# Expressing Disambiguation Filters as Combinators

José Nuno Macedo
jose.n.macedo@inesctec.pt
HASLab/INESC TEC
University of Minho, Portugal

João Saraiva
saraiva@di.uminho.pt
HASLab/INESC TEC & Depart. of Informatics,
University of Minho, Portugal

## ABSTRACT

Contrarily to most conventional programming languages where certain symbols are used so as to create non-ambiguous grammars, most recent programming languages allow ambiguity. These ambiguities are solved using disambiguation rules, which dictate how the software that parses these languages should behave when faced with ambiguities. Such rules are highly efficient but come with some limitations - they cannot be further modified, their behaviour is hidden, and changing them implies re-building a parser.

We propose a different approach for disambiguation. A set of disambiguation filters (expressed as combinators) are provided, and disambiguation can be achieved by composing combinators. New combinators can be created and, by having the disambiguation step separated from the parsing step, disambiguation rules can be changed without modifying the parser.

## CCS CONCEPTS

• **Theory of computation → Parsing**; *Grammars and context-free languages*; *Program reasoning*;

## KEYWORDS

parsing, disambiguation filters, combinators

**ACM Reference Format:**
José Nuno Macedo and João Saraiva. 2020. Expressing Disambiguation Filters as Combinators. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3341105.3374123

## 1 INTRODUCTION

The evolution of programming languages in the 1960s was accompanied by the development of techniques for the syntactic analysis of programs. While techniques for processing text have evolved since then, the general approach has remained the same. To define and implement a new programming language, the general approach tends to be the use of context-free grammars to specify the programming language syntax. Then, a grammar-based tool, called parser generator, automatically generates programs known as *parsers*. Such parsers are able to syntactically recognize whether a text is a program in the specified programming language. Thus, for a parser to be generated, a context-free grammar is needed.

From such grammar, a parser generator, such as the widely used YACC parser generator system [4], produces a parser (implemented in a specific programming language) that given a text (i.e. a sequence of characters) accepts/rejects it. If the text is accepted a (abstract) syntax tree may be constructed. For unambiguous grammars, a single tree is built, meaning that there is only one possible way of accepting the text from such a grammar. However, programmers often write ambiguous grammars. Firstly, because they are easier to write/understand and evolve. Secondly, because modern languages provide a "cleaner" syntax, which make programs look nicer, but are easier to express by ambiguous grammars.

Regular parser generators do not support ambiguous grammars. Thus, the grammar writer has to provide the priority/associativity rules of the ambiguous operators, in two ways: by refactoring the grammar to eliminate ambiguity (which can be complex, and results in a more complex and hard to understand grammar), or by providing the so-called disambiguation rules, which are specified in the grammar. These disambiguation rules are pre-defined for most parser generators, and are directly imbued into the parser itself when it is generated, effectively modifying it. If the disambiguation rules are well-defined, there will be no ambiguity problems and the parser will be able to recognize text without any problem. However, There are several problems with this approach:

- The only rules available are the pre-defined rules and they are not extensible: the parser generator itself would need to be updated in order to support new rules.
- Because disambiguation rules are part of the grammar, they are context-free too. Thus, it is impossible to define context-dependent rules like for example to express that `'+'` operator has a different priority/associative when inside a while loop.
- It is not modular. In fact, when the developer changes a disambiguation rule, the grammar changes, therefore a new parser must be generated.
- Since the only rules available are the pre-defined rules, the developer is unable to observe the source code of these rules, instead opting to trust a black box that could potentially not behave as desired.

This paper presents an alternative to the classical approach, which does not suffer from these drawbacks: Disambiguation rules are modular combinators that are kept separate from the parser, being instead used as filters that are applied to the results of parsing an input. In this way, changes to the disambiguation rules do not affect the parser, allowing for an efficient development cycle around disambiguation rules. Because we express disambiguation rules as combinators, new rules can be easily defined by combining existing ones. Moreover, our approach allows the definition of context dependent disambiguation filters which behave differently according to the context they are applied to.

