SOURCE TRANSFORMATION
WITH BOOLEAN GRAMMARS

by

ANDREW STEVENSON

A thesis submitted to the
Graduate Program in Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
September 2015

Copyright © Andrew Stevenson, 2015
Abstract

We propose an enhancement to current parsing and transformation systems by leveraging the expressive power of Boolean grammars, a generalization of context-free grammars that adds conjunction and negation operators. In addition to naturally expressing a larger class of languages, Boolean grammars capture multiple parse trees of the same document simultaneously, allowing one to switch between these parse “views”. In particular, source transformation and reengineering tasks can benefit from parse views by recasting the input text into whichever parse is most suitable for the task at hand.
Acknowledgments

I would like to thank, first and foremost, my supervisor Jim Cordy. I could not have accomplished this without his guidance and unwavering support. We worked closely together to develop many of the ideas in this thesis, and this collaboration is the reason I use the first person plural voice (we, us, our) within.

I’d also like to give a special thanks to Anne Wills for her extensive editing and insights to improve this document, and to Martin and Peter Wills for their valuable edits.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices. A short summary of the work reported in this thesis has previously appeared as [22], co-authored with my supervisor James R. Cordy.

Andrew Stevenson

September 29, 2015
Table of Contents

Abstract i

Acknowledgments ii

Statement of Originality iii

Table of Contents iv

List of Tables vii

List of Figures viii

Chapter 1:

Introduction 1

1.1 Motivation and Contributions 2

Chapter 2:

Background 5

2.1 Boolean Grammars 6

2.2 The TXL language 9

2.3 Parse Views 16

2.4 Related Work 21
Chapter 3:

Overview .................................................. 26
3.1 Parser Module ........................................... 27
3.2 Transformer Module ...................................... 29
3.3 Boolean Transformation Semantics .................... 34
3.4 Summary .................................................. 36

Chapter 4:

A Practical Boolean Parser ................................. 37
4.1 New grammar syntax ...................................... 37
4.2 Parsing Semantics ........................................ 39
4.3 Expressiveness advantages of Boolean parser ........... 43
4.4 Parsing Efficiency ........................................ 44
4.5 Summary .................................................. 46

Chapter 5:

A Practical Boolean Transformer ............................ 47
5.1 Boolean transformation semantics ....................... 49
5.2 Ongoing Reparse Semantics .............................. 54
5.3 Direct vs. Indirect Conjunctive Targets .................. 57
5.4 Optimizations ............................................. 62
5.5 Summary .................................................. 64
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The TXL Boolean grammar operators and their meaning</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>A backtracking parse trace of the string <code>ababa</code> using the grammar in Figure 4.1. The details of the trace unrelated to conjunction have been ommitted for clarity.</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>The circumstances under which disjunction and conjunction are most and least efficient due to short circuiting.</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>The intermediate result of a transformation may be malformed even if its final result is well-formed</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A context-free parse tree (left) and a Boolean grammar parse DAG (right). The concatenation of leaves, $w$, must be the same for all positive conjuncts in a Boolean grammar production. [16] ............ 7
2.2 The TXL transformation pipeline ........................................... 10
2.3 An island grammar in SDF [14] ................................................. 20
2.4 An island grammar in TXL ....................................................... 20
3.1 The enhanced TXL transformation pipeline ............................... 26
3.2 Grammars and their corresponding parse of the math expression $5+3*7$ for (a) CFG with a left-to-right parse, (b) CFG with a typical precedence parse, and (c) Boolean grammar that incorporates both (a) and (b). The parse of a Boolean grammar can be a DAG, unlike for CFGs which must be trees. ......................................................... 28
3.3 (a) A simple English grammar ($S$=sentence, $NP$=noun phrase, $VP$=verb phrase, $A$=adjective, $N$=noun, $V$=verb) in EBNF notation, and (b) the same grammar but allows embedded HTML tags (shown in boldface). The $Any$ nonterminal represents any valid text between < > in a markup tag, including “/” to denote a closing tag. .................. 30
3.4 A Boolean grammar parse of the string <QUOTE>Good boys <EM>deserve</EM> fun</QUOTE> using an English focused parse tree and an HTML focused parse tree (inverted). Both parse trees share the same leaves.  
3.5 A TXL program to change all <EM> HTML markup to <STRONG>, using a Boolean grammar and HTML-specific view. 
3.6 A TXL program to change all instances of the adjective “close” to “nearby”, using a Boolean grammar and English-specific view. 
4.1 A Boolean grammar with annotated sublanguages. 
4.2 A Boolean grammar to ensure the number of variables, types, and values are equal in a variable declaration statement. 
5.1 (a) Boolean grammar for the language \( \{a^n b^n c^n | n \geq 0\} \) and (b) a parse of the string aaabbcccc with the \( AB c^* \) view and the \( a^* BC \) view (inverted) 
5.2 (a) a well-formedness preserving rule, and (b) a rule that invalidates other views 
5.3 A TXL rule directly targeting a conjunctive type, before and after its meta-transformation. Common lines have been aligned to better demonstrate the effect of the meta-transformation. 
5.4 The parse tree structure for a rule that directly or indirectly targets a conjunctive type. In both cases the conjunction needs to be reparsed. 
5.5 A TXL rule indirectly targeting conjunctive type \( S \), before and after its meta-transformation.
6.1 (a) A simple but highly ambiguous expression grammar, and (b) its
unambiguous precedence version. Both grammars generate the same
language. .......................................................... 67

6.2 A TXL program to parenthesize subexpressions. Each level in the
precedence grammar requires its own rule. .......................... 70

6.3 A TXL program, using Boolean grammar overrides, to parenthesize
subexpressions. Each precedence level can be viewed as an expression,
requiring a single rule that targets \texttt{Expr}. ......................... 71

6.4 A TXL program, using Boolean grammar overrides, to parenthesize
subexpressions. The \texttt{NotPrimary} nonterminal encodes the precise
transformation target, further simplifying the transformation rule. . . 72

6.5 A standard TXL program to type check subexpressions (continued on
next page). The \texttt{propagateTypes} rule calls 5 subrules, one for each
level in the precedence grammar. ............................... 75

6.5 A standard TXL program to type check subexpressions (continued
from previous page). The \texttt{propagate\*} rules are all similar, suggesting
redundancy. ............................................................ 76

6.6 A TXL program, using Boolean grammar overrides, to type check
subexpressions. Each precedence level can be viewed with a generic
\texttt{Expr} and \texttt{Op}, requiring a single rule in the general case. ....... 77
Chapter 1

Introduction

Ever since Noam Chomsky’s formalism of them in 1956 [2], context-free grammars have been the overwhelming choice of programming language designers to describe the basic syntax of their creations. However, modern languages require some context-sensitive checks, such as verifying correct variable scope, typically done in later passes of a compiler after the context-free parsing is complete. The context-free parsing problem is well understood and many tools are available to generate an efficient parser from a context-free grammar description.

This thesis explores an alternative to context-free grammars for parsing programming languages, namely Boolean grammars. Boolean grammars were conceived by Alexander Okhotin and represent a natural grammar formalism whose expressiveness lies between context-free and context-sensitive grammars [16]. Okhotin has proven several characteristics of Boolean grammars that make them desirable for describing programming languages, most significantly: (1) text can be parsed by a Boolean grammar in polynomial time [16], and (2) Boolean grammars can encode some context-sensitive traits (for example checking a variable reference is within the...
variable’s scope [17]).

We explore the practical implications of Boolean grammar parsing by enhancing an existing grammar engineering tool called TXL [4]. TXL not only parses documents via a context-free grammar, but is capable of transforming those documents as well. This offers us an opportunity to explore, for the first time, the transformation of parse trees derived from Boolean grammars.

**Statement of Thesis:** The unique characteristics of Boolean grammars can be used to enhance current parsing and source transformation systems for software reengineering tasks.

### 1.1 Motivation and Contributions

The broad motivation for this research comes from three observations: (1) context-free grammars are an valuable formalism in practical software tools, (2) Boolean grammars are a more powerful formalism, and (3) there is a lack of software tools supporting Boolean grammars. At a high level, this research aims to fill this gap by applying Boolean grammars to existing reengineering tools, such as a parser and source transformation engine, and highlighting the resulting improvements.

The mathematical description of Boolean grammars is now well developed, but there are few practical implementations of a Boolean grammar based parser. Such an implementation is a key contribution of this research. Grammar authors and programming language designers benefit from a Boolean grammar based parser because it allows them to express and parse languages that are impossible to express with a context-free grammar.
Another motivation for this work is the realization that many reengineering tasks benefit from having access to multiple parses of the input text, and that Boolean grammars have a natural mechanism to provide this. We have called this idea parse views, where a “view” is a way of looking at complex data that hides irrelevant details. Parse views and their applications are elaborated in Section 2.3.

One reengineering task that benefits from parse views is source transformation, the process of transforming a structured document (often source code) into another structured document. The implications of transforming documents parsed with a Boolean grammar is a major motivation of this research, as we are not aware of any other research attempting to look at this problem.

With these motivations in mind, the concrete contributions of this thesis are:

- A parser implementation supporting Boolean grammars
- A transformer implementation supporting Boolean grammars
- Several approaches for replacing subtrees in a Boolean grammar’s parse tree, and a discussion about the advantages of each
- Applications of software transformation tasks with Boolean grammars

Chapter 2 contains the necessary background about Boolean grammars, the TXL language and transformation engine, and related work to understand subsequent chapters. Chapter 3 shows the overall design and approach used in the parser and transformer based on Boolean grammars. Chapter 4 describes the new grammar syntax and parsing semantics needed to support Boolean grammars, and discusses the advantages of a Boolean parser. Chapter 5 goes into detail about the implications and unique problems when transforming parse DAGs, and discusses several possible
transformation semantics along with the advantages of each. Chapter 6 shows how Boolean grammars and parse views can be beneficial for practical grammar reengineering problems, in particular those related to agile parsing. Chapter 7 concludes with a discussion about lessons learned and the direction future work may take.
Chapter 2

Background

The author’s work builds on the work of others. The purpose of this chapter is to provide enough background information to the reader to inform the rest of the thesis. At a high level, this thesis talks about how the theoretical Boolean grammar formalism relates to the practical idea of parse views within the context of the TXL source transformation tool. This chapter focuses on these three concepts.

Alexander Okhotin formalised Boolean grammars in 2004 [16] and explored their implications in subsequent years [17, 21, 12]. Although he considered several practical uses of conjunctive and Boolean grammars like parsing [15, 20] and programming language design [18], his focus was mostly from the perspective of formal language theory.

The goal of our research is to raise awareness of Boolean grammars as a beneficial alternative to context-free grammars by incorporating them into a mainstream grammarware tool. We chose the TXL source transformation system [4] for this purpose because of its maturity and our existing experience with it.

In Section 2.1 we introduce the theory of Boolean grammars and discuss how
its expressiveness applies to programming languages. Section 2.2 describes the TXL language and source transformation engine. Next we discuss the idea of parse views, in particular agile parsing and island grammars, in Section 2.3. The chapter closes with Section 2.4 on related work in Boolean and context-free parsers.

2.1 Boolean Grammars

The familiar context-free grammar $G = (\Sigma, N, P, S)$ contains productions of the form

$$A \rightarrow \alpha$$
$$A \rightarrow \beta$$

where $\alpha$ and $\beta$ are juxtapositions of terminals and nonterminals ($\alpha, \beta \in (\Sigma \cup N)^*$) and the right-hand sides of a particular nonterminal are separated by disjunction (i.e. $A$ derives $\alpha$ or $\beta$). Boolean grammars generalize this formalism by introducing conjunction ($\&$) and negation ($\neg$)

$$A \rightarrow \alpha_1 \& \cdots \& \alpha_m \& \neg \beta_1 \& \cdots \& \neg \beta_n$$

while still maintaining disjunction from context-free grammars [16]. The production $S \rightarrow A \& B \& \neg C$ captures the idea that the input must be parsable as an $A$ and also as a $B$ but must not be parsable as a $C$. Successfully parsing the goal $S$ requires the parse goals $A$, $B$, and $\neg C$ to all succeed on the same text.

A \textit{conjunct} is an operand of the conjunction ($\&$) operator. Conjuncts can be either \textit{positive} or \textit{negative}. In the production above, $\alpha_1, \cdots, \alpha_m$ are positive conjuncts and
CHAPTER 2. BACKGROUND

Figure 2.1: A context-free parse tree (left) and a Boolean grammar parse DAG (right). The concatenation of leaves, \( w \), must be the same for all positive conjuncts in a Boolean grammar production. [16]

\[ \beta_1, \ldots, \beta_n \] are negative conjuncts (note the preceding \( \neg \) symbol). Positive conjuncts are equivalent to parse views, and we use both terms throughout this thesis. We tend to use the term “conjunct” in a more theoretical context (when speaking of strings, grammars, productions, etc.) and “parse view” in a more practical context (when speaking of programs, source code, types, etc).

A Boolean grammar’s parse tree leaves are shared among its positive conjuncts, resulting in a directed acyclic graph (DAG) instead of a tree. This DAG is not a general DAG, but rather a restricted form of DAG where the only nodes that can have multiple parents are leaf nodes. Figure 2.1 shows the difference between a parse tree for a context-free grammar and a parse DAG for a Boolean grammar [16].

