may arise is when negation is applied to more than one symbols.

Considering the following rule:

$$A \to \neg(B\,C)\,\neg D$$

This is something that needs to be addressed as some Datalog engines do not allow multiple predicates inside the **not** predicate. A simple solution is to create intermediate rules for these occasions. For the above example, a new rule for B and C would be created and it would take their place in the tail of rule A.

$$A \to \neg X\,\neg D$$
$$X \to B\,C$$

As an example, the corresponding Boolean grammar of the language $L = \{ww : w \in \{a,b\}^*\}$ is:

$$S \to \neg XY \& \neg YX \& Z$$

$$X \to TXT \mid a$$
$$Y \to TYT \mid b$$
$$Z \to TTZ \mid \epsilon$$
$$T \to a \mid b$$

For this grammar, the complement character in $\neg XY$ and $\neg YX$ encapsulates both symbols hence we would have to write intermediate clauses converting the XY and YX sequences into a single conjunct which is negated in the original rule (S) like so:

$$S \to \neg K \;\&\; \neg L \;\&\; Z$$
$$K \to X\ Y$$
$$L \to Y\ X$$
$$\ldots$$

Accounting for this minor conversion, the resulting program is:

```
s(Start, End) :- not(k(Start, End)), not(l(Start, End)), z(Start, End).

k(Start, End) :- x(Start, I2), y(I2, End).
l(Start, End) :- y(Start, I2), x(I2, End).

x(Start, End) :- t(Start, I2), x(I2, I3), t(I3, End).
x(Start, End) :- a(Start, End).

y(Start, End) :- t(Start, I2), y(I2, I3), t(I3, End).
y(Start, End) :- b(Start, End).

z(Start, End) :- t(Start, I2), t(I2, I3), z(I3, End).
z(I, I).

t(Start, End) :- a(Start, End).
t(Start, End) :- b(Start, End).
```

The addition of negation to the transformation allows us to express Boolean grammars as well. The transformation rules are clearly backwards compatible concerning the subsets of Boolean grammars. Thus, we now have converted any grammar into a Datalog program which is ready to be evaluated through an engine.

3.3 Datalog Evaluation

We mentioned earlier that the Boolean grammars we use in this transformation follow the well-founded semantics. These semantics, in short, ensure that negation circles do not break the grammar, rather they evaluate that these rules produce the language that returns an *undefined* value for any queried string. Equivalently, the Datalog program that is produced also uses its own well-founded semantics that we covered previously. This happens because the negation circles that exist in a Boolean grammar carry over to the

Datalog transformation, hence any other form of logic programming semantics would have invalid evaluations in some cases.

Consider the simple Boolean grammar G which consists only of the following rule:

$$S \to \neg S$$

Through the well-founded semantics for Boolean grammars, S would be evaluated to produce the language that for every queried string, its membership in the language produces $\frac{1}{2}$ or *undefined*. The produced Datalog program using our transformation is:

```
s(Start, End) :- not(s(Start, End)).
```

The evaluation of this logic clause into $\frac{1}{2}$ for every query can only be achieved through the well-founded semantics. Other negative-logic-programming semantics like Stratified [12] or Locally Stratified [1] semantics do not apply in the aforementioned condition (as S in both cases belongs in the same *stratum*).

4. CONCLUSIONS AND FUTURE WORK

In this thesis we have provided an extension to the transformation of CFG to Datalog introduced in the Kanazawa paper [7]. Our transformation is extended to include two powerful formal grammars, conjunctive and Boolean grammars. These grammars were introduced by Okhotin in [10] and [11] respectively. Conjunctive grammars add the conjunction operation to CFGs, thus achieving the fact that they are closed under conjunction, in contrast to CFGs. Boolean grammars add the negation operation to conjunctive grammars, thus being closed under complement, which is something we are unaware whether is true or not regarding conjunctive grammars. In order to achieve this transformation, we delved into the Well-Founded semantics for Boolean grammars in order to encapsulate their expressiveness in full. Equivalently we used the Well-Founded semantics for logic programs as other proposals for semantics were lacking in terms of handling negation circles.

The work on this thesis can be clearly expanded by creating a similar transformation for other grammar formalisms like context-sensitive grammars or regular grammars. Furthermore, a compelling thought would be proving that the aforementioned transformation of a boolean grammar alongside the evaluation of the resulting logic program through the WF semantics is equivalent to evaluating the grammar by using the respective WF semantics for Boolean grammars. Finally, we have not added any segment regarding either time or space complexity of the transformation alongside the evaluation, something which would also be an interesting topic.

Grammars and logic programming, although having many differences, they have similar syntax. Our transformation enhances our intuition on the connection between logic programs and formal grammars, bringing these two worlds closer and giving us an insight regarding their nuanced relationship.

BIBLIOGRAPHY

[1] Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–21, 1989.

[2] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[3] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.

[4] John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969.

[5] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1), 2001.

[7] Makoto Kanazawa. Parsing and generation as datalog queries. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 176–183, 2007.

[8] Vassilis Kountouriotis, Christos Nomikos, and Panos Rondogiannis. Well-founded semantics for boolean grammars. *Information and Computation*, 207(9):945–967, 2009.

[9] Hans-Jörg Kreowski. A pumping lemma for context-free graph languages. In *International Workshop on Graph Grammars and Their Application to Computer Science*, pages 270–283. Springer, 1978.

[10] Alexander Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[11] Alexander Okhotin. Boolean grammars. *Information and Computation*, 194(1):19–48, 2004.

[12] Teodor C Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming*, pages 193–216. Elsevier, 1988.

[13] Itiroo Sakai. Syntax in universal translation. In *Proceedings of the International Conference on Machine Translation and Applied Language Analysis*, 1961.

[14] Allen Van Gelder, Kenneth Ross, and John S Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 221–230, 1988.

[15] Allen Van Gelder, Kenneth A Ross, and John S Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3):619–649, 1991.