## 2 GRAMMARS AND PARSERS

In the early ages of programming languages, it was usual to include certain symbols in a language's grammar so that the generated parser for that language would be more efficient. One such example is found in the C programming language: the semicolon found at the end of each instruction is a statement terminator [3].

Modern programming languages, however, tend to avoid the use of too many syntactic symbols. This has the clear advantage that it not only allows developers to write fewer symbols and less code while programming, but also makes programs simpler and easier to understand. Although it helps program comprehension, it also comes at a price: it requires large and complex grammars and corresponding parsers to handle them! In the next section we discuss grammars and generalized parsers that can handle ambiguity.

While programming languages are usually specified via grammars, for example using the BNF notation [1], there are various ways to generate a parser given such specification. Each alternative method has its own advantages and disadvantages. One of the most well-known parser generators, YACC, uses an efficient LALR parsing technique [4]: this relies on a lightweight table-driven algorithm which was developed when runtime and memory size were one of the main concerns. Other popular parser generators include ANTLR [7], which uses the ALL(*) [8] algorithm, and Happy [6], which produces Haskell source code.

### 2.1 Generalized Parsing

Several parsing techniques do not deal with ambiguity properly. The input is expected to be unambiguous, and when it is not, a certain interpretation of such ambiguity is chosen so as to continue parsing. This results in runtime-wise efficient but not so expressive parsers, as they ignore any ambiguity problems that may arise.

Ambiguity can be dealt with using Generalized LR (GLR) parsing, which just try all different parsing paths when an ambiguity occurs. Thus, when faced with an ambiguous input, a GLR parser [12] produces possible outputs instead of selecting one of them. That is to say that it produces a set of abstract syntax trees, called parse forest, and not a single one. As a consequence, they are slower than their non-generalized counterparts, due to their additional flexibility in dealing with non-determinism. If no non-determinism is present, a GLR parser will be as efficient as a regular LR parser [5].

### 2.2 Scannerless Parsing

Grammars usually rely on regular expressions to specify their terminal symbols, which are then processed via very efficient finite state automata-based recognizers [10], the so-called scanners [4]. The use of scanners provides also some form of modularity in grammar/parser writing. However, this approach has a severe limitation: because some parts of the language are defined outside the BNF grammar formalism, and handled by a scanner, there is not an unified and coherent way of processing all symbols of the language.

Scannerless parsing [9] consists of skipping the lexer-phase entirely and treating each character from the input as a token, which is directly processed by the parser. As a consequence, every program's character is processed by the parser, and not by an external lexer. Usually, terminal grammar symbols are specified via regular expressions, still, but they are transformed into an equivalent regular CFG before a parser is produced. There are two key advantages in this approach: First, ambiguous grammars are compositional, and as a result two grammars can be merged and a (ambiguous) parser can be generated, which will not be the case when merging regular expressions. Second, because all grammar symbols are handled in the same way, *i.e. via a parser*, advanced parsing techniques can be applied to all symbols. This is the case, for example, of the disambiguation rules that we discuss in the next section.

## 3 DISAMBIGUATION FILTERS FOR SCANNERLESS GENERALIZED PARSING

Scannerless Generalized parsers handle ambiguous grammars and inputs. Moreover, they handle all grammar symbols in a uniform and canonical way, not relying on external recognizers to process part of the input. Since they deal with ambiguous grammars/inputs, it is expected that they generate a set of outputs as a result, which represent all possible interpretations. However, not all possible interpretations are desired: depending on the situation, a developer might want to only get one or a small subset of parse trees.

The task of processing the set of ambiguous parse trees generated by a parser and eliminating the undesired ones is called disambiguation. Typically, such filtering is done on the parser itself, by adding disambiguation rules or by modifying part of the parser, so that the undesired interpretations cannot be produced. When dealing with scannerless parsing new disambiguation rules are needed. In this section, we present a set of disambiguation filters following the the work of van den Brand *et al.* [14].

The **priority** filter specifies that certain productions have a higher priority than others, while the **associativity** filter specifies that an operator associates left or right. These filters can be specified as annotations in the productions they refer to.

