Parse Views with Boolean Grammars

Andrew Stevenson, James R. Cordy

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

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 and the ability 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.

Keywords: Boolean grammar, agile parsing, source transformation

> *"Syntactic sugar" is a myth. Syntax is not the least important thing, it is the most important thing.*
>
> There are only a few things that Paul and I agree on completely, but I think that's one of them. It took over a decade for us to discover one another's work, and then it seemed so close we thought we had found kindred souls. We then spent the next decade misunderstanding one another so completely that it took two days of dueling presentations at GTTSE for each to finally figure out what the other was talking about. This is a good thing – if we were all using exactly the same ideas for the same problems, life would be boring. Paul, congratulations – I look forward to us continuing to misunderstand one another for many years to come.
>
> *Jim Cordy*

1. Introduction

Modern programming languages are typically context-dependent, and traditional context-free grammars are ill suited to deal with them. Compilers make up for this with additional checks or hacks to the LL or LR parsing algorithm so the full language can be accepted, but this is less than ideal. Progress has been made on this problem by backtracking LR [14] and generalized LR (GLR) [16] parsing algorithms to provide a restricted or localized form of context-dependence. We strive to continue this progress using *Boolean grammars* [11], a recent result from formal language theory.

In addition to their expressiveness, we propose Boolean grammars for their ability to simutaneously capture multiple different parse trees of the same input and treat these parses as "views" on the input text. Dean et al. use the phrase "agile parsing" [5] to describe how parse views can simplify tasks that transform or walk an abstract syntax tree, and Van den Brand et al. make a similar point on the benefits of grammar composition [15].

Section 2 presents background knowledge about the formal extension of Boolean grammars from context-free grammars and explains its significance to parsing text. Section 3 motivates our work from two perspectives: enhanced language expressiveness, and the benefits of parse views. Section 4 describes our efforts and plans to augment the TXL language and transformation engine with support for Boolean grammars.

Email addresses: andrews@cs.queensu.ca (Andrew Stevenson), cordy@cs.queensu.ca (James R. Cordy)

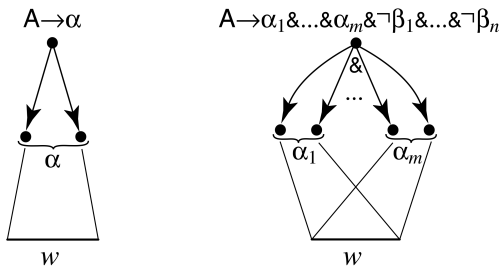

Figure 1: A context-free parse tree (left) and a Boolean grammar parse DAG (right). All positive conjuncts in a Boolean grammar production must share leaves (adapted from [11]).

2. Boolean Grammars

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

$$A \to \alpha$$
$$A \to \beta$$

where α and β are juxtapositions of terminals and non-terminals ($\alpha, \beta \in (\Sigma \cup N)^*$) and the right-hand sides of a particular non-terminal are separated by disjunction (i.e. A derives α or β, often written $A \to \alpha \mid \beta$). Boolean grammars generalize this formalism by introducing conjunction (&) and negation (¬)

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

while still maintaining disjunction from context-free grammars [11]. The production $S \to A \ \& \ B \ \& \ \neg C$ 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.

A Boolean grammar's parse tree leaves can be shared among its conjuncts, resulting in a directed acyclic graph instead of a tree. Figure 1 shows the difference between a parse tree for a context-free grammar and a parse DAG for a Boolean grammar [11].

Other useful generalizations of context-free grammars have been proposed [4, 1, 7, 8]. However, the danger of grammar generalization is in the formalism becoming Turing complete. One such example are two-level grammars, grammars that generate (possibly infinite) productions for a second grammar, a variant of which was used to define the Algol 68 programming language [13]. For these powerful grammars it is undecidable whether a string can be recognized/generated. In contrast, Boolean grammars have a known cubic-time algorithm to recognize and parse a string and they exist in the computational complexity space between context-free and context-sensitive grammars [11].