2.1.1 Expressiveness for Programming Languages

In this section we discuss how the expressiveness of Boolean grammars relates to programming language parsing and design. We begin with some limitations of context-free grammars that effect real programming languages, then mention some common non-context-free programming language constraints Boolean grammars can enforce.
Most programming language specifications contain constraints that cannot be expressed in a context-free grammar with Backus-Naur productions alone. For example, Floyd showed that the Algol 60 language cannot be fully defined using a context-free grammar because the program

\[
\begin{align*}
\text{begin real } x^{(n)}; x^{(n)} := x^{(n)} \text{ end}
\end{align*}
\]

(where \(x^{(n)}\) is \(n\) occurrences of \(x\)) corresponds to the non-context-free language \(\{a^n b^n c^n | n \geq 0\}\) \cite{8}. This language can, however, be expressed with a Boolean grammar.

Another well known problem is the “typedef-name: identifier” problem in the C programming language, where the meaning of an expression such as \((T)*x\) is interpreted as the dereference of \(x\) casted to type \(T\) (if \(T\) is a type), or the multiplication of \(T\) by \(x\) (if \(T\) is not a type). Disambiguation depends on whether \(T\) was previously defined in a typedef or not, which is context-sensitive information.

Boolean grammars, however, can express a larger class of languages than the class of context-free languages, allowing a grammar author to specify language constraints not possible with current grammar authoring tools. Alexander Okhotin illustrates some programming language constraints that can be expressed in a Boolean grammar but not in a context-free grammar \cite{18} using a Boolean grammar containing 123 nonterminals and 368 productions.

The starting production of this grammar specifies the structure of the language as well as each high-level constraint (i.e. all functions are defined and their arguments are
consistent, there exists a main function, and there are no duplicate function names):

\[
S \rightarrow \text{Functions} \& \text{AllFunctionsDefined} \& \\
\text{HasMainFunction} \& \neg\text{DuplicateFunctions}
\]

This grammar also enforces the following non-context-free constraints:

1. Function constraints:

   (a) All functions must have unique names

   (b) The number of arguments in a function call must match the number of
       formal arguments in the associated function definition

2. Variable scoping:

   (a) Defining the scope of a variable from its declaration context

   (b) Different variables of the same name do not exist in the same scope

   (c) A variable assignment statement must be in that variable’s scope

2.2 The TXL Language

TXL is a mature, industrial strength transformation engine well suited to both aca-
demic and industrial transformation tasks [3]:

TXL has been widely used in research applications in industry and academia
as well as in production commercial applications handling inputs of up
TXL is intended for the transformation of documents with structured text, in particular source code. It does this by parsing the input text according to a grammar, changing the resulting parse tree according to a set of rules, then unparsing the tree back into text again. Figure 2.2 shows this transformation pipeline.

The meanings of the terms “parsing” and “unparsing” are overloaded [27]. In this thesis we use them in the same way that TXL uses them. Specifically, parsing involves tokenizing the input text (combining finite sequences of characters into typed tokens, typically with layout information removed, yielding a sequence of tokens) then building a syntax tree that conforms to a context-free grammar. We use unparsing to mean transforming the syntax tree back into text, with layout characters inserted by default rules and from formatting cues defined in the grammar.

The term “TXL” is used for both (1) the language used to specify the grammar and transformation rules, and (2) the interpreter which executes the actual parsing and transformation steps. For this section, these differences will be made explicit by
using the terms “TXL language”, “TXL parser”, and “TXL transformer”.

The TXL language can be characterized by having strongly, statically-typed variables, a functional paradigm for task decomposition, pattern matching with unification, and implied iteration. Like other statically-typed languages, the types of TXL variables are specified when they are first declared (e.g. `Count [number]`). Furthermore, variables are bound to an immutable value upon declaration and these bindings are final. Values can be any terminal or nonterminal allowed by the grammar definitions.

The functional call semantics of TXL work slightly differently than those of other functional languages. Specifically, the value returned by a TXL rule or function is always the same type as the value given to it. The input value is known as the `scope` of a rule or function. Rather than thinking of rules and functions as things that take an input tree and return an output tree, it is more convenient to think of rules and functions as things which are applied to a tree and can modify it. Rules and functions can only modify the scope to which they are applied, a key property of TXL that enhances program comprehension by limiting side effects. For instance, in the TXL expression `P [R1][R2][R3]`, `P` is the scope and the rules/functions `R1`, `R2`, `R3` are each applied to that scope in succession. In other words, the result of the expression is `R3(R2(R1(P)))` which is guaranteed to be of the same type as `P`.

### 2.2.1 Syntax

The TXL language is a domain-specific programming language whose syntax can be split into two main categories: grammar definition and transformation rules. The `define` and `redefine` statements define the grammar productions, while the `rule`
and function statements specify the transformation rules. For example, to capture a
programming language construct such as the common if-then control flow structure:

```plaintext
if (...) then
    ...
end if
```

one could use the TXL definition:

```plaintext
define if_statement
    if ( [expression] ) then
        [repeat statement]
    'end if
end define
```

In this example, the keyword end in the language to be transformed overlaps
with the syntax of the TXL language itself, so it must be quoted using a leading
apostrophe. Keywords which don’t overlap, such as if or then can be quoted or left
unquoted.

The redefine statement is used to override a previously defined nonterminal by
either extending it or replacing it entirely. For example, the definition of an if
statement above can be extended to include a version with an else clause:

```plaintext
redefine if_statement
    ...
    | if ( [expression] ) then
    |     [repeat statement]
    | else
    |     [repeat statement]
    'end if
end define
```

Rules and functions describe the transformation behaviour of a TXL program,
and are defined by rule and function statements respectively. The general form of
both is:
rule <rule_name>
  replace <rule_type>
  <pattern>
  by
  <replacement>
end rule

This syntax specifies a pattern to match in the parse tree and its corresponding replacement, thus representing a transformation. The rule type determines both the input and output types of the rule or function.

2.2.2 The TXL Parser

Although TXL is primarily a transformation engine, its parser is an important component and warrants its own discussion, especially in the context of this thesis. TXL uses a top-down parser with full backtracking, meaning that it starts at the highest level of the grammar tree and works down the tree by following the rewriting rules defined by the grammar productions. In TXL, the overall goal is to parse the input as the top-most type, [program]. If the definition of [program] is [A][B] then the overall goal becomes two subgoals: (1) parse the first segment of the input as a [A], and (2) parse the remaining input as a [B].

The TXL parser supports full backtracking, meaning if the current subgoal fails the parser backtracks to an earlier point and unreads some of the input it has consumed thus far. It then retries with the next viable alternative as the new subgoal. A backtracking parser will eventually find a parse if one exists, but may take exponential time in the worst case. Care is needed when writing grammars in TXL to prevent inefficient parses, and tricks like not nesting [repeat x] definitions are well known to TXL programmers.
Another such trick involves TXL’s special \( \text{not } X \) nonterminal, which causes the current parse goal to fail if the prefix of the unconsumed input tokens can be parsed as an \( X \). This \( \text{not} \) nonterminal causes a parse goal failure and backtracking based on look-ahead input, but never causes input to be consumed. In contrast, a Boolean grammar’s negation operator does cause input to be consumed, but ensures that input does not parse as a certain form. For example, the right-hand side nonterminal TXL definition \( [A] \text{not } B \) means “parse the first part of the unconsumed input as an \( A \) and the prefix of the remaining part must not be parsable as a \( B \).” On the other hand, the nonterminal definition \( [A] \text{ &! } [B] \) (using the Boolean grammar syntax introduced in Section 4.1) means “parse the first part of the unconsumed input as an \( A \) and that same part must not be parsable as a \( B \).”

In contrast to some other grammar description languages [9], the order of alternatives in a TXL grammar production is significant. The TXL parser tries alternatives in the order they appear in the \texttt{define} statement, and only tries later alternatives if all the earlier ones fail to produce a valid parse.

2.2.3 The TXL Transformer

The purpose of the TXL transformer is to apply the written functions and rules to the parse tree supplied by the TXL parser. The difference between functions and rules in TXL are in the matching and transforming semantics:

- A function’s type must match the type of the scope to which it is applied. When both types are equal and the function’s pattern matches the scope, the semantics during execution is to replace the scope with the function’s replacement. (Note: a function of type \( A \) applied to a scope of type \( B \) will fail to
• A rule’s type is not required to match the type of the scope to which it is applied. The semantics here are to search (depth-first) within the scope for a subtree that matches the rule’s type and pattern. If such a subtree is found, replace it with the rule’s replacement and re-search the scope from the top until no more matches are found.

An example of an appropriate use for a TXL rule is transforming the redundant expression \( x == \text{false} \) (some boolean variable is compared to the constant \text{false}) into its terser form \(!x\):

```plaintext
rule falseCheckToNegation
  replace [expression]
    Var [id] == false
  by
    !Var
end rule
```

When such a rule is applied to a much larger scope, for example the entire program, then all instances of the \( x == \text{false} \) pattern in the whole program are replaced with \(!x\). This offers a lot of power for a relatively small amount of code.

TXL ensures that functions and rules are homomorphic (input type and output type are the same) by parsing both the pattern and replacement as the rule/function type according to the grammar definition. In the example above, the pattern \( \text{Var [id] == false} \) and its replacement \(!\text{Var}\) can both be parsed as type \[\text{expression}\]. This check happens at compile time, so a mismatched transformation is guaranteed to never happen during execution.
2.3 Parse Views

A parse view is our name for a useful idiom in the grammar engineering community that has gained wide adoption in various forms. Our choice of name is inspired by a database view: a perspective that restructures and filters the underlying data in a way that makes it easy for a task to consume. In our case the data is in a parse tree and the task is a transformation.

Dean et al. refer to the technique of modifying a grammar to suit a specific task as agile parsing [6], which we introduce and summarize in Section 2.3.1. In Section 2.3.2, we discuss another common use of parse views known as island grammars.

2.3.1 Agile Parsing

When parsing a software program it is often useful to have slightly different “views” of the input text depending on the task at hand [6]. Consider some grammar productions for an assignment statement in an imperative programming language:

\[
\text{AssignStmt} \rightarrow \text{Id} := \text{Expr} \\
\text{Expr} \rightarrow \text{Id}
\]

If we want to perform a task (e.g. transformation or fact extraction) on the variable definitions but not the variable references it is useful to have a parse that distinguishes
these cases. We might rewrite the grammar as

\[
\begin{align*}
AssignStmt & \rightarrow \text{VarDef} := \text{Expr} \\
\text{VarDef} & \rightarrow \text{Id} \\
\text{Expr} & \rightarrow \text{VarRef} \\
\text{VarRef} & \rightarrow \text{Id}
\end{align*}
\]

to insert extra nonterminal nodes in the parse tree that are semantically relevant to our task. This represents a task-specific view of the input and makes the task at hand much easier to read, write, and reason about.

This is an example of grammar specialization, one of the agile parsing idioms described by Dean et al. [6]. These idioms are intended for TXL programmers and make use of its redefine statement to override nonterminals in a base grammar to suit a specific transformation task. Each set of overrides creates a parse view of the input that differs from the parse of the base grammar in some way.

Dean et al. describe seven agile parsing idioms TXL programmers find useful:

1. **Rule Abstraction**: The base grammar is too specific for the desired transformation task. The programmer can solve this by combining several related forms into a common nonterminal, then target that nonterminal in the transformation rule.

2. **Grammar Specialization**: The same nonterminal is reused in different contexts of the grammar, but the transformation task depends on this context. Override each context with a unique intermediate nonterminal that simply derives the original nonterminal, then the transformation rule can target whichever
3. **Grammar Categorization**: The opposite of rule abstraction – the base grammar is too general for the transformation task. Create a new nonterminal that captures a subset of the existing general form, then override the general nonterminal to prefer the subset (with a fall-through to the general form if the subset form fails).

4. **Union Grammars for Translation**: When performing an inter-language transformation (e.g. Pascal to C), override nonterminals at each level where the languages share a common structure to allow the forms of both languages. Successive transformations can then target specific language elements to translate until the program is completely translated.

5. **Markup**: Override a nonterminal to allow markup tags in the output. A transformation rule can then apply tags to interesting code segments. This is mostly used for code analysis and error identification.

6. **Semiparsing**: Grammar overrides and TXL’s [not] nonterminal are used to parse only a portion of the input in detail. This is similar to island grammars as discussed in the following section. (Section 2.3.2).

7. **Data Structure Grammars**: Use a new nonterminal, possibly completely detached from the input parse, to represent a data structure such as a table, map, list, etc. These data structures are often used to store global data or data read from an external file.

This thesis contains many examples of these agile parsing idioms using Boolean grammars instead of context-free grammars. The English-HTML example introduced in
Section 3.2 is an instance of two simultaneous semipareses (also known as island grammars). In Chapter 6 we give examples of grammar specialization (Section 6.2), grammar categorization (Section 6.3), and use markup to identify expressions with inconsistent types (Section 6.1.2).

### 2.3.2 Island Grammars

A practical example of parse views are island grammars [26], an abstraction of the input text that captures the parts of the input that are interesting (the “islands”) and ignores the rest (the “water”). The task at hand determines which structures are interesting or not.