The **reject** filter enables the creation of keywords in the grammar. In other words, it rejects some productions from deriving into certain sequences. This is extremely useful as in most programming languages, some keywords cannot be used as variable names, and this filter allows for a clean implementation of this incompatibility. For example, in the C language, it is not allowed for a variable to have the name of the reserved keyword *"while"*.

The **follow** filter (also known as longest match filter) solves a less obvious ambiguity that arises in scannerless parsing. When the grammar dictates that a sequence of symbols can be parsed using one single production or a sequence of productions, for example, a sequence of digits which could be read as a single number or several numbers with no separators, the follow filter specifies that the longest match is to be performed.

When there are several correct input interpretations, but some are preferred to others, a **preference** filter is used. It specifies which parse results should be removed when there are several correct outputs but the developer wants to select only some of them. This filter is used to disambiguate the dangling else problem, which can be exemplified by the input *if bool1 then if bool2 then out1 else out2* that can be interpreted in two ways,*if bool1 then (if bool2 then out1 else out2)* or *if bool1 then (if bool2 then out1) else out2.*

## 4 EMBEDDED DISAMBIGUATION FILTERS

This section presents a new approach for parser disambiguation, where instead of expressing the disambiguation rules in the parser itself, they are kept separate. The parser is generated once, and it produces a possibly ambiguous result. Afterwards, the disambiguation rules are applied to the parser's forest, removing some or all of the ambiguities, according to the rules specified by the developer. There are several advantages and disadvantages in using this process instead of the classical approach. Since the parser is a pure generalized one, it is less efficient, as the classical approach uses disambiguation rules at parse time thus reducing the number of parser results. However, the development cycle of the developer is more efficient, as there is no need to constantly produce a new parser after an update in a disambiguation rule. Only the disambiguation rules are to be changed, and this can be easily done if the implementation is user-friendly. Therefore, the disambiguation rules are implemented as filter combinators, where the developer starts with basic blocks that perform very simple filtering, combining them in easy-to-understand ways to produce complex filters that perform the desired disambiguation rules.

### 4.1 Abstract Syntax Trees

A parser typically constructs the parsing result as an abstract syntax tree. This is a tree that contains all the information from the input categorized in accordance to the grammar. For a generalized parser, the output is not a tree but a ***forest***, that is, a set of trees. Since our disambiguation process occurs after the parsing phase, the disambiguation rules are applied to the syntax forest, trying to find undesired patterns in them. This process will be described in more detail in the following sections.

### 4.2 Haskell XML Toolbox and HAGLR

Syntax trees are generalized trees which can represent a program. Generalized trees are often called Rose Trees in the functional programming setting and are well studied in several contexts. One of them is XML, for which there are several generic tools that can be used. In this work, the filter variant of the Haskell XML Toolbox [11], where Rose Trees are known as *NTree*, is used as a base for building combinators for filtering syntax trees.

The HaGLR tool [2] is a Haskell implementation of a GLR parser generator, which was implemented with pedagogic purposes. Since performance is not our main focus and HaGLR is a generalized parser generator, it is adequate for this work. It produces as result a pure parse tree forest, which is a list of parse trees.

### 4.3 Disambiguation Filters

To be able to build complex filters to disambiguate the result of a parser, some basic combinators to build upon are needed. Most of the combinators described in this section are defined in the Haskell XML Toolbox library. They enable the creation of filters, as well as manipulation and composition. Some new combinators were also created to better fit the needs of this work. They are available in the repository of this work, but as they are simple and intuitive, they are not described in this paper.

Having defined these Rose Tree filter combinators, we have all ingredients to implement the aforementioned disambiguation filters.

In the following sections, the types of filters described in section 3 are implemented using these combinators. However, it is important to note that they apply to the parse trees produced by the HaGLR parser, and if a different parser is used, it might be needed to change the filters accordingly.

To build a filter that defines disambiguation rules, it is first needed to take a look at a parse tree and devise an algorithm for checking if it is a valid parse tree. To do so, it is important to understand the structure of the parse trees produced by the parser. The combinators used in this work refer to the nodes by their production's name (e.g. "InfixExp1" or "Values1"), as this allows for the manipulation of any node, regardless of how it is built. A small change to the combinator that check the nodes could allow for the matching of the nodes by the symbols they represent (e.g. "+" or "*"), which is more practical for the implementation of simple rules but less expressive.