3. Motivation

Most programming language specifications contain constraints that cannot be expressed in a context-free grammar with Backus-Naur productions alone. For example, Floyd shows that the Algol 60 language cannot be fully defined using a context-free grammar because the program

$$\texttt{begin real } x^{(n)}; x^{(n)} := x^{(n)} \texttt{ end}$$

(where $x^{(n)}$ is n occurrences of x) corresponds to the non-context-free language $\{a^n b^n c^n | n \geq 0\}$ [6]. This language can, however, be expressed with a Boolean grammar [12]. The C programming language has the "typedef-name: identifier" problem, where the meaning of a statement such as `func((T) * x);` can either be a dereference of `x`, casted to type `T`, then passed to function `func` (if `T` is a type), or the `T` and `x` are multiplied and the result is passed to `func` (if `T` is not a type). Disambiguation depends on whether `T` was previously defined in a typedef or not, which is context-sensitive information. Section 3.1 illustrates some programming language constraints that can be expressed in a Boolean grammar but not in a context-free grammar.

Van den Brand, Sellink, and Verhoef identify several ways in which modern parsing tools are insufficient for reengineering tasks [15]. They emphasize that grammars change frequently in practice and a variety of language dialects are troublesome for grammar authors. In particular, they identify two desirable characteristics of grammars written for reengineering: modularity and composition. Dean et al. discuss the value of rapidly changing and flexible grammars in their paper on agile parsing [5]. Both papers recognize that language views can be used to help with these problems, where a language "view" is analogous to a database view: a perspective that restructures and filters the underlying data in a way that makes it easy for a task to consume. Section 3.2 elaborates on language views and describes how our plan to represent them using Boolean grammars contrasts with current approaches.

3.1. Language Expressiveness

Boolean grammars 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. Okhotin has demonstrated several such constraints (illustrative, not exhaustive) for an imperative programming language [12]:

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
   (c) All control-flow paths in a function must end with a return statement
2. Variable scoping
   (a) Defining the scope of a variable from its declaration context
   (b) Ensuring two variables of the same name do not exist in the same scope
   (c) A variable assignment statement must be in that variable's scope

Current grammar authoring tools usually provide some explicit mechanisms (e.g. precedence, associativity) to partially control the parse algorithm semantics through an extended syntax to the BNF-like notation used for production rules. These additional constraints help to disambiguate an otherwise ambiguous context-free language, and can sometimes define a context-sensitive language. Boolean grammars provide a similar power to these extensions, but this power arises out of the mathematical expressiveness of Boolean grammars themselves rather than an engineered solution to circumvent the limits of context-free grammars for common ambiguities.

A practical example of this enhanced expressiveness is in 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" [3]. An example of coupled artifacts are data types, their variable instances, and the programs that produce or consume them. For example, when a data type is transformed, instances of that data type also need to be transformed to keep them consistent. Traditionally this coupling is managed manually, or by adding a layer of control to the transformation that maintains the coupling. Unlike context-free grammars, Boolean grammars can have this coupling built in to the grammar definition. TXL currently requires homomorphic transformation rules – a rule targeting a non-terminal of type T must have a replacement pattern that can be parsed as a T. We are unsure if this homomorphic transformation property can be retained with Boolean grammars, but if so then the coupling expressed in the grammar will guarantee a consistent transformation result. Alternatively, the result can simply be reparsed with the Boolean grammar to verify it is consistent.

3.2. Parse Views

When parsing a software program it is often useful to have slightly different "views" of the input text depending on the task at hand, a technique known as *agile parsing* [5]. Consider some grammar productions for an assignment statement in an imperative programming language:

$$AssignStmt \rightarrow Id := Expr$$
$$Expr \rightarrow Id$$

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 those cases. We might rewrite the grammar as

$$
\begin{aligned}
AssignStmt &\rightarrow VarDef := Expr \\
VarDef &\rightarrow Id \\
Expr &\rightarrow VarRef \\
VarRef &\rightarrow Id
\end{aligned}
$$

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

Unfortunately, the state of the art in agile parsing is simply "grammar hacking" in the sense of Klint et al. [9], lacking for any firm engineering basis for views or the relationships between them. We propose to remedy this situation by expressing many views with a single formal construct, a Boolean grammar, which has the ability to specify multiple overlapping parse trees. Each of these parse trees represents a particular view on the input text, with the advantage of being able to switch between views easily depending on the current transformation task.

This advantage is particularly evident when sequential transformation stages requiring different views are interleaved. In current transformation systems, this situation requires the output of each stage to be unparsed then completely reparsed with a different view for the next stage. Using a Boolean grammar, all parsing is done up front and transformation stages can switch between views seamlessly.

Boolean grammars can differentiate views through its conjunction operator (&). Conceivably the entire input can be viewed in different ways using conjunction at the top-level production: $S \rightarrow View1 \ \& \ View2 \ \& \ View3$. But more likely one wants different views of a subset of the input text. Boolean grammars allow conjunction on the right hand side of any production rule, making these mid-level views possible without duplicating the rest of the grammar.

