sbp: A Scannerless Boolean Parser

Adam Megacz

Computer Science
UC Berkeley

Abstract

Scannerless generalized parsing techniques allow parsers to be derived directly from unified, declarative specifications. Unfortunately, in order to *uniquely* parse (disambiguate) existing programming languages, extensions beyond the usual context-free formalism must be added to handle a number of specific cases.

This paper describes sbp, a scannerless parser for *boolean* grammars, a superset of context-free grammars. In this expanded formalism, special-purpose disambiguation constructs become special cases of a broader formalism, yet all grammars admit $O(n^3)$ parsers using a derivitave of the Lang-Tomita algorithm. The implementation is publicly available as Java source code.

Key words: boolean grammar, scannerless, GLR

1 Introduction

Although scannerless parsing was first introduced in [8], it was not practical for general use until Visser implemented a variant of generalized LR parsing [9] at the character level and identified six constructs necessary for disambiguation (FOLLOW, REJECT, PREFER, AVOID, associativity, and precedence). These extensions can be grouped into three categories based on their impact on the expressivity of the resulting algorithm.

Associativity and precedence attributes can be expressed cleanly within the context-free formalism by replacing a self-reference in one of a nonterminal's productions with a reference to *a subset of* that nonterminal's productions. Viewed in this light, the attribution mechanism serves mainly as a convenience when writing a grammar and a signal that the restriction can be implemented more efficiently as a modification to the parse table.

The presence of PREFER and AVOID attributed productions does not alter the expressive power of the formalism (ie they cannot be used to recognize any

[1] Email: megacz@cs.berkeley.edu

non-context-free languages); however, these constructs do make a context-sensitive choice between two possible parsings of an expression.

The follow and reject features each increase the power of the formalism to encompass some non-context-free languages. The precise set of languages accepted is not clearly defined.

2 Conjunctive and Boolean Grammars

Conjunctive Grammars [2] augment the juxtaposition ($\cdot$) and language-union (`|`) operators of context-free grammars with an additional language-intersection (`&`) operator. Boolean grammars [6] further extend conjunctive grammars by permitting the language-complement operator ($\neg$) to be used, subject to some basic well-formedness constraints [2] .

It should be noted that Visser arrives at a similar result from the opposite direction: [4] includes reject productions, which effectively function as conjunction with a negation. The paper goes on to reconstruct simple negation as well as intersection in terms of this negated-conjunction primitive, noting that "this feature can give rise to as yet unforseen applications."

3 SBP: a Scannerless Boolean Parser

We have implemented the Lang-Tomita Generalized LR Parsing Algorithm [9], employing Johnstone & Scott's RNGLR algorithm [1] for handling ϵ-productions and circularities. All of the disambiguation constructs from [3] are supported for purposes of comparison.

The input alphabet for `sbp` is typically the set of individual Unicode characters, though any topological space [3] can be used. An interesting consequence is that `sbp` can parse sentences constructed from non-discrete alphabets [4] .

In its current form, the parser lets the user specify rewrites to be performed on the parse tree on a per-nonterminal basis. These rewrites correspond to the four promotion operators (drop, keep, promote, promote-over) from `rdp` [7] along with the addition of constant subtrees in response to matching a rule.

The parser's grammars are built programmatically and can be manipulated and modified through a clean and simple API. Textual grammar descriptions can be parsed using the included metagrammar, which supports subexpressions, regular expression operators (`*`, `+`, `?`) with or without separator nonterminals, n-ary associativity, ordered choice, general priorities (prefer/avoid over a partial ordering), promotion/rewrite operators, character ranges, and automatic whitespace insertion.

[2] for example, a nonterminal cannot be defined to produce exactly its own complement

[3] any space for which $\cup$, $\cap$, $\neg$, and $\subseteq$ are supplied

[4] although we have not yet found a practical use for this capability

4 Example: Indentation Block Structure

Aside from subsuming the usual disambiguation constructs, boolean grammatical operators lend themselves to a number of applications. Here, we show how to handle a form of indentation-based block structure by imposing a well-formedness constraint using conjunction, in a style inspired by[10].

We begin with the sbp grammar for a simple fragment of a C-like language. Note that the ++ operator denotes maximal repetition, ~ denotes negation, and & denotes intersection. The grammar uses conjunction with a negated term to exclude identifiers whose names happen to be keywords, similar to [3].

```
Statement ::= Call                      Call      ::= Expr "()"
            | "while" Expr block
                                        op        ::= ">" | "<"
Expr      ::= ident                     num       ::= [0-9]++
            | Call
            | Expr op Expr              ident     ::= [a-z]++ & ~keywords
            | num                       keywords  ::= "while" | "if"
```

We can now use boolean language operations to impose additional structure. We will do this by defining a nonterminal for syntactic blocks, and intersecting it with another production which requires that no line in a block be indented less than the first line. Lastly, we use the metagrammatical "ordered choice" operator (>) [5] to prefer "tall" block productions in a manner reminiscent of the "dangling else" convention.

```
indent  ::= " "*
outdent ::= " "  outdent " "
          | " "  [~]*    "\n"

block     ::= "\n" indent  BlockBody
           &~ "\n" outdent [~ ] [~]*

BlockBody ::= Statement
            > Statement BlockBody
```

The block rule matches code blocks which start a new line. The rule requires a newline, followed by some number of spaces, followed by a BlockBody. This production is intersected with another which acts as a well-formedness constraint: after the newline, a block cannot match an outdent.