Figure 2.3 shows an island grammar, written in the Syntax Definition Formalism (SDF), that identifies program calls in COBOL [14]. The program calls of interest include ones loaded dynamically by populating a global variable named “CALLEE” then calling a call-handler routine named “HANDLER”. For this reason, the “MOVE TO CALLEE” statement is of interest and “CALL HANDLER” is not. Figure 2.4 shows the same island grammar in TXL.

In Figure 2.3, the SDF \{avoid\} directive tells the parser to avoid chunks of input becoming water if it can become an island instead (i.e. one of the “call” productions applies). The \{reject\} directive enforces a “lake” within the island, a part of the island that is not of interest. Specifically, it rejects calls to a specific handler named HANDLER.

In Figure 2.4, TXL achieves the same disambiguation as \{avoid\} by its implicit ordering of alternatives; [call] appears before [water] in the chunk definition so the parser will try to parse chunks as call statements first. The \{reject\} effect is
lexical syntax
\[
\begin{align*}
&\lfloor \backslash \wedge \backslash t \backslash n \rfloor \rightarrow LAYOUT \\
&\sim \lfloor \backslash \wedge \backslash t \backslash n \rfloor^+ \rightarrow Water \{avoid\} \\
&A-Z[A-Z0-9]^* \rightarrow Id
\end{align*}
\]
context free syntax
\[
\begin{align*}
&Chunk^* \rightarrow Input \\
&Water \rightarrow Chunk \{cons(Call)\} \\
&“CALL” Id \rightarrow Chunk \{cons(Call)\} \\
&“MOVE” Id “TO” “CALLEE” \rightarrow Chunk \{cons(Call)\} \\
&“CALL” “HANDLER” \rightarrow Chunk \{reject\}
\end{align*}
\]

Figure 2.3: An island grammar in SDF [14]

define program
[repeat chunk]
end define

define call
CALL [not,’HANDLER’][id]
| MOVE [id] TO CALLEE
end define

define chunk
[call] | [water]
end define

define water
[key] | [token]
end define

Figure 2.4: An island grammar in TXL
achieved with the built-in [not] lookahead nonterminal, which fails and backtracks if the next token is HANDLER.

The following section explains how these directives and ordering semantics are used as an engineering solution to approximate context-sensitivity, and how Boolean grammars have this extra expressiveness built in.

2.4 Related Work

There is a body of work that already exists in this field. The most relevant are parsers based directly on the Boolean grammar formalism [19, 20, 13], and context-free parsers that extend their expressiveness through an engineered solution rather than through the characteristics of a formal grammar [9, 11, 4].

The ASF+SDF meta-environment and TXL both extend the power of their parsers – SDF with six explicit disambiguation constructs (follow, reject, prefer, avoid, associativity, precedence), and TXL with an implicit ordering of alternatives in a production. Boolean grammars, on the other hand, have this power baked into its mathematical formalism making these types of parser extensions unnecessary.

Agile parsing [6] and island grammars [26] have evolved around these context-free parsers, and involve altering a grammar to view the input in a certain way. Though we call these alterations parse views, the idea is already well established in the literature.
2.4.1 Parsers over Boolean Grammars

A generalized LR parsing algorithm, also known as a Tomita parser [24], was implemented by Okhotin for Boolean grammars [19]. Tomita’s original algorithm generalizes well to Boolean grammars, including the use of a graph-structured stack to simultaneously follow both possibilities of a shift-reduce conflict. The algorithm, as adapted by Okhotin, supports (1) conjunction by allowing reductions to have multiple premises instead of just one, and (2) negation by adding an invalidate operation that undoes a prior reduction operation. This is an example of a bottom-up parser, which differs considerably from our top-down implementation.

Okhotin also described a top-down (also known as recursive descent) parsing algorithm for Boolean grammars that is much closer to our own [20]. He implemented conjunction in a similar way to our parser – by using the input consumed by a parse of the first positive conjunct to verify the validity of consecutive conjuncts. Negation is implemented using exception handling – parse errors in subprocedures raise exceptions which propagate up the call stack until caught by an exception handler. In this way exception handlers implement negation, because they can catch a parse error thrown in a lower subprocedure (subgoal) and interpret it as a successful parse in a higher procedure (goal) where the negation is defined. This implementation is deterministic/predictive because the next rule to use is determined by a bounded lookahead. In contrast, our implementation supports full backtracking and unbounded lookahead.

Megacz provides an implementation of a Boolean grammar parser called SBP (Scannerless Boolean Parser) [13]. As the name suggests, his parser is scannerless meaning it works directly at the character level without a lexer. He demonstrates how a Boolean grammar can resolve the “dangling else” problem, as well as enforce
properly indented blocks in whitespace-sensitive languages like Python. Megacz’s syntax choice for grammar operators is similar to ours (introduced in Section 4.1), including a $\&$ operator for conjunction and $\&\neg$ for conjunction plus negation.

Megacz also maps the six disambiguation constructs from SDF and the implicit ordered-choice from TXL to their equivalent Boolean grammar representation [13]. In the following section we discuss how modern context-free based parsers can approximate the expressiveness of Boolean grammars with an engineered approach.

### 2.4.2 Parsers over Context-free Grammars

The negation operator in Boolean grammars is used to selectively restrict the valid forms of a production. Modern context-free based meta-programming languages have a similar mechanism to restrict a particular production from being applied. When using the Syntax Definition Formalism [9] (SDF), such as in the ASF+SDF meta-environment, one can mark productions as reject to force a failure when that production can normally be applied. RASCAL, a meta-programming language that supports automatic parser generation from a syntax definition, uses the Reserve disambiguation symbol ($\backslash$) that serves the same purpose and is typically used for keyword reservation [1]. The TXL language uses the [not X] nonterminal when the next token to be read cannot be parsed as X.

Modern parsers based on context-free grammars cannot simulate conjunction as easily as negation. Nevertheless, the multiple parse trees captured by a Boolean parse with conjunction are similar to the parse forests produced by a generalized LR parser. Tomita describes a shared packed parse forest (SPPF) representation, a directed acyclic graph (DAG) to share subtrees in an ambiguous parse [24]. Although the SPPF from a generalized LR parse and the parse DAG produced from conjunction...
in a Boolean grammar both use DAG structures, they differ in two significant ways.

First, an SPPF DAG can have multiple edges entering an internal node, whereas a Boolean parse DAG cannot. Boolean parse DAGs can only merge at the leaves (terminals), not at internal nodes (nonterminals). This is because SPPFs are designed to maximally share subtrees for efficiency. Boolean parse DAGs are, therefore, a more constrained form of directed acyclic graph compared to SPPFs.

Second, an SPPF is intended to efficiently store multiple ambiguous parses, meaning each forest of subtrees that captures a local ambiguity has the same root type, the same leaves, but a different tree structure. The roots of these subtrees are merged into a single packing node whose children are the subtrees representing each possible parse. The parses of positive conjuncts in a Boolean grammar are also required to share the same leaves, but the root types can be different.

The purpose of conjunction in Boolean grammars is primarily to improve language expressiveness, not to deal with ambiguity. In contrast, the purpose of SPPFs are to help deal with ambiguity in context-free grammars, and in particular manage the exponential growth of possible parses for ambiguous grammars. At a high level, Boolean and conjunctive grammars address a theoretic concern whereas shared packed parse forests address an engineering concern.

The ASF+SDF meta-environment uses generalized LR parsing to first build a forest of all possible parse trees, then uses user-defined disambiguation directives to prune the forest until a single parse tree remains. By selectively activating or deactivating these directives the user can control which parse, and thus which view, is produced by the generalized LR parser.
2.5 Summary

In this chapter we have introduced Boolean grammars, TXL, and parse views to inform the details of the following chapters. In Chapter 3 we give an overview of our approach and describe the design considerations when applying a Boolean grammar to TXL. In Chapters 4 and 5 we discuss the TXL implementation details of the Boolean parser and transformer, respectively. In Chapter 6 we show examples of the benefits of source transformation with Boolean grammars and parse views.
Chapter 3

Overview

A concrete software tool is needed to explore the idea and implications of parsing and transforming programming languages with Boolean grammars [16]. We chose TXL [4] for this purpose because of its maturity as a parsing/transformation tool, and for practical reasons such as the availability of its source code and knowledge of its inner workings.

This choice influenced many decisions for the Boolean grammar enhancements: grammar syntax, parsing semantics, internal representation of parse DAG, and transformation semantics. We strove to make these harmonious to the existing TXL style and philosophy, and therefore the overall approach closely resembles that of TXL as described in Section 2.2.

Figure 3.1: The enhanced TXL transformation pipeline
Figure 3.1 shows the transformation pipeline of TXL enhanced with Boolean grammars. A Boolean grammar is provided to the parser module, producing a parse DAG. The parse DAG is then passed to the transformation module which uses transformation rules to restructure the DAG. The parser and transformer modules are described in more detail in the following sections.

### 3.1 Parser Module

A significant contribution of this research is a top-down, backtracking parser that supports Boolean grammars. The standard TXL parser takes a context-free grammar and a text document as input, producing a parse tree as output. In contrast, the enhanced TXL parser takes a Boolean grammar and text as input, producing a parse DAG as output. Figure 3.2 gives an example of these differences by showing how different grammars parse a mathematical expression.

When interpreting a mathematical expression without parentheses, such as $5 + 3 \times 7$, most programming languages assign different precedence levels to the various operators in accordance with common mathematical conventions (e.g. $3 \times 7$ takes precedence over $5 + 3$, giving the result 26). However, some languages evaluate such expressions from left-to-right (e.g. Smalltalk, giving 56) or right-to-left (APL). One of these interpretations can be captured in a parse tree of the expression by specially crafting the context-free grammar used to parse it. Boolean grammars, however, have the ability to capture all interpretations simultaneously.

Figure 3.2(a) and (b) show context-free grammars for the left-to-right and precedence interpretations respectively, along with their parse trees of expression $5 + 3 \times 7$. Context-free grammars cannot capture both interpretations at the same time, so
Figure 3.2: Grammars and their corresponding parse of the math expression 5+3*7 for (a) CFG with a left-to-right parse, (b) CFG with a typical precedence parse, and (c) Boolean grammar that incorporates both (a) and (b). The parse of a Boolean grammar can be a DAG, unlike for CFGs which must be trees.
grammar authors are often forced to choose one even if they might like both. Figure 3.2(c) shows how both these interpretations can be captured simultaneously in a Boolean grammar, and the resulting parse. Unlike a context-free parser which produces a tree, a Boolean parser can output a directed acyclic graph whose leaves are shared.

3.2 Transformer Module

The transformation phase follows the parsing phase in the TXL process. The standard TXL transformer takes parse trees as input, but as Figure 3.2(c) shows, this transformer module must be able to handle parse DAGs produced by a Boolean grammar.

The purpose of the enhanced TXL transformer module remains unchanged: to transform the parse using a series of rewriting steps while keeping the integrity of the parse structure intact along the way. The high level objective is to be able to write a transformation rule that targets a specific parse view in the grammar and have the engine automatically consolidate any changes within that targeted view with the other views in the grammar.

Consider, for example, an HTML document that contains marked up English prose. There are two ways to view/parse this document: as HTML or as English. An HTML parser does not care about the English text, it only cares about making sure all tags are closed and nested correctly. Conversely, the English parser cares only about the structure of the English sentences and not about the interspersed HTML. A Boolean grammar can capture both these views simultaneously with a top-level production like $\text{Program} \rightarrow \text{EnglishView \& HTMLView}$. 
A rule to transform the markup part is easiest to write using HTMLView, whereas a rule to transform English sentences is easiest to write using EnglishView. In current transformation systems, if one wishes to transform both HTML and English in the same document they must parse the document using a unified grammar with both language elements supporting all possible configurations.

Figure 3.3(a) shows a greatly simplified English grammar, able to describe sentences containing only adjectives, nouns and verbs. This grammar can be changed to allow arbitrary HTML markup by replacing nonterminals that represent a word (\(A\), \(N\), and \(V\)) with versions surrounded by optional HTML tags (see Figure 3.3(b)).

Likewise, an HTML grammar can focus on the nested structure of HTML tags
and ignore the structure of text between them:

\[ HTMLView \rightarrow HTML \]

\[ HTML \rightarrow Word^* \text{Markup?Word}^* \]

\[ Markup \rightarrow < Id > HTML < /Id > \]

An example of where these two views intersect is in marked up English prose, for instance \texttt{<QUOTE>Good boys \textit{deserve} fun</QUOTE>}. Figure 3.4 shows how this sentence is parsed by both the HTML focused grammar (above) and the English focused grammar (Figure 3.3(b)).

To parse such a string the elements of each sublanguage are necessarily tangled together, eliminating the benefits of view-specific transformations. Our transformer module aims to reestablish these benefits by allowing rules to target a specific view, either HTMLView or EnglishView, in the Boolean grammar.

Figure 3.5 shows an example of a TXL program that changes all \texttt{<EM>} tags in a document to \texttt{<STRONG>} tags. Note how the specific HTML view is targeted in the \texttt{main} function, which allows the \texttt{emToStrong} rule to ignore the details between HTML tags and use \texttt{[repeat word]} as a catch-all.