The benefit of this from a source transformation perspective is the easy transition between language views. In current systems the transformation engine only has access to one parse of the input at a time, whereas we propose giving access to all interesting views simultaneously. We believe this will encourage multi-pass transformation rules that are individually more well-defined and easier to understand because each transformation rule can operate on a parse tree tailored for it.

4. Towards an Implementation

Despite the differences between a parse tree and a parse DAG, our experience suggests modifying existing meta-programming languages to support Boolean grammars is not that difficult. We found that, structurally and algorithmically, conjuncts can be represented similarly to alternatives. The parse DAG adds some complexity, but not as much as a general DAG because only leaf nodes can have multiple parents. A generalized LR parsing algorithm, such as that used by ASF+SDF, has been developed and implemented for the closely related *conjunctive grammars*, essentially Boolean grammars without negation [10].

We are in the process of enhancing the TXL [2] language and source transformation engine to support Boolean grammars. The first step, parsing documents using a Boolean grammar, is nearing completion. To achieve this the TXL language has been augmented with an *and* operator (&) and an *and-not* operator (&!). This second operator was chosen because in the Boolean grammar formalism negation applies to the conjunct as a whole, not to individual elements within the conjunct. Many modern meta-programming languages already have a mechanism to restrict a particular production from being applied, similar to Boolean grammar's *not*. SDF's *reject* productions and Rascal's *reserve* declaration both serve this purpose, and are typically used for keyword reservation.

Figure 2 shows an example TXL Boolean grammar production corresponding to $Program \rightarrow A \ \& \ B \ \& \ \neg C$. In this example, the resulting parse DAG will contain the input text parsed as both an A and a B. The parser attempts to parse the input text as a C only to ensure it cannot be done. Negative conjuncts are not represented in the resulting parse DAG.

The second step of this research is to transform the parsed document. TXL currently transforms documents by matching and replacing subtrees in the whole parse tree. Transformation rules preserve well-formedness of the result by preserving non-terminal type. Type-preserving subtree replacement in a parse tree is straightforward because each

```
define program
  [A] & [B] &! [C]
end define
```

Figure 2: A Boolean grammar production in TXL

branch of a tree is isolated from other branches at the same level. A parse DAG generated from a Boolean grammar can contain overlapping subtrees, which cannot be replaced without considering how the replacement affects related branches.

5. Conclusion

We believe Boolean grammars have much to offer the source transformation and reengineering communities. Their inherent expressiveness allows for more context-dependence and can encode the coupling between source elements in a coupled transformation. Furthermore, parallel views of the input text allows transformation rules to switch among these views without unparsing and reparsing the input every time. Going forward we hope to uncover new use cases, idioms and paradigms using Boolean grammars in this environment.

References

[1] Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, October 1968.

[2] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.

[3] Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. In *Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006), ENTCS*, pages 17–34. Elsevier, 2006.

[4] J. Dassow and G. Paun. *Regulated Rewriting in Formal Language Theory*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[5] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in TXL. *Automated Software Engg.*, 10(4):311–336, October 2003.

[6] Robert W. Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Commun. ACM*, 5(9):483–484, September 1962.

[7] Gerald Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, number 35 in Studies in Linguistics and Philosophy, pages 69–94. Springer Netherlands, January 1988.

[8] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, February 1975.

[9] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005.

[10] Alexander Okhotin. Whale calf, a parser generator for conjunctive grammars. In *Proceedings of the 7th international conference on Implementation and application of automata*, CIAA'02, pages 213–220, Berlin, Heidelberg, 2002. Springer-Verlag.

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

[12] Alexander Okhotin. On the existence of a boolean grammar for a simple programming language. In *11th International Conference on Automata and Formal Languages*, Hungary, May 2005.

[13] M. Sintzoff. Existence of van wijngaarden syntax for every recursively enumerable set. In *Annales de la Socit Scientifique de Bruxelles*, pages 115–118, 1967.

[14] Adrian D. Thurston and James R. Cordy. A backtracking LR algorithm for parsing ambiguous context-dependent languages. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, CASCON '06, Riverton, NJ, USA, 2006. IBM Corp.

[15] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *6th International Workshop on Program Comprehension, 1998. IWPC '98*, pages 108–117, June 1998.

[16] M.G.J. van den Brand, A.S. Klusener, L. Moonen, and J.J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. *Electronic Notes in Theoretical Computer Science*, 82(3):575–591, December 2003.