### 4.4 Associativity Filter

For a given example where *InfixExp2* is a node where associativity ambiguity can occur, the defined filter will look at the root node, check if it is an *InfixExp2* node, and, if it is, check if there is an *InfixExp2* node in the right (or left) child of said node. If there is not, then the tree is correct according to the filter and thus the filter will do nothing. If there is, the tree is deemed invalid and discarded.

```
associativity :: TFilter String
associativity = every (neg rightNodeCheck `when` matches
    ↪ "InfixExp2")
 where rightNodeCheck = matches "InfixExp2" . head .
     ↪ getChildren . (!!2) . getChildren
```

Therefore, the filter is finished and can be read in a reasonably easy way. **When** the root **matches** the string *"InfixExp2"*, the rightmost child must not **match** the string *"InfixExp2"*.

However, this filter does not work as expected on a real parse tree without the **every** combinator, which applies the filter to all nodes of the tree, and discards the tree if any of the nodes fail to satisfy it. This pattern is repeated for all filters.

### 4.5 Priority Filter

The priority filter is rather similar to the associativity filter in functionality, as it restricts some children nodes from existing for a given node. In the following example, the node *InfixExp2* will not have any node *InfixExp1* as a direct child, meaning that *InfixExp1* has an higher priority than *InfixExp2*.

```
priority :: TFilter String
priority = every (neg anyChildrenMatches `when` matches "
    ↪ InfixExp2")
 where anyChildrenMatches = (matches "InfixExp1" $$). (
     ↪ concatMap getChildren) . getChildren
```

### 4.6 Reject Filter

Regarding the reject filter, for a given node, we want the children not to match certain keywords. In the following example, the node *Id* (containing an identifier, for example a variable name) is ruled not to have any children containing the strings "true" or "false". Note that, because HaGLR is a scannerless parser, an auxiliary *implodeSubTree* function is needed to recover the string which is split up during parsing, and *reverse* is used for the same reason.

```
reject :: TFilter String
reject = every (neg stringMatchesKeywords `when` matches
     ↪ "Id")
 where stringMatchesKeywords = (matches (reverse "true")
      ↪ `orElse` matches (reverse "false")) . head .
      ↪ getChildren . implodeSubTree
```

## 4.7 Follow Filter

In the follow filter, we define that, if there are several values generated by a production, then they are in different nodes only when there is a separator between them. We do this by verifying that, when there are two nodes, either the first one ends with a separator or the second one starts with a separator. In this example, the node *Values1* represents a list of values.

```
follow :: TFilter String
follow = every ((firstEndsGood `orElse` secondStartsGood)
     ↪  `when` matches "Values1")
 where firstEndsGood =    isOf (not . isDigit . head .
      ↪ getLastTerm . (!!0) . getChildren)
       secondStartsGood = isOf (not . isDigit . head .
            ↪ getFirstTerm . (!!1) . getChildren)
```

## 4.8 Preference Filter

The preference filter is used to choose one production over another when both are valid. For this, we refer back to the subsection 3 where the dangling else problem is mentioned.

This filter, however, does not behave similarly to the other filters. It works by comparing different parse trees, and then choosing the best one, which is fundamentally different to the other filters which operate on a single tree. Therefore, the implementation of this filter is just a function which operates on lists. This function compares each parse tree to every other tree, and discards a parse tree if it is considered less interesting than any other one.

## 4.9 Context Dependent Filters

In our approach disambiguation rules can be used to implement already known disambiguation rules. However, they can also be used to implement new concepts and ideas that generally are not possible to implement in the disambiguation rules of most parser generators. This allows the developer to express any desired disambiguation rule - specific to the language the developer is defining - without the limitations of not being able to fine-tune the filters.