Similar to the sort of rule used to match balanced parenthesis, the outdent rule will match any text which begins with indentation and also contains some other (disjoint) instance of indentation which is *shorter* than the first instance. In the context of the block production, this would describe any block containing a line with indentation less than that of the first line in the block.

[5] which is implemented by intersecting lower priority productions with the complement of higher priority productions

5 Related Work

The original scannerless generalized parser, `sglr`[11] was designed as an improved parser for the ASF+SDF[12] framework.

Dparser [16] is an implementation of the GLR algorithm in ANSI C, with support for most of Visser's disambiguation rules.

Several GLR parsers are available which require a tokenizer (some can be used as "character level" parsers, but lack the disambiguation capabilities necessary to parse real grammars at this level). These include Elkhound[13], and the GLR extensions to `bison`.

Although Parsing Expression Grammars (PEG)s[14] and their companion parsing algorithm[15] do not handle all context-free grammars, they do provide enough expressivity to handle the requirements of scannerless parsing and include a limited form of intersection and complement.

6 Future Directions

The current implementation is written in Java. It generates parse tables (which can be saved and restored), but currently only provides support for *interpreting* these tables. Emitting compilable source code equivalent to parsing from these tables will be an important step in improving the performance of `sbp`.

Like the `sglr` parser, `sbp` deliberately excludes support for semantic actions, preferring to keep grammar definitions implementation-language-neutral. One consequence is that parsing requires space which is linear in the input, since the entire parse tree (modulo portions removed using the drop operator) must be constructed before any part of it can be consumed. An important future direction is the exploration of constructing *lazy parse forests* which can be incrementally consumed and discarded by a process running concurrently with the parser.

7 Availability

The source code for `sbp` is available under the terms of the BSD license, at `http://www.cs.berkeley.edu/~megacz/sbp/`.

References

[1] Johnstone, Adrian and Scott, Elizabeth. *Generalised reduction modified LR parsing for domain specific language prototyping.* Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02), IEEE Computer Society, New Jersey, (January 2002).

[2] Okhotin, Alexander. *Conjunctive Grammars.* Journal of Automata, Languages and Combinatorics 6(4): 519-535 (2001)

[3] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. *Disambiguation filters for scannerless generalized LR parsers.* In N. Horspool, editor, Compiler Construction (CC'02), Lecture Notes in Computer Science. Springer-Verlag, 2002.

[4] Visser, E. (1997b). *Scannerless generalized-LR parsing.* Technical Report P9707, Programming Research Group, University of Amsterdam.

[5] Okhotin, Alexander. *LR Parsing for Boolean Grammars.* Developments in Language Theory 2005: 362-373.

[6] Okhotin, Alexander. *Boolean Grammars.* Developments in Language Theory 2003: 398-410.

[7] Johnstone, Adrian and Scott, Elizabeth. *Constructing reduced derivation trees.* University of London CSD-TR-97-27 (1997)

[8] Daniel J. Salomon and Gordon V. Cormack. *Scannerless NSLR(1) parsing of programming languages.* In Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation, pages 170-178. ACM Press, 1989.

[9] Tomita, M. (1987). *An efficient augmented-context-free parsing algorithm.* Computational Linguistics, 13(1-2), 31–46.

[10] Okhotin, Alexander. *On the existence of a Boolean grammar for a simple procedural language.* Proceedings of AFL 2005.

[11] Visser, Eelco. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[12] A. van Deursen, J. Heering and P. Klint (eds.), *Language Prototyping: An Algebraic Specification Approach*, AMAST Series in Computing, Volume 5, World Scientific, September 1996

[13] Scott McPeak and George C. Necula. *Elkhound: A Fast, Practical GLR Parser Generator.* Proceedings of Conference on Compiler Constructor (CC04), April 2004.

[14] Bryan Ford. *Packrat Parsing: Simple, Powerful, Lazy, Linear Time.* International Conference on Functional Programming, October 4-6, 2002, Pittsburgh

[15] Bryan Ford. *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.* Symposium on Principles of Programming Languages, January 14-16, 2004, Venice, Italy.

[16] http://dparser.sourceforge.net/