Figure 3.6 shows an example of a transformation with the counterpart English view, to change all instances of the adjective “close” to “nearby”. This transformation, applied to the string \texttt{close \textit{countries} close borders}, yields \texttt{nearby \textit{countries} close borders}. There is no confusion between the adjective “close” and the verb “close” because the transformation occurs in the context of the English view, which parses the input into nouns, verbs, and adjectives.
Figure 3.4: A Boolean grammar parse of the string `<QUOTE>Good boys <EM>deserve</EM> fun</QUOTE>` using an English focused parse tree and an HTML focused parse tree (inverted). Both parse trees share the same leaves.
define program
    [html_view] & [english_view]
end define

% ... other defines here

function main
    replace [program] % type that defines the views
    HTMLView [html_view] % target a specific view
    by
        HTMLView [emToStrong]
end function

rule emToStrong
    replace [markup]
        <EM> Words [repeat word] </EM>
    by
        <STRONG> Words </STRONG>
end rule

Figure 3.5: A TXL program to change all <EM> HTML markup to <STRONG>, using a Boolean grammar and HTML-specific view.
define program
    [html_view] & [english_view]
end define

% ... other defines here

function main
    replace [program] % type that defines the views
        EnglishView [english_view] % target a specific view
    by
        EnglishView [closeToNearby]
end function

rule closeToNearby
    replace [A] % target adjectives only
        close
    by
        nearby
end rule

Figure 3.6: A TXL program to change all instances of the adjective “close”
to “nearby”, using a Boolean grammar and English-specific view.

This allows only adjectives to be targeted by the closeToNearby rule, something not
possible using a text-based find-and-replace tool, nor using the HTMLView parse.

3.3 Boolean Transformation Semantics

The multi-phase architecture of the TXL engine, shown in Figure 2.2, means that
the parsing step is very loosely coupled to the transformation step. This loose cou-
pling allows for the two main software objectives of this thesis, a Boolean parser and
Boolean transformer, to be approached as two separate problems. While we expected
a Boolean grammar based parser could be integrated in TXL, we were more skeptical
that the TXL transformer could be adapted to operate on parse DAGs. The loose coupling between parser and transformer allowed us to envision three steps ranging from low impact/high certainty to high impact/low certainty:

1. Boolean grammar is enforced on initial parse, but only one parse view is kept and the others are thrown away. The parser emits a tree structure and the transformer is unchanged. The input must conform to the grammar but the output does not.

2. Same as 1, but the final output is reparsed by the Boolean grammar. This ensures the output conforms to the Boolean grammar, but only provides a relatively unhelpful syntax error that does not indicate where in the transformation process the invalid form was introduced.

3. Boolean grammar is enforced on initial parse and all parse views are kept. The parser emits a DAG structure that the transformer must know how to consume. This case can be subdivided based on transformer semantics, discussed in detail in Section 5.1 and in particular the four possibilities on page 53.

An implementation strategy to integrate Boolean grammars into an existing tool can use this road map to slowly introduce features while still having a usable end-to-end tool chain. This flexibility was desirable since we could not foresee all the corner cases of a fully fledged Boolean grammar supported transformation engine.

We use two techniques, often complementary, to implement Boolean grammar support in TXL: source code modification and meta-transformation. The Boolean parsing phase is completely implemented by modifying the parser’s source code,
while the transformation phase uses a mix of source code modification and meta-transformations. The advantage of source code modification is flexibility; one has direct access to manipulate data structures and implement new semantics. The advantage of meta-transformation is rapid prototyping and well understood semantics.

3.4 Summary

In this chapter we apply the features of Boolean grammars to TXL. We explain the idea of parse views with an HTML markup example and describe some of the considerations encountered modifying the parser and transformer modules. The following two chapters elaborate on the implementation details of the enhanced parser and transformer, and in particular their semantics. Chapter 6 gives more examples, similar to the HTML markup transformation, where parse views benefit practical programming language source transformations.
Chapter 4

A Practical Boolean Parser

This chapter describes the process of applying Okhotin’s theoretical formalism of Boolean grammars [16] to the parser implementation of TXL [4]. Section 4.1 outlines the new operators added to support conjunction and negation and the design rationale behind them. Section 4.2 details the top-down, backtracking parsing algorithm to handle conjunction and negation. Section 4.3 explores the benefits of having a Boolean parser alone, even without a complementary Boolean implementation for the transformer.

4.1 New Grammar Syntax

Backward compatibility is a significant design goal when implementing the Boolean grammar enhancements to TXL. To achieve this goal, syntax changes are kept to a minimum and most of the enhancements are to the backend/engine. Every program written in standard TXL is a valid program in enhanced TXL, with the exception of the ampersand character (&) in productions and patterns, which now must be quoted...
when referring to the literal character.

The TXL language uses the vertical bar (|) operator to express disjunction in a grammar production. We implement two new operators to represent the Boolean grammar concepts of conjunction and negation: *and* (&), and *and-not* (&!). All three operators and their meanings are summarized in Table 4.1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example Use</th>
<th>Equivalent Language Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>or (</td>
<td>)</td>
<td>define A [B]</td>
</tr>
<tr>
<td>and (&amp;)</td>
<td>define A [B] &amp; [C] end define</td>
<td>( L(A) = L(B) \cap L(C) )</td>
</tr>
<tr>
<td>and-not (&amp;!)</td>
<td>define A [B] &amp;! [C] end define</td>
<td>( L(A) = L(B) \cap \overline{L(C)} )</td>
</tr>
</tbody>
</table>

Table 4.1: The TXL Boolean grammar operators and their meaning

There is a slight mismatch between the three boolean operators present in the formal description of a Boolean grammar (disjunction, conjunction, negation) and the three operators shown in the table above (or, and, and-not). When the negation operator appears in a conjunctive phrase, it applies only to an entire conjunct. For example, the phrase \( B \& \neg CD \) means \( B \& \neg(\neg CD) \), and a phrase such as \( B \& C \neg D \) is not possible. The & operator indicates the following operand is a positive conjunct and the &! operator indicates the following operand is a negative conjunct.

One implication of this syntax is that the first conjunct must be positive, so while Boolean grammars allow productions like \( A \rightarrow \neg B \) there is no way to directly express this in the enhanced syntax of TXL. However, in practice this is not a significant
restriction because typically one wants the input form to be constrained in some way, which can be used as the initial positive conjunct.

Another restriction on the TXL syntax is that define statements cannot mix disjunctive and conjunctive forms. This limitation exists only to simplify our parser implementation and not for any deeper reason. Therefore the production \( A \to B \mid C&D \) must be expressed as \( A \to B \mid A2 \) and \( A2 \to C&D \). Negative and positive conjuncts, however, can be mixed in any way within a conjunctive production as long as the first conjunct is positive.

### 4.2 Parsing Semantics

Recall from the previous section that the first conjunct in a conjunctive define statement must be positive. This fact is used by the parser to demarcate the segment of input text that must be matched by the other conjuncts. Consider the general parsing case of a conjunctive production \( A \to \alpha_1\&\alpha_2\&\ldots\&\alpha_n \) when the parser has already consumed \( t \) tokens of the input stream. Conjunct \( \alpha_1 \) must be positive while conjuncts \( \alpha_2 \ldots \alpha_n \) can be either positive or negative. Algorithm 1 describes how to parse goal \( A \) in this situation, and can be summarized as “parse the first conjunct, then verify the others.” If the goal fails, the parse needs to backtrack and retry. If the goal succeeds, the tokens for the parse of \( A \) are consumed and the parse proceeds with the next goal.

Omitted from Algorithm 1 is the process of building the parse DAG. Whenever the input is successfully parsed as a positive conjunct, its parse tree must be remembered and linked into the resulting parse for the overall goal. We are guaranteed that all positive conjuncts will share the same leaves because parsing the first conjunct sets
if a portion of the remaining input can be parsed as a $\alpha_1$ then
    $k \leftarrow$ number of input tokens consumed by $\alpha_1$ parse;
    for $i = 2 \ldots n$ do
        if $\text{parsable} \leftarrow$ input substring from $t + 1$ to $t + k$ can be parsed as a $\alpha_i$;
        if $\text{parsable} \text{ xor } \alpha_i \text{ is positive conjunct}$ then
            /* negative conjunct succeeded to parse or */
            /* positive conjunct failed to parse */
            Goal to parse as an $A$ fails.
        end
    end
    Goal to parse as an $A$ succeeds.
else
    Goal to parse as an $A$ fails.
end

Algorithm 1: How the general conjunction goal $A \rightarrow \alpha_1 \& \alpha_2 \& \ldots \& \alpha_n$ succeeds or fails when $t$ tokens of input have been read.

the boundaries of the input to check the subsequent conjuncts. Negative conjuncts do not have a corresponding parse tree, but in the TXL implementation they are represented in the parse DAG with a special placeholder node. This is done for implementation convenience so the children of conjunction nonterminal $X$ in the grammar tree correspond one-to-one (same number and order) with the children of a conjunction node of type $X$ in the parse DAG.

The parse DAG data structure is implemented as a tree with additional constraints enforced in the parser. These constraints ensure the leaves of all subtrees representing positive conjuncts under a conjunctive node are the same. In this way we can treat the resulting data structure as a parse DAG logically even though it is a tree structurally.

The semantics for backtracking over a conjunction parse depends mostly on its first conjunct. The first conjunct, $\alpha_1$, is the only one that can yield a successful parse with just a portion of the remaining input stream. The other conjuncts, $\alpha_2, \ldots, \alpha_n$, must accept (for positive) or reject (for negative) the same fixed string and not just
some prefix thereof. Conjuncts $\alpha_2, \ldots, \alpha_n$ do not have the option of changing the number of consumed tokens when searching for an alternate parse, so we simply throw them away and instead backtrack over the first conjunct. If an alternate parse of the first conjunct is found, the other conjuncts $\alpha_2, \ldots, \alpha_n$ are again checked against the leaves of this alternate parse just as they were in Algorithm 1.

Table 4.2 traces the backtracking process of conjunction in a simple grammar, emphasizing the importance of the first conjunct. In this example the parser tries to solve the conjunctive goal $X$ by first trying $ababa$, then backtracking to $aba$, then backtracking to $a$. These strings are chosen because the first conjunct of $X$ is $P$, a greedy sublanguage for non-empty palindromes. $P$ is greedy in the sense that it matches the longest palindrome first, which is why the entire input $ababa$ is consumed right away. Backtracking causes shorter palindromes – $aba$ and $a$ – to be subsequently selected.

Consider a grammar identical to that in Figure 4.1 except that the first and second conjuncts of production 4.2 are swapped, yielding the conjunctive production $X \rightarrow B_0 \& P \& \neg B_1$. Although this modified grammar is semantically identical to the original, the trace will be significantly different because the first conjunct has changed. The first conjunct is now $B_0$, representing a greedy language for any string with zero or one $b$’s. Therefore, the strings tried during the course of backtracking will be $aba$ first, then $ab$, then $a$. $aba$ fails because it contains exactly one $b$ ($B_1 \Rightarrow aba$) and $ab$ fails because it is not a palindrome ($B_1 \not\Rightarrow aba$). No more backtracking is needed after $a$ is tried because $X \Rightarrow a$ and $BA \Rightarrow baba$, satisfying the top-level goal $S \rightarrow X BA$. 
$S \rightarrow X BA$  \hspace{1cm} (4.1)

$X \rightarrow P \& B0 \& \neg B1$ \hspace{1cm} (4.2)

$P \rightarrow aPa \mid bPb \mid a \mid b$ \hspace{1cm} non-empty palindrome \hspace{1cm} (4.3)

$B0 \rightarrow a^*b?a^*$ \hspace{1cm} any string with zero or one b’s \hspace{1cm} (4.4)

$B1 \rightarrow a^*ba^*$ \hspace{1cm} any string with exactly one b \hspace{1cm} (4.5)

$BA \rightarrow ba BA \mid \epsilon$ \hspace{1cm} any repetition of ‘ba’ \hspace{1cm} (4.6)

Figure 4.1: A Boolean grammar with annotated sublanguages.

Table 4.2: A backtracking parse trace of the string ababa using the grammar in Figure 4.1. The details of the trace unrelated to conjunction have been omitted for clarity.


```verbatim
define var_declaration
    var [varnames_of_types] := [value+];
& var [var_name+] of [types_assign_values];
end define

define varnames_of_types
    [var_name] [varnames_of_types] [type] | of
end define

define types_assign_values
    [type] [types_assign_values] [value] | :=
end define
```

Figure 4.2: A Boolean grammar to ensure the number of variables, types, and values are equal in a variable declaration statement.

### 4.3 Expressiveness Advantages of Boolean Parser

The two major goals of this thesis are a practical parser based on Boolean grammars, and a transformer for the resulting parse DAG. The Boolean parser is not just a prerequisite for the transformer; it provides tangible benefits of its own. These benefits derive from the provably larger class of languages expressible by Boolean grammars as compared to context-free grammars [16].

Consider a programming language with syntax to declare, specify the type of, and initialize multiple variables at once:

```verbatim
var shape x y r of string int int int := 'circle' 2 -1 5;
```

The number of variables, types, and values must all be consistent in such a statement, but this constraint is impossible for a context-free grammar to capture because it reduces to the non-context free language \( \{a^n b^n c^n \mid n > 0\} \). This constraint can, however, be expressed in TXL with Boolean grammar support, as shown in Figure 4.2.

The `varnames_of_types` definition ensures the number of variables matches the
number of types, and the `types_assign_values` definition ensures the number of types matches the number of values. By taking the intersection of these sublanguages using the conjunction operator (&) we can ensure that both constraints are upheld simultaneously, and thus the number of variables, types, and values are equal.

The grammatical constraints imposed by this enhanced expressiveness has implications for coupled transformations. According to Cunha and Visser, “coupled transformations occur in software evolution when multiple artifacts must be modified in such a way that they remain consistent with each other” [5].

For example, suppose one wants to transform the variable declaration statement mentioned above by adding or removing variables. Such a transformation must ensure that all three parts of the declaration statement (variable name, type, initial value) are consistent. A context-free grammar is not expressive enough to capture this constraint, so cannot verify consistency after the transformation. A Boolean grammar can, however, express this constraint, and cause such an erroneous transformation to fail because it invalidates the grammar.

### 4.4 Parsing Efficiency

The addition of conjunction and negation operators significantly changes the parsing process in TXL. In this section we explore whether these changes have an effect on the runtime efficiency of this parsing process, and do so by examining the consequence of each Boolean grammar operator. We start by showing negation has no effect on the average runtime. We then show that both conjunction and disjunction are, in some sense, two sides of the same coin and thus both are equivalent in their average runtimes.
In the grammar syntax of our enhanced TXL, negation can occur only in a conjunctive production after an *and* (&) operator, and is used to flag the following conjunct as a negative conjunct. This flag refers to the subexpression “α_i is positive conjunct” after the xor in Algorithm 1 on page 40. This subexpression is evaluated after the conjunct parse is attempted (stored in the variable parsable in Algorithm 1), so whether a conjunct is negative or positive has no bearing on whether that conjunct’s parse is attempted or not. Without negation, the inner if condition of Algorithm 1 becomes “parsable” instead of “parsable xor α_i is positive conjunct”. The extra xor operation represents a miniscule constant time increase, which does not change the average runtime efficiency.

Conjunction affects parsing in a more significant way. The for loop in Algorithm 1 iterates over the conjuncts in a conjunctive production so, in the worst case, all conjuncts must be attempted to be parsed. This worst case scenario can be short-circuited by finding a positive conjunct that does not parse or a negative conjunct that does.

A similar situation exists for disjunction where, in the worst case, all alternatives must be tried when finding an appropriate parse, but when one is found the others need not be tried. Conjunction and disjunction are, in fact, most and least efficient in opposite circumstances as shown in Table 4.3. This follows the standard short circuiting rules for && and || found in many C-like programming languages. The average case, exactly half of the conjuncts/disjuncts in the production need to be tried, represents the same runtime efficiency for both disjunction and conjunction.

Disjuncts and conjuncts are both ordered in TXL, so their respective parsing algorithms require them to be tried in the specified order. One way TXL programmers
<table>
<thead>
<tr>
<th>Efficiency</th>
<th>Disjuncts</th>
<th>Conjuncts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slowest</td>
<td>All fail</td>
<td>All succeed</td>
</tr>
<tr>
<td>Slow</td>
<td>Succeed late</td>
<td>Fail late</td>
</tr>
<tr>
<td>Fast</td>
<td>Succeed early</td>
<td>Fail early</td>
</tr>
<tr>
<td>Fastest</td>
<td>First succeeds</td>
<td>First fails</td>
</tr>
</tbody>
</table>

Table 4.3: The circumstances under which disjunction and conjunction are most and least efficient due to short circuiting.

can tune grammars to improve parsing efficiency is to place disjuncts that are more likely to succeed first, and ones that are less likely to succeed last. Likewise, conjunction parsing efficiency can be improved with the opposite approach: conjuncts that are more likely to fail are placed first, and those more likely to succeed are placed last. These tuning techniques are typically unnecessary and only valuable if the grammar causes excessive backtracking.

4.5 Summary

In this chapter we describe the implementation details of a top-down, fully backtracking parser for Boolean grammars. We introduce a new grammar syntax for conjunction and negation, discuss their parsing semantics, and show how these improvements alone can benefit grammar authors. The output of the Boolean parser passes to the Boolean transformer, which we describe in depth in the next chapter.
Chapter 5

A Practical Boolean Transformer

The conjunctive properties of Boolean grammars [16] can be used to view the same input text in different ways, similar to the agile parsing techniques introduced by Dean et al. [6] and described in Section 2.3.1. Cordy pointed out that cascaded transformations – multiple transformation passes on the input text – are often necessary to accomplish a complex transformation in TXL [4].

TXL makes these multiple passes easy if each pass uses the same view of the input, in fact TXL rule/function application is designed for exactly this scenario. If, however, the goals of each pass are sufficiently distinct, it is usually more convenient to use a view tailored for each pass. Without the benefits of Boolean grammars, there are two options in TXL:

1. **Use a single view for all passes.** This solution typically requires writing more complicated rules to deconstruct general grammar forms into ones suitable for the current task.

2. **Unparse back into text after each pass, then reparse with a new
grammar tailored for the next pass. This solution has the overhead of extra parse and unparses steps of the entire document between transformation passes.

The Boolean grammar enhancements to TXL are intended to solve this problem by making different views available without an intermediate reparse step between transformation passes. These enhancements do not remove the need for reparsing entirely, but rather perform reparsing during the transformation as necessary.

There are tradeoffs with each technique when counting the total reparses required. In our enhanced TXL, reparsing is localized to particular subtrees whereas with option 2 above the entire parse tree is unparsed and reparsed, even parts that do not require it. However, the downside of our approach is that runtime transformations trigger reparses which may not be necessary. We address possible ways to mitigate these concerns through optimizations (Section 5.4) and a proposal to temporarily delay reparsing (Section 7.1.1).

Each transformation pass on the input corresponds to a TXL rule or function, so these rules and functions need a way to declare which views to use. The view of a rule or function is determined by the target type and pattern expression in its replace clause:

```
replace [target_type]
   <pattern>
```

For example, there are two possible views for the conjunctive production $S \rightarrow A \& \neg B \& C$, namely $A$ and $C$. To target view $A$ in a TXL rule, the rule type must be $S$ and the pattern must parse as an $A$:

```
replace [S]
```
MyViewA [A]

and likewise for view C:

replace [S]
  MyViewC [C]

Since a rule pattern can be a decomposition of a single type it is possible, indeed likely, that a pattern can match multiple views of a conjunctive type. Recall the definition of var_declaration from Figure 4.2:

define var_declaration
  var [varnames_of_types] := [value+] ;
  & var [var_name+] of [types_assign_values] ;
end define

We can easily write a rule pattern that matches both views:

replace [var_declaration]
  var V1 [var_name] V2 [var_name] of int int := 0 0 ;

In such situations we follow the principle of least surprise to TXL programmers and use the first view that can parse the pattern. These semantics are consistent with the significance placed on the order of alternatives in a grammar production during parsing: the first alternative is tried first, then the second, etc.

### 5.1 Boolean Transformation Semantics

Recall from Section 2.2 that TXL transformations are guaranteed to keep the result well-formed because the output type of a rule or function must be the same as its input type. This is a fundamental characteristic of TXL, but also the most confounding factor when devising transformational semantics for Boolean forms.
$S \rightarrow a^*b^*c^* \& AB c^* \& a^* BC$

$AB \rightarrow a \ AB \ b \ | \ \epsilon$

$BC \rightarrow b \ BC \ c \ | \ \epsilon$

(a)

Figure 5.1: (a) Boolean grammar for the language $\{a^n b^n c^n \mid n \geq 0\}$ and (b) a parse of the string $aaabbbccc$ with the $AB c^*$ view and the $a^* BC$ view (inverted)

The complication comes about when a transformation changes the leaves of one branch of a conjunctive tree but not the other branches. It is possible that the new leaves are consistent with the other conjuncts, but there is no way to determine this a priori. When different branches are inconsistent the tree is no longer well-formed.

For example, Figure 5.1 shows a Boolean grammar for the language $\{a^n b^n c^n \mid n \geq 0\}$ with three views, and a parse of the string $aaabbbccc$ with the second and third views. The first view, $a^*b^*c^*$, is easiest to visualize and thus omitted in Figure 5.1(b) for simplicity. If a rule targets the view $a^*b^*c^*$, it can conceivably change the quantity of a’s, b’s, or c’s to anything as long as their order is maintained. A rule that changes
CHAPTER 5. A PRACTICAL BOOLEAN TRANSFORMER

51

Figure 5.2: (a) a well-formedness preserving rule, and (b) a rule that invalidates other views

these quantities consistently (e.g. remove_2 in Figure 5.2(a)) will not invalidate the other views. However, a rule can easily be constructed (e.g. remove_2a_and_1b in Figure 5.2(b)) that changes the quantities of each letter inconsistently, invalidating the other views which ensure equal counts of the letters.

There is no general algorithm to determine if, given a set of views and a transformation rule, the transformation rule will invalidate the views or not. Even if this analysis was possible in some cases, the problem is exacerbated by decomposing the rule into subrules and then applying those subrules sequentially. Consider a decomposition of the consistent rule remove_2 above into three subrules

and then reimplementing remove_2 by applying those subrules to the relevant parts of the \( a^*b^*c^* \) view:
rule remove_2
    replace [S]
    A['a*] B['b*] C['c*]
    by
    A[remove_2a] B[remove_2b] C[remove_2c]
end rule

This will compile and run just as it did before without invalidating the other views because all quantities are kept consistent in the replacement.

A complication arises, however, if we apply these same subrules sequentially on a scope representing the entire view. For this example, suppose we have a nonterminal named \(ABC\) to represent the first view (i.e. \(ABC \rightarrow a^*b^*c^*\))

rule remove_2
    replace [S]
    ABCView [ABC]
    by
    ABCView [remove_2a][remove_2b][remove_2c]
end rule

This rule returns a well-formed final result, but a malformed intermediate result, as demonstrated in Table 5.1.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Well-formed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCView</td>
<td>aaabbbccc</td>
<td>Yes</td>
</tr>
<tr>
<td>ABCView [remove_2a]</td>
<td>abbbccc</td>
<td>No</td>
</tr>
<tr>
<td>ABCView [remove_2a][remove_2b]</td>
<td>abccc</td>
<td>No</td>
</tr>
<tr>
<td>ABCView [remove_2a][remove_2b][remove_2c]</td>
<td>abc</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.1: The intermediate result of a transformation may be malformed even if its final result is well-formed

Allowing a malformed intermediate result can often simplify TXL rules which TXL programmers can take advantage of by slightly loosening the constraints of a grammar, a technique that uses grammar overrides as elaborated in Section 2.4 on agile
parsing. The “loosening” of conjunctive nonterminals already exists in a sense be-
cause each conjunct further constrains the valid forms of that nonterminal. Loosening
can, therefore, be achieved by deferring the enforcement of conjuncts. The decision
about when to defer the enforcement of conjuncts is key and we have considered four
possibilities, outlined below.

1. **Abandon well-formedness.** The output of rules and functions do not need
to have the same form as their input. This solution goes quite far against the
philosophy of TXL and therefore was not seriously considered for an implemen-
tation in TXL. It may be reasonable in an unconstrained term rewriting system
like ASF [25].

2. **Turn views into disjuncts.** This is the type of override that a TXL program-
ner would typically use to loosen a grammar. However, treating conjuncts as
alternatives fundamentally changes the meaning of the grammar, specifically
any conjunctive type is guaranteed to be one of its positive conjuncts but not
all them. The onus is on the programmer to ensure all views are consistent if
they want that.

3. **All patterns include all views.** A pattern would contain the & operator like
a Boolean grammar production and bind variables for each view (e.g. \texttt{ViewA [A] \& ViewB [B]}). The replacement would also have this format and describe
new values for each view: \texttt{NewA \& NewB}. This ensures results are well-formed
but defeats the point of views, namely that rules are easier to write because
they only consider a single view.

4. **Ongoing reparse.** Rules target a single view and when the rule completes
the result is parsed as the other conjuncts. If all reparses succeed then the rule
succeeds, but if any reparse fails then the rule silently fails and no changes are
made to the scope.

The abandon well-formedness strategy defers all constraints indefinitely, including
those typically enforced by TXL. The turn views into conjuncts strategy maintains
all TXL’s standard well-formedness guarantees, but for conjunctive types one view
is enforced at compile time while enforcement of the other conjuncts are deferred
indefinitely. The all patterns include all views strategy is the only one that
enforces all views at compile time, making it safe but restrictive from a practical
standpoint. Finally, the ongoing reparse strategy enforces a single view at compile
time and enforces the other views at runtime.

We settle on the ongoing reparse strategy because it strikes a good balance
between safety, by ensuring final and intermediates results are well-formed, and prac-
ticality, by allowing each rule to target a single view. The following section describes
this strategy in detail.

5.2 Ongoing Reparse Semantics

This strategy attempts to maintain a well-formed parse DAG by reparsing at runtime
whenever a change is made to a descendant of a conjunctive type. TXL has a built-in
function called [reparse] that can be used to dynamically reparse the leaves of any
subtree as a new type. The signature of this function is \( X_1 \) [reparse \( X_2 \)], which
means “reparse the leaves of \( X_2 \) as the type of \( X_1 \)” If the reparse fails TXL aborts
loudly with a fatal error. This is unfortunate for our purpose because we want to catch
and handle the case when a subtree is not parsable as another type. We therefore created a new built-in function, [parsable], with the same signature but instead returns true if $X_2$ is parsable as the type of $X_1$ and false otherwise.

The [reparse] and [parsable] functions can be used along with a meta-transformation to describe the semantics of Ongoing Reparse. A meta-transformation is a transformation of a transformation rule, and has been used in the past to implement or prototype TXL language enhancements [23]. The point is to recognize an invalid TXL code pattern that can be converted to valid regular TXL then executed by the existing TXL engine. The meta-transformation defines, precisely, the semantics of the new language feature in terms of existing TXL semantics.

The easiest way to visualize a source transformation is to compare the code snippet before and after transformation, and in the case of this meta-transformation the code snippet is a TXL function. Assuming a generic Boolean grammar production $S \rightarrow C_1 \& C_2 \& \neg C_3$ and some subrules $R_1$ and $R_2$, Figure 5.3 shows the TXL rule before and after the meta-transformation.

The first step in Figure 5.3(b) is to bind the variable AllViews. The sole point of binding this variable is so the [parsable] and [reparse] functions can use its type, $S$; the value of AllViews is never used. The desired view is then selected with a deconstruct clause and the original replacement is computed and stored in the variable Result. The where clause enforces the Boolean constraints by guarding the success of the function on the condition “Result is parsable as a $S$.” This ensures the original result is accepted by each positive conjunct and rejected by each negative conjunct. The [reparse Result] call does the actual reparsing and the final replacement is a tree of type $S$ including all its views.
function apply_R1_R2
replace [S]

% select desired view
View1 [C1]
by
View1 [R1][R2]
end function

(a) Before

function apply_R1_R2
replace [S]

% type of AllViews is [S]
AllViews [S]
deconstruct S
% select desired view
View1 [C1]
construct Result [C1]
View1 [R1][R2]
where
% true iff Result is
% parsable as a [S]
AllViews [parsable Result]
by
% guaranteed to succeed
AllViews [reparse Result]
end function

(b) After

Figure 5.3: A TXL rule directly targeting a conjunctive type, before and after its meta-transformation. Common lines have been aligned to better demonstrate the effect of the meta-transformation.
5.3 Direct vs. Indirect Conjunctive Targets

Figure 5.3 describes a meta-transformation for when a TXL rule or function directly targets a conjunctive type. The automatic search-and-replace semantics of TXL rules, in particular, allow them to also indirectly target conjunctive types. This occurs when the rule’s pattern matches a descendant of a conjunctive node but this conjunctive node is still within the scope of the rule application. We need not be concerned with conjunctive nodes outside this scope (i.e. higher up in the parse tree) because caller rules/functions handle this as the call stack unwinds.

Figure 5.4 shows both the direct and indirect case in a generic parse tree. In the indirect case, the search passes through a conjunctive node and matches one of its
descendants. However, replacing the matched node may invalidate the conjunction that was passed through earlier so this conjunction must be reparsed to ensure it is still well-formed.

As a concrete example, recall the grammar from Figure 5.1(a). A rule that directly targets conjunctive type $S$ requires a reparse of $S$ after a replacement. Any rule that targets a descendant of $S$, such as a node of type $AB$, also requires a reparse of $S$ because any leaves of one conjunct that changes requires a reparse of the other conjuncts to ensure they are all consistent. Note that it is possible for a conjunctive target to be both direct and indirect, for example if $AB$ was itself a conjunctive type and a rule targeted it.

We consider two ways to implement reparsing an indirectly targeted conjunctive node:

1. Change the TXL engine implementation for rule search, match, and replace to detect passing through a conjunction and automatically trigger a reparse upon replacement.

2. Use meta-transformations to reduce the problem to only directly targeted conjunctive types, for which we already have a solution.

We opt to use the latter approach because we cannot easily adapt TXL’s current implementation to the former. TXL’s search-match-replace algorithm does not match and replace at the same time as it searches through the parse tree, but rather performs it in two separate steps. The first step is an initial search from the root node that returns completely when a match is found. The second step applies the replacement at the subtree discovered by the first step. This two-step algorithm is difficult to adapt to automatic reparsing because by the time the replacement occurs, the context about
the rest of the parse tree is no longer available. This context is needed to determine how the replacement may invalidate conjuncts higher up in the parse tree.

In the meta-transformation approach, the indirect case is transformed into a direct case by intercepting execution when the conjunctive type is found by the search and applying a different action guaranteeing a reparse. Figure 5.5 shows an example of a meta-transformation using the grammar from Figure 5.1 where a TXL rule targets type $AB$ and indirectly targets conjunctive type $S$.

The meta-transformation splits the original rule into two cases: (1) matching the original target type, and (2) matching a conjunctive type; any calls to the original rule ($\text{Scope [add\_ab]}$) must now call two rules: $\text{Scope [add\_ab][add\_ab\_to\_S]}$. The original rule is repurposed for case 1, the only change being that it skips the conjunctive type handled by case 2.

A new rule which targets the conjunctive type is added for case 2, which acts similarly to the directly targeted meta-transformation shown in Figure 5.3(b). A notable difference, however, is construction of the $\text{Result}$ variable using the expression $\text{View1 [add\_ab][add\_ab\_to\_S]}$. This explicit recursive call is necessary to retain the implicit depth-first search semantics of the original rule.

There is no reason to prefer $\text{View1}$ over another view in this situation because the original rule in Figure 5.5(a) contains no mention of type $S$ or any of its views. This raises an open research question: Which view do we choose when propagating a rule through a conjunctive node in the parse tree?

A natural answer to this question is “propagate the rule through all of the views.” On the surface this seems consistent with the semantics of a TXL rule, but the complication arises when a rule transforms multiple views. Which view do we take
Figure 5.5: A TXL rule indirectly targeting conjunctive type $S$, before and after its meta-transformation
as the view, the one whose leaves will be reparsed as the conjunctive type? If one view changes and the others remain unchanged, then perhaps it makes sense to prefer the changed view. However, if all views change as a result of the rule then their leaves may all be different and there’s no reason to prefer one over another. So when propagating a rule through a conjunctive node, the question “Which view will be the sole view upon which the rule is applied?” is, in the most general case, equivalent to “After applying the rule to all views, which result is used for reparsing?”

We propose a resolution: use the first view that is changed by the rule. The rule is applied to each view, in order, until the rule causes a view to change in some way. The leaves of that view (positive conjunct) are then used as the string upon which the other conjuncts are reparsed and checked. If all other positive conjuncts succeed to reparse and all negative conjuncts fail to reparse then the rule succeeds and the changes are committed. Otherwise the rule fails and changes to all views are discarded.

Relying on the ordering of conjuncts in TXL is natural and mirrors the existing significance attached to the ordering of disjuncts (i.e. try the first and if there’s no match then try the next, then the next, etc.). There are three cases of interest when a rule is applied to a conjunctive type: no views change, one view changes, or multiple views change. The last case is the problematic one, which our proposal attempts to resolve. With our proposed resolution, the result of each case is:

1. If no views are changed then nothing changes and there is no conflict.

2. If changes occur in only one view then that view is used as the reparse candidate. Changes are either committed or discarded based on conjunct reparsing as described above.
3. If changes occur in multiple views, the first changed view is used as the reparse candidate. Changes are either committed or discarded based on conjunct reparsing as described above.

We believe this offers the most intuitive result overall. The downside is that if the programmer wishes for a rule to apply to a specific view in the current scope, they must explicitly deconstruct down to that view before applying the rule.

These semantics can also be used for similar situations such as the built-in TXL rule extract $[^*]$. The purpose of this rule is to retrieve all subtree instances of a particular type contained in the rule scope (e.g. retrieve all assignment statements inside an if branch). If the rule happens to traverse through a conjunctive type, a naive implementation would retrieve matching subtrees from each view. This is not ideal because each view represents a parse of the same text, causing more subtrees to be retrieved than should be. A more accurate solution is to search each view in order. If a view does not contain any matches then start searching the next view. If a view contains one or more matches, the matches for that view are retrieved and no more views are searched.

### 5.4 Optimizations

The following subsections describe compile-time, static analyses that determine the possibility of an event during execution, but cannot determine the actuality of that event occurring at runtime. In other words, these analyses can be used to eliminate impossible reparse or search candidates but cannot predict at compile time the exact path execution will take.
5.4.1 Only Reparse Potentially Used Views

TXL functions and rules that target conjunctive types can deconstruct that type as a specific view, determinable at compile time. Likewise, a call graph of the program’s rules and functions can also be determined at compile time. The amount of ongoing reparsing needed can be optimized by combining these separate static analyses and only reparsing views used by later rules or functions.

For example, suppose two functions $f_1$ and $f_2$ are applied consecutively to the same scope: `Scope [f1][f2]`. $f_1$ and $f_2$ both target the conjunctive type $S \rightarrow C_1 \& C_2 \& C_3$ and $f_2$ uses the $C_2$ view. Normally when $f_1$ succeeds, its replacement view must be reparsed as an $S$ thereby validating its views $C_1$, $C_2$, and $C_3$. However, most of this reparsing is redundant because we know this same reparse will happen again after the call to $f_2$. The only reparse necessary after $f_1$ is for the view that $f_2$ uses, namely $C_2$.

5.4.2 Indirect Conjunctive Target

The meta-transformation in Figure 5.5 contains a conjunctive type $S$ in the result that was never mentioned in the original. In fact, a naive implementation needs to do this meta-transformation for every conjunctive type in the grammar because the automatic search-and-replace semantics of TXL rules can encounter any conjunctive type in the subtrees of the rule’s scope.

We can improve on this by recognizing that not all conjunctive types are reachable by every rule call. Since a rule application can only change the part of the parse tree upon which it is called (i.e. its scope), a static analysis of the relationships between grammar nonterminals can determine which nonterminals may or may not fall within
that scope.

The relationships between nonterminals in a grammar can be represented as a directed graph, with nonterminals as the vertices and productions determining the edges. Specifically, every unique pair of nonterminals \((A, B)\) on the left and right-hand side of grammar productions (i.e. \(A \rightarrow \ldots B \ldots\)) becomes a directed edge from \(A\) to \(B\) in the graph.

Now the question “Is conjunctive type \(S\) reachable by a rule applied to a scope of type \(T\)?” can be rephrased as “Is vertex \(S\) reachable from vertex \(T\)?” If the answer is “no”, we can skip the meta-transformation for indirectly target conjunctive type \(S\).

### 5.5 Summary

In this chapter we discuss possible transformation and reparse semantics for parse DAGs. We identify the more difficult research questions of this thesis and our efforts to address them. The next chapter gives examples of how to use Boolean grammars and transformations in practice, and talks about their benefits over traditional context-free grammar based solutions.
Chapter 6

Applications

The purpose of this chapter is to show the effect of using a Boolean grammar supported tool, like our enhanced TXL implementation, on typical source code transformation tasks. The examples in this chapter are taken directly from or inspired by Dean et al.'s paper on agile parsing [6]. The first example, in Section 6.1, shares the same motivation described by Dean et al. on transforming precedence grammars:

The COBOL base grammar has the normal multiple levels of precedence in the expression grammar, and the nonterminal [statement] derives all of the different statements. Without grammar modification, we must write separate rules that target each level of precedence and each type of arithmetic statement (e.g. ADD statement). By modifying the grammar, we can significantly reduce the number of rules needed to extract the information.
Section 6.2 shows the second example, an agile parsing technique called *grammar specialization*, which involves gradually refining a parse of variable uses and modifications with intermediate nonterminals to specialize them, allowing for more targeted transformation rules.

The final example, in Section 6.3, is an instance of an agile parsing idiom called *grammar categorization*: “Grammar categorization refers to modification of the grammar to make finer distinctions appropriate to the task at hand.” [6] This technique exploits the order-sensitive parsing of alternatives in TXL [4] to prefer a certain parse, but falls through to a default parse in the general case.

Cordy reminds us that large scale transformations in TXL typically require multiple stages of reparsing to take advantage of different parse views [4]:

In order to organize large scale transformations and make them easier to develop and maintain, TXL programmers normally separate them into a cascaded set of separate TXL programs, each of which attacks only one issue or set of issues. The output of each stage returns to a text file in the hybrid language which is then reparsed using the overridden grammar to the next stage.

Our transformation system based on Boolean grammars [16] reformulates this process by capturing all required parse views in the initial parse then reparsing smaller sections of the parse tree as necessary while the transformation runs.
\[
\begin{align*}
Expr &\rightarrow Expr \ BinaryOp \ Expr \mid - \ Expr \mid Literal \mid Id \mid (\ Expr ) \\
BinaryOp &\rightarrow \text{and} \mid \text{or} \mid \text{xor} \mid < \mid > \mid <= \mid >= \mid == \mid + \mid - \mid * \mid /
\end{align*}
\]

(a)

\[
\begin{align*}
Expr &\rightarrow Expr \ BoolOp \ Comp \mid Comp \\
Comp &\rightarrow Comp \ CompOp \ Sum \mid Sum \\
Sum &\rightarrow Sum \ AddOp \ Term \mid Term \\
Term &\rightarrow Term \ MultOp \ Factor \mid Factor \\
Factor &\rightarrow UnOp? \ Factor \mid Primary \\
Primary &\rightarrow Literal \mid Id \mid (\ Expr ) \\
BoolOp &\rightarrow \text{and} \mid \text{or} \mid \text{xor} \\
CompOp &\rightarrow < \mid > \mid <= \mid >= \mid == \\
AddOp &\rightarrow + \mid - \\
MultOp &\rightarrow * \mid / \\
UnOp &\rightarrow -
\end{align*}
\]

(b)

Figure 6.1: (a) A simple but highly ambiguous expression grammar, and (b) its unambiguous precedence version. Both grammars generate the same language.

6.1 Transforming a Precedence Grammar

One of the main benefits of Boolean grammars is the ability to craft parse views specific to a particular transformation task, greatly simplifying the transformation code. We demonstrate this idea with a type analysis transformation on a language whose expressions are described with a precedence grammar.

A precedence grammar is a carefully crafted unambiguous grammar that forces an expression to parse in a way that captures the precedence and associativity of its operators. Figure 6.1(b) shows a precedence grammar for the expression language
more naturally described by the grammar in Figure 6.1(a). Indeed, the first production of Figure 6.1(a) reads like a natural language specification: “an expression is two expressions separated by a binary operator or a negated expression or a literal or an identifier or a parenthesized expression.”

Some programming languages have expressions which can be described by an extensive precedence grammar, such as the C programming language which has 15 precedence levels [10]. Without some technique to capture parse views like agile parsing, source transformation tasks on such programs are forced to use a precedence grammar even if it is more convenient to use a different grammar representation. We demonstrate the benefit of parse views in transformations with two examples. The first, a task to parenthesize subexpressions, is elaborated in Section 6.1.1. The second, an analysis of type consistency, is explained in Section 6.1.2.

6.1.1 Parenthesize Subexpressions Example

We demonstrate the benefit of parse views with a transformation task to parenthesize subexpressions, for example to turn the expression \( a > 1 \) or \( a \leq 5 \) into \(((a > 1) \text{ or } (a \leq 5))\). Literals and identifiers should not be parenthesized, to avoid subexpressions like \((a)\) and \((5)\). The current way to accomplish this task in TXL is shown in Figure 6.2, using a separate rule for each precedence level. The way these rules are written have several downsides:

1. The rules contain a lot of duplication and redundancy

2. The rules obscure the high-level intent of the transformation

3. Any change to the precedence grammar such as reordering/adding/removing
precedence levels requires a corresponding change to the transformation

A Boolean grammar simplifies this transformation by allowing each precedence level to be viewed as a top-level expression. Figure 6.3 shows the necessary grammar overrides and single rule to accomplish this. The deconstruct not clause in the rule prevents it from matching Primary subexpressions (i.e. literals, identifiers, and expressions that are already parenthesized). This solution is considerably simpler than the standard TXL solution in Figure 6.2 because the transformation is decoupled from the precedence view and more clearly matches the intent of the transformation goal.

The Boolean grammar solution in Figure 6.3 uses a significantly simpler set of transformation rules at the cost of a slightly more complex grammar. Furthermore, the exemption for Primary subexpressions is moved from the rule to the grammar. Recall that the parenthesize transformation should apply to all expressions except Primarys (literals, identifiers, existing parenthesized expressions). This requirement is naturally captured using the Boolean grammar phrase Expr & ¬Primary.

Figure 6.4 shows a solution where this restriction is encoded in the the grammar view rather than the transformation rule. The rule no longer needs the deconstruct not clause because it directly targets a NotPrimary type, preventing it from being a literal or identifier because of the production NotPrimary → Expr & ¬Primary. The NotPrimary_or_Expr → NotPrimary | Expr production is an example of a categorization production, preferring to parse the input as a NotPrimary but falling through to an Expr if it cannot. Categorization productions are defined and discussed further in Section 6.3.
Figure 6.2: A TXL program to parenthesize subexpressions. Each level in the precedence grammar requires its own rule.
Figure 6.3: A TXL program, using Boolean grammar overrides, to parenthesize subexpressions. Each precedence level can be viewed as an expression, requiring a single rule that targets `Expr`.
redefine Comp
    ... & NotPrimary_or_Expr
end redefine

define NotPrimary_or_Expr
    NotPrimary | Expr
end define

redefine Sum
    ... & NotPrimary_or_Expr
end redefine

define NotPrimary
    Expr &! Primary
end define

redefine Term
    ... & NotPrimary_or_Expr
end redefine

rule main
    replace [NotPrimary]
        E [Expr]
    by
        ( E )
end rule

Figure 6.4: A TXL program, using Boolean grammar overrides, to parenthesize subexpressions. The NotPrimary nonterminal encodes the precise transformation target, further simplifying the transformation rule.

The three solutions to the parenthesize transformation problem presented in this section (Figures 6.2, 6.3, 6.4) suggest a tradeoff between grammar and rule complexity. The core idea behind agile parsing [6] is that rules can be greatly simplified by adding additional complexity to the grammar. In practice, it is up to the developer to decide at which level this tradeoff makes the most sense.

6.1.2 Type Consistency Analysis Example

The purpose of this transformation example is to perform a strong type analysis on expression operands, to catch erroneous expressions such as ‘‘1’’ + 3 where the two operands have different types. The solution involves annotating each subexpression with its type by appending :: plus its type name, for instance ‘‘1’’::string +
3::int. Consistent types are propagated from inner subexpressions to outer subexpressions by annotating the outer subexpression via transformation, for example 
\((1::\text{int} + 3::\text{int})\) becomes \((1::\text{int} + 3::\text{int})::\text{int}\). The full transformation to perform this analysis involves four steps:

1. Fully parenthesize all subexpressions, for example using the transformation discussed in Section 6.1.1.

2. Annotate literals and variables with their type information, for example:
   - ‘‘Hello’’ becomes ‘‘Hello’’::string
   - 3 becomes 3::int
   - isActive becomes isActive::boolean

3. Perform the type propagation described above for consistent types:
   (a) Comparison subexpressions are a special case. Consistent types surrounding a comparison operator propagate a boolean type, for example \((a::\text{int} > b::\text{int})\) becomes \((a::\text{int} > b::\text{int})::\text{boolean}\).
   (b) All other cases propagate the same type as the operands, for example 
\((a::\text{int} + b::\text{int})\) becomes \((a::\text{int} + b::\text{int})::\text{int}\).

4. Flag any unannotated subexpressions with an error, for example \((a::\text{string} + b::\text{int})\) becomes \((a::\text{string} + b::\text{int})::\text{ERROR}\).

This example uses the precedence grammar from Figure 6.1(b) as a starting point. Figure 6.5 shows the rules to implement the solution in standard TXL without any Boolean grammar features. The precedence grammar again intrudes on the implementation, requiring duplicate code and tightly coupling the transformation rules to
the grammar precedence levels. In particular, the \texttt{propagateTypes} rule calls five subrules, one for each precedence level, whose implementations are all similar.

We can remove the redundancies in the TXL rules of Figure 6.5 by again using Boolean grammar overrides to create a view that better suits our transformation task, resulting in the solution shown in Figure 6.6. The same overrides from Figure 6.3 are reused, to view each precedence level as an \texttt{Expr}. In addition, each operator non-terminal has been overridden to view them all as indistinguishable binary operators. This allows for the generic \texttt{Expr Op Expr} pattern used in the solution.

Figure 6.6 omits the implementations of grammar types \texttt{Primary} and \texttt{TypeAnnotation}, and rules \texttt{main}, \texttt{parenthesize}, \texttt{annotateLeaves}, and \texttt{flagErrors} because it uses the same implementations from the standard TXL solution in Figure 6.5; only the implementation of the \texttt{propagateTypes} rule and its subrules changes. The new implementation of \texttt{propagateTypes} has two subrules, one for the special case (comparisons) and one for the general case. This general case rule does not depend on the levels of the precedence grammar, unlike the standard TXL solution.

This example demonstrates the advantages of switching between parse views in a single transformation. In the enhanced solution in Figure 6.6, the old implementations of several rules (using the original precedence grammar view) are reused along with the new implementation of the \texttt{propagateTypes} rule (using the custom view). Consider the views used for each subrule of \texttt{main}:

\begin{verbatim}
P [parenthesize]       % Expr view
[annotateLeaves]      % precedence view
[propagateTypes]      % Expr and Op views
[flagErrors]          % precedence view
\end{verbatim}
redefine Primary

    ... [TypeAnnotation?]

end redefine

define TypeAnnotation
    :: [Type]
end define

function main
    replace [program]
        P [program]
    by
        P [parenthesize] % step 1 (implementation omitted)
        [annotateLeaves] % step 2 (implementation omitted)
        [propagateTypes] % step 3
        [flagErrors] % step 4
end function

% Step 3, propagate types. Must be done as a
% separate subrule for each precedence level
rule propagateTypes
    replace [program]
        P [program]
    by
        P [propagateUnary]
        [propagateMults]
        [propagateAdds]
        [propagateComps]
        [propagateBools]
end rule

Figure 6.5: A standard TXL program to type check subexpressions (continued on next page). The propagateTypes rule calls 5 subrules, one for each level in the precedence grammar.
rule propagateUnary
  replace [Primary]
  by
    (Op F::Ty)::Ty
end rule

rule propagateMults
  replace [Primary]
  by
    (T::Ty Op F::Ty)::Ty
end rule

rule propagateAdds
  replace [Primary]
  by
    (S::Ty Op T::Ty)::Ty
end rule

rule propagateBools
  replace [Primary]
    ( C [Comp] :: Ty [Type] Op [CompOp] S [Sum] :: Ty )
  by
    (C::Ty Op S::Ty)::boolean  % Step 3a, special case
end rule

rule flagErrors
  replace [Primary]
    ( E [Expr] )
  by
    ( E ) :: ERROR
end rule

Figure 6.5: A standard TXL program to type check subexpressions (continued from previous page). The propagate* rules are all similar, suggesting redundancy.
define Op
    [BoolOp] | [CompOp]
    | [AddOp] | [MultOp]
end define

redefine Comp
    ... & Expr
end redefine

redefine Sum
    ... & Expr
end redefine

redefine Term
    ... & Expr
end redefine

redefine Factor
    ... & Expr
end redefine

rule propagateTypes
    replace [program]
        P [program]
    by
        P [propagateComps] % step 4a, special case
        [propagateRest] % step 4b, general case
end rule

rule propagateComps
    replace [Expr]
    by
        (E1::Ty 0p E2::Ty)::boolean
end rule

rule propagateRest
    replace [Expr]
    by
        (E1::Ty 0p E2::Ty)::Ty
end rule

Figure 6.6: A TXL program, using Boolean grammar overrides, to type check subexpressions. Each precedence level can be viewed with a generic Expr and Op, requiring a single rule in the general case.
Without the enhanced TXL parser, the parse tree would need to be unparsed and reparsed with a different grammar between each transformation step. Boolean grammars capture multiple parse views simultaneously, allowing one to easily switch between them.

### 6.2 Grammar Specialization

Consider the following code snippet:

```plaintext
var x : int // variable declaration
x = 1     // variable modification
print x   // variable access
```

Although the variable `x` is used in three different ways here, for the sake of syntactic correctness a grammar can use the same nonterminal in all three cases. The three production definitions below for variable declaration, modification, and access (as part of a precedence grammar) all use the same nonterminal `[name]` where a valid variable name is expected.

```plaintext
define var_decl
  var [name] : [type]
end define

define var_mod
  [name] = [expr]
end define
```

For some tasks it is useful to distinguish between the declaration of a variable and its other references (i.e. modification and access). A specialization grammar can be constructed by introducing nonterminals `[decl_name]` and `[ref_name]` that serve as intermediate nonterminals in the parse tree [6].
define var_decl
  [decl_name] : [type]
end define

define var_mod
  [ref_name] = [expr]
end define

define factor
  'not [factor]
  | [ref_name] [opt arguments]
end define

define decl_name
  [name]
end define

define ref_name
  [name]
end define

define mod_ref
  [ref_name]
end define

define acc_ref
  [ref_name]
end define

This grammar can be further specialized in the same way to distinguish between variable modification and variable access:

define var_mod
  [mod_ref] = [expr]
end define

define factor
  'not [factor]
  | [acc_ref] [opt arguments]
end define

To summarize, we have discussed three different grammar views of variables in a program:

1. Declaration, modification, and access are all represented by the nonterminal [name].

2. Declaration is represented by [decl_name], modification and access are represented by [ref_name].

3. Declaration, modification, and access are each represented separately by non-terminals [decl_name], [mod_ref], and [acc_ref] respectively.
This section showed the process of grammar specialization, one of the agile parsing techniques [6]. However, unlike specialization with context-free grammars, the three different views mentioned above can be represented in a single Boolean grammar:

```plaintext
define var_decl [name] : [type] & [decl_name] : [type] end define
define var_mod [name] = [expr] & [ref_name] = [expr] & [mod_ref] = [expr] end define
```

6.3 Grammar Categorization

Agile parsing, described by Dean et al. [6], includes the technique of grammar categorization to make transformation rules easier to write. This takes the form of a production with two alternatives, $A \rightarrow \kappa | \gamma$, where $\gamma$ is the general form of a construct and $\kappa$ is a categorized subset of those forms.

**Definition 1.** A grammar production is a categorization production if it has the form $A \rightarrow \kappa | \gamma$, and $L(\kappa) \subset L(\gamma)$.

In TXL it is important that the categorized alternative, $\kappa$, appears first because the parser will try that alternative first and only proceed to the default general alternative, $\gamma$, if the first one fails to parse. One downside of this approach is that text parsed as a $\kappa$ cannot also be viewed as a $\gamma$. Boolean grammars can, however, allow both views.
Refactoring 1. A context-free grammar $G = (\Sigma, N, P, S)$ with categorization production $A \rightarrow \kappa | \gamma$ can be converted into a Boolean grammar $G' = (\Sigma, N', P', S)$ by replacing the categorization production with two new productions:

$$A \rightarrow X \& \gamma \quad X \rightarrow \kappa | \gamma$$

where $X \notin N$ is a new nonterminal not already in use.

These new productions simply parse the input as both the generic form and the original form (i.e. the categorized form is prioritized, but falls through to the generic form if necessary). This way if the categorized form is present the generic parse is still available. The next section investigates the Boolean grammar version of a categorization grammar example discussed in [6].

### 6.3.1 Method Call Categorization

In this example from Dean et al., JDBC (Java database connection) method calls are prioritized in the parse over other types of method calls because TXL will try to parse a `[method_call]` as a `[jdbc_call]` first before trying its alternatives [6]. This makes it easier to write rules that target JDBC calls only, but we lose the ability to easily target all method calls – JDBC and non-JDBC method calls alike.

```xml
define method_call
[jdbc_call] | [call_name] [arguments]
end define

define jdbc_call
[jdbc_name] [arguments]
end define

define jdbc_name
'createStatement' | 'prepareStatement' | 'executeUpdate' | 'executeQuery' | 'getResultSet'
end define
```
We can apply Refactoring 1 to nonterminal \texttt{method\_call} to yield a Boolean grammar with new nonterminal definitions:

\begin{verbatim}
redefine method_call
    [jdbc_or_default]
&     [call_name] [arguments]
end redefine

define jdbc_or_default
    [jdbc_call]
|     [call_name] [arguments]
end define
\end{verbatim}

Refactoring 1 can be applied in this instance because the language that \texttt{jdbc\_call} generates is strictly less than the language \texttt{call\_name \ arguments} generates (i.e. a JDBC call is a subset of a general method call).

\section{Summary}

In this chapter we explore several practical examples of Boolean grammar parsing and transformation. These examples demonstrate the applicability of Boolean grammars, in particular parse views via conjunction, to Dean et al.'s technique of agile parsing \cite{6}. We show the main difference between agile parsing using context-free grammars and Boolean grammars is that only one parse view is available at a time for context-free grammars, whereas multiple views are available in a Boolean grammar parse. The next and final chapter concludes this thesis and mentions possible future work in this area.
Chapter 7

Summary and Future Work

In this thesis we aim to advance the use of Boolean grammars in practical software reengineering tools. We accomplish this by making four major contributions to the field:

- A TXL parser implementation that supports Boolean grammars.
- A TXL transformer implementation that supports Boolean grammars.
- An analysis of different transformation semantics for parse DAGs.
- An examination of several reengineering tasks which benefit from using Boolean grammar features.

We recognize that Boolean grammars contain a natural way to represent parse views – the essence of grammar engineering techniques such as agile parsing [6] and island grammars [26]. However, these techniques lack the ability to capture multiple parse views simultaneously, a property Boolean grammars naturally possess. We explore
the implications of multiple parse views and the ability to switch between them in source transformation tasks, using TXL as the implementation platform.

In Section 7.1 we talk about ideas to improve this work in the future and discuss lessons we have learned in the course of our research. We give our final thoughts in Section 7.2.

7.1 Future Work

In this thesis we investigate the implications of transforming a parse tree produced from a Boolean grammar [16]. This investigation raised new challenges that, while trying to overcome them, gave us insight into the fundamentally difficult problems of this domain.

One such problem is dealing with malformed intermediate transformation results, mentioned on page 52. We introduce a conceptual solution to this problem in Section 7.1.1 using the idea of reparse transactions. In Section 7.1.2 we discuss our experiences and lessons we have learned by trying to implement Boolean grammar features in TXL [4]. We give our final thoughts and conclude the thesis in Section 7.2.

7.1.1 Reparse Transactions

Recall the problem of transformations that produce well-formed final results but malformed intermediate results, such as the example shown in Table 5.1 on page 52. We propose a solution to this called reparse transactions, inspired by database transactions and their properties of atomicity and consistency.
Atomicity refers to an all-or-nothing approach. In database transactions, all operations in the transaction must either succeed or all of them must fail; the database is never left in a state where some operations succeed but others fail. Consistency refers to the idea that all data constraints hold at the end of the transaction. If the series of operations in a transaction lead to inconsistent data then all those operations must be rolled back so the data is once again consistent.

Consistency, in the context of TXL, refers to a transformed parse tree conforming to the grammar used to parse the original input text. TXL places a high priority on keeping the parse tree consistent throughout its transformation, and ensures complete consistency by a compile-time static analysis of the rule patterns and replacements. The addition of conjunction and negation operators increases the possible constraints that can be present in a parse tree, requiring such constraints to be considered when ensuring consistency during a transformation.

TXL rules and functions are the mechanism by which a parse tree is transformed, similar to how INSERT, UPDATE, and DELETE operations are the mechanism by which data is changed in a database. And just as these database operations can invalidate data constraints, so too can TXL rules and functions invalidate constraints defined by the grammar.

We propose to complete this analogy for transactions. Database transactions are a series of INSERT, UPDATE, or DELETE operations consecutively applied to a database. Likewise, our proposed reparse transactions are a series of rules/functions applied to the same parse tree. Conveniently, TXL already has the notion of a series of rules applied to the same parse tree – rule application – using the syntax $T^\text{R1} [R2] [R3]$. 
Reparse transactions are meant to allow the programmer to include a consecutive subset of these rules into an atomic unit of work that guarantees consistency. We propose a syntax that demarcates the beginning and end of a reparse transaction with the [ and ] symbols respectively. For example, applying the transaction \( R_1; R_2 \) followed by rule \( R_3 \) to parse tree \( T \) would be: \( T \ [ \ [R_1] [R_2] \] [R_3] \). In this way, a sequence of rule/function calls can be grouped into a single unit of work.

In the context of TXL, a consistent parse tree is one that conforms to its grammar (i.e. the parse tree is derivable by applying a sequence of rewrite steps to the start symbol according to the grammar productions). Boolean grammars can enforce additional constraints, which a context-free grammar cannot, by way of their conjunction and negation operators, producing new challenges when trying to keep a parse tree consistent with its grammar. We propose that when program execution reaches the end of a transaction it triggers an automatic reparse of the current scope’s leaves as the scope type, because the leaves of one conjunct may no longer be consistent with its sibling conjuncts.

Atomicity is determined by the action taken after the consistency/repars check mentioned above. If the reparse is successful, the transaction succeeds and the changes made by the rules/functions in the transaction are committed to the parse tree. If the reparse is unsuccessful, the transaction fails and all changes made by the rules/functions in the transaction are rolled back. This rollback behaviour is exactly the same as a typical TXL rule or function; a failed rule/function makes no changes to its scope and execution continues with the next rule/function.

Our proposed syntax for transactions offer a way to express the implicit reparse
alternatives discussed throughout this thesis, assuming reparse verification only happens at the end of a transaction. One possibility is to only check consistency after all transformation rules have completed, which is equivalent to a transaction around the main function replacement:

```
function main
    replace [program]
        P [program]
    by
        P [ [R1][R2][R3] ]
end function
```

The ongoing reparse strategy (Section 5.2 on page 54) we chose to employ in our implementation is equivalent to a transaction around each subrule in a rule application: $T \ [[[R1]][[R2]][[R3]]]$. This reparse strategy is the safest possible one because it catches inconsistent forms as early as possible at runtime. However, it causes some transformations to be more difficult to write. Reparse transactions offer more flexibility to the programmer by letting them control when to verify consistency.

It is possible for an explicit reparse feature like transactions to coexist with an implicit reparse strategy like Ongoing Reparse. In this case, rules and functions not explicitly wrapped in a transaction cause a reparse when they complete, but rules and functions within a transaction delay reparsing until the end of the transaction. This allows transactions to be used as a way to temporarily disable automatic reparsing when appropriate, while keeping automatic reparsing as a sensible default.
7.1.2 Lessons Learned

In this section we attempt to convey the implications of our early design decisions with the benefit of hindsight. This information is valuable for anyone implementing Boolean grammars into another transformation tool or parser generator. In the following three subsections we briefly discuss lessons we have learned that were not obvious to us at the beginning of this project.

Importance of Lexer

In many parsers there is an initial phase prior to parsing that converts the input text into a stream of tokens. For example, the text `int x = 5;` is converted into five tokens: `[int] x [ = ] 5 [ ; ]`. This process is performed by a lexer (also known as a scanner), and its resulting token stream is fed to the parser. Although TXL can be configured to read each character in the input as a separate token (i.e. scannerless), its default behaviour is to use a lexer prior to parsing.

Our implementation of parse views in TXL allows one to represent a separate parse of the input, but not a separate scan of the input. This is unfortunate, as one cannot assume all views of the data will require the same lexer. An improved mechanism would not only capture multiple parses of the input, but also capture multiple scans, and any automatic reparsing would also include automatic rescanning.

This problem can be avoided completely by using a scannerless parser because the lexer is nonexistent. We suspect implementing a Boolean grammar parser in a scannerless context will be much easier for this reason. However, if a lexer is present it cannot be ignored as it is a fundamental part of how structured text gets interpreted by a parser.
Initial Positive Conjunct

On page 38 we mention how the choice of the and-not ($\&!$) operator guarantees the first conjunct in a production is always positive. This design decision was initially made to eliminate the possibility of misplacing a not operator, but turned out to have several unforeseen benefits in other areas. The implementation of the Boolean parser, in particular the backtracking implementation, was greatly simplified by knowing at least one parse tree will always exist for every conjunctive node.

Consider the opposite case, where the right-hand side of a conjunctive production contains only negative conjuncts. Negative conjuncts have no representation in the parse tree, making standard tree operations like traversal through that node impossible. The choice of $\&$ and $\&!$ operators in TXL was truly serendipitous.

Managing Grammar Complexity

Boolean grammars that generate non-context-free languages are more difficult to reason about than context-free grammars. Figure 4.2 on page 43 shows a Boolean grammar for variable declarations, but the fact that it is based on the $\{a^nb^nc^n \mid n > 0\}$ language is not immediately apparent. Okhotin defines another useful grammar for programming language constraints, but unless one has seen it before it is difficult to understand [18]:

\[
\begin{align*}
S & \rightarrow C \& D \\
A & \rightarrow aAa \mid aAb \mid bAa \mid bAb \mid cEa \\
C & \rightarrow aCa \mid aCb \mid bCa \mid bCb \mid c \\
B & \rightarrow aBa \mid aBb \mid bBa \mid bBb \mid cEb \\
D & \rightarrow aA \& aD \mid bB \& bD \mid cE \\
E & \rightarrow aE \mid bE \mid \epsilon
\end{align*}
\]
This is the \( \{ wcw \mid w \in \{a, b\}^* \} \) language, where two identical words \( w \) are separated by some text in the middle \( c \). The words in this grammar are composed of a restricted alphabet containing only \( \{a, b\} \), and would need to be considerably more complex for a practical alphabet such as all the valid characters in identifiers \( \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\} \), a more realistic case for a programming language grammar. The complexity and size of such a subgrammar easily dominates other productions in the main grammar, and it would aid readability and comprehension to untangle them.

We propose using Fischer’s idea of \textit{macro grammars} to help alleviate this problem \cite{7}. Macro grammars are a generalization of context-free grammars where nonterminals can have parameters. Each production in the grammar represents a macro: the left-hand side is the macro name (nonterminal name plus parameters) and the right-hand side is the macro body (some combination of terminals, macro calls, and parameter names from the left-hand side). For example, this is a macro grammar for the language \( \{a^n b^n c^n \mid n \geq 1\} \) \cite{7}:

\begin{align*}
S & \rightarrow F(a, b, c) \quad (7.1) \\
F(x, y, z) & \rightarrow F(xa, yb, zc) \quad (7.2) \\
F(x, y, z) & \rightarrow xyz \quad (7.3)
\end{align*}

The nonterminal \( F \) has three parameters and two possible expansions. The derivation for string \( aabbc \) is: \( S \xrightarrow{7.1} F(a, b, c) \xrightarrow{7.2} F(aa, bb, cc) \xrightarrow{7.3} aabbc \). Macro grammars without any parameters reduce precisely to context-free grammars.

We propose reusing this idea for Boolean grammars, specifically to define useful
non-context-free subgrammars as reusable macros separate from the main grammar, but which can be expanded into the main grammar. Given some arbitrary nonterminal $X$, it is possible to determine all terminal symbols in the alphabet of $L(X)$, and this information can be used to generate the appropriate Boolean grammar productions.

We do not currently have a suggestion for the syntax sugar in the main grammar to mark an expansion site, but its purpose would be to make the structure and constraints of the main grammar as clear as possible. This solution allows useful subgrammar patterns to be written once as a macro and plugged into a main grammar anywhere they are needed without polluting the grammar with productions that obscure its intent.

7.2 Conclusion

Context-free grammars have been the universal standard to describe programming language syntax for many years. Boolean grammars offer the opportunity for greater complexities to be imagined and expressed. Ultimately we will not know the best uses of Boolean grammars until users start experimenting on their own, and tools like these enable the first steps in that direction.
Bibliography