As an example, in this section, it will be presented a filter that associates any sum operations to the left, until an *if* clause is found, and inside the *if* blocks, the sum operations will associate to the right. While there is no immediate use for this filter , it is a good example of different behaviour implemented into a filter.

```
ff :: TFilter String
ff = iff (matches "Instr3") rAssocAll leftAssocUntil
 where rAssocAll = every (right_assoc "InfixExp2")
       leftAssocUntil = isOf (all (satisfies ff) .
            ↪ getChildren) `o` left_assoc "InfixExp2"
```

The *iff* combinator is fed three arguments, where the first is just to check if the current node matches the *if* instruction. If so, the *rAssocAll* portion of the code is run, which just applies the *right_assoc* (associate right) combinator to all the subtrees from that point onwards. If the matching fails, as seen in the *leftAssocUntil* portion of the code, the *left_assoc* (associate left) combinator is applied to the current node, and the whole filter is recursively applied to all the subtrees.

## 5 CONCLUSIONS

This paper presents a novel approach to express grammar disambiguation rules as abstract syntax tree filter combinators. Disambiguation rules are first-class citizens: new rules can be defined by combining existing ones and they can be also passed as input to parsers. As a result, grammar writers are not limited to a set of pre-defined rules offered by the parser generator, instead they can easily express new rules and experiment with them without having to re-generate a new parser.

We have developed a combinator library of such disambiguation rules and we have defined various rules with we validate by combining our Haskell-based filter combinators with the HaGLR parser generator: The AST forest produced by the HaGLR parser is pruned into a single correct AST by our disambiguation filters.

Because HaGLR was developed in a pedagogic setting, it does not use the most efficient data structures. Thus we are refactoring HaGLR so that uses a better representation for parse-tables (which steer the parser at runtime), and we are also considering to use shared packed parse forests[13] which use sharing to reduce runtime and memory consumption. We are also integrating the disambiguation filters with BiYacc [15], a tool for generating both a parser and a reflective printer [16] for an unambiguous context-free grammar. The use of disambiguation filters can be helpful in extending this tool to also support ambiguous context-free grammars, therefore increasing its expressiveness and allowing for more test cases to be supported by this tool.

## REFERENCES

[1] John W. Backus. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.. In *IFIP Congress* (2002-01-03). 125–131.
[2] João Fernandes, João Saraiva, and Joost Visser. 2004. Generalised LR Parsing in Haskell. In *Advanced Functional Programming (AFP'04) - Students Workshop.*
[3] ISO. 2011. IEC 9899: 2011 Information technology—Programming languages—C. *International Organization for Standardization, Geneva* 27 (2011), 59.
[4] Stephen C. Johnson. 1979. *Yacc: Yet Another Compiler-Compiler.*
[5] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. 2004. Generalised Parsing: Some Costs. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 89–103.
[6] Simon Marlow and Andy Gil. 2001. *Happy User Guide.*
[7] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '11).* ACM, New York, USA, 425–436.
[8] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices* 49, 10 (2014), 579–598.
[9] D. J. Salomon and G. V. Cormack. 1989. Scannerless NSLR(1) Parsing of Programming Languages. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89).* ACM, New York, NY, USA, 170–178.
[10] João Saraiva. 2002. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In *Proc. of the ACM Workshop on Functional and Declarative Programming in Education (FDPE) (Univ. of Kiel, TR 0210).* 133–140.
[11] Uwe Schmidt, Martin Schmidt, and Torben Kuseler. 2016. hxt: A collection of tools for processing XML with Haskell. https://github.com/UweSchmidt/hxt.
[12] Masaru Tomita. 1985. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Norwell, MA, USA.
[13] Masaru Tomita. 1985. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Vol. 8. Springer Science & Business Media.
[14] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. *Disambiguation Filters for Scannerless Generalized LR Parsers.* Springer Berlin Heidelberg, Berlin, Heidelberg, 143–158.
[15] Zirun Zhu, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2015. BiYacc: Roll Your Parser and Reflective Printer into One. In *Proc. of the 4th Int. Workshop on Bidirectional Transformations co-located with STAF 2015, L'Aquila, Italy, July 24, 2015.* 43–50.
[16] Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2016. Parsing and Reflective Printing, