Scannerless Boolean Parsing

Adam Megacz[1]

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 existing programming languages at the character level, disambiguation extensions beyond the usual context-free formalism are required.

This paper explains how scannerless parsers for *boolean grammars* (context-free grammars extended with intersection and negation) can specify such languages unambiguously, and can also describe other interesting constructs such as indentation-based block structure.

The `sbp` package implements this parsing technique and is publicly available as Java source code.

Key words: boolean grammar, scannerless, GLR

1 Introduction

Although scannerless parsing[2] was first introduced in [1], it was not practical for general use until combined with the Lang-Tomita Generalized LR parsing algorithm [2,3] by Visser [7]. Unfortunately, the context-free grammars for most programming languages are ambiguous at the character level, which motivated the introduction of six empirically-chosen disambiguation constructs: FOLLOW, REJECT, PREFER, AVOID, ASSOCIATIVITY, and PRECEDENCE.

2 Conjunctive and Boolean Grammars

Conjunctive grammars [8] augment the juxtaposition (·) and language-union (|) operators of context-free grammars with an additional language-intersection (&) operator. Boolean grammars [10] further extend conjunctive grammars by

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

[2] also called "lexerless" or "character-level"

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

permitting the language-complement operator (~) to be used, subject to some basic well-formedness constraints[3] .

It should be noted that Visser arrives at a similar result from the opposite direction by using reject productions, which act as conjunction with a negation. The paper goes on to reconstruct simple negation as well as intersection in terms of this negated-conjunction primitive, even noting that "this feature can give rise to as yet unforeseen applications." [7]

3 Disambiguating with Boolean Constructs

We now examine each of the disambiguation constructs and explain how to recast it as a boolean expression.

The prefer and avoid attributes effectively make a context-sensitive choice between ambiguous parsings, thus turning an ambiguous context-free grammar into an *un*ambiguous context-*sensitive* grammar. We can replace prefer and avoid with an ordered-choice operator (`>`), which is a metagrammatical abbreviation for intersecting the lower-priority expression with the complement of the higher-priority expression [4]

The associativity and precedence features cannot be expanded into simple context-free expressions when they span multiple nonterminals, as in example 4.5.11 of [5]. In this example, addition and multiplication expressions are defined for the real numbers (`R`) and natural numbers (`N`). A subsumption production (`N` → `R`) is included, but operations on the natural numbers assume higher priority than corresponding operations on the reals. This sort of rich priority specification can be expressed in a manner similar to the ordered-choice operator: expressions are intersected with the complement of all higher-priority expressions, even those which involve productions from multiple nonterminals.

Uses of the reject attribute can be trivially translated into intersection with the complement of the rejected expression.

A follow restriction can be written as a boolean expression if one considers *character boundaries* (pairs of adjacent characters) as input tokens. From this perspective, a follow restriction amounts to intersecting an expression with the set of all strings ending with a valid follow-boundary.

4 SBP: a Scannerless Boolean Parser

The `sbp` package is an implementation of the Lang-Tomita Generalized LR Parsing Algorithm [2,3], employing Johnstone & Scott's RNGLR algorithm [13] for handling ϵ-productions and circularities.

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

[4] for example, `a > b` expands to `a | (b & ~a)`

The input alphabet for sbp is typically the set of individual Unicode characters, though any topological space[5] can be used. An interesting consequence is that sbp can parse sentences constructed from non-discrete alphabets.[6]

The parser's grammars are built programmatically and can be manipulated and through a simple API. A sample metagrammar is included; it supports alternation (|), intersection (&), complement (~), intersect-with-complement (&~), subexpressions (()), regular expressions (*, +, ?), repetition with a separator (*/, +/), maximal character repetition (++, **), ordered choice (>), promotion operators (as in [12]), character ranges ([a-z]), and whitespace insertion (/ws).

5 Examples

5.1 Dangling Else

A classic example of grammatical ambiguity is the so-called "*dangling else*" construct [18]. Straightforward application of negation and intersection can exclude expressions with a trailing else clause from the then-branch of an if statement, leaving the intended interpretation as the only valid parse:

```
Expr   = "if" "(" Expr ")" IfBody
       | ...

IfBody = Expr              "else" Expr
       | Expr   &  ~([~]* "else" Expr)
```

The complement of the empty character class ([~]) is an idiom used to match any character.

5.2 Indentation Block Structure

Besides disambiguation, boolean grammatical constructs have an number of other applications. The following example parses a language with indentation-based block structure by imposing a well-formedness constraint on blocks. The technique employed was inspired by [11].

We begin with the grammar for a simple fragment of a C-like language. The grammar uses conjunction with a negated term to exclude identifiers whose names happen to be keywords, just as in [6].

```
Statement = Expr "()"              ident       = [a-z]++ & ~keywords
          | "while" Expr block     keywords    = "while" | "if"

Expr      = ident
          | [0-9]++
```

We can now use boolean language operations to impose additional struc-

[5] one for which the ∪, ∩, ∼ operators and the ⊆ (or simply =) test are supplied

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

ture. 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 can be indented less than the first line.* Lastly, we use the ordered choice operator to prefer "tall" (left-associative) `BlockBody` productions.

```
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 a well-formedness production: the newline must not be followed by an `outdent`.

Similar to the sort of rule used to match balanced parentheses, the `outdent` rule matches 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.

6 Related Work

The original scannerless generalized parser, `sglr`[5] was designed as an improved parser for the ASF+SDF[4] framework. Dparser [17] 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.[7] These include Elkhound[14], and the GLR extensions to `bison`.

Parsing Expression Grammars (PEG)s[15] include a limited form of intersection and complement, and the corresponding algorithm [16] is effective at parsing character-level grammars. However, many interesting context-free grammars are not PEGs, and cannot be parsed this way.

7 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

[7] some can be used as "character level" parsers, but lack the disambiguation capabilities necessary to parse most programming languages at this level

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 possibility of constructing *lazy parse forests* which can be incrementally consumed and discarded by a process running concurrently with the parser.

8 Availability

The source code for sbp is available under the terms of the BSD license, at http://research.cs.berkeley.edu/project/sbp/.

References

[1] Daniel J. Salomon and Gordon V. Cormack. *Scannerless NSLR(1) parsing of programming languages.* SIGPLAN '89, pp 170-178. ACM Press, 1989.

[2] Lang, Bernard. *Deterministic Techniques for Efficient Non-deterministic Parsers.* Automata, Languages and Programming, Springer, 1974.

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

[4] 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

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

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

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

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

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

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

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

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

[13] 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).

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

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

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

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

[18] Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language, Second Edition* Prentice Hall, Inc., 1988. ISBN 0-13-110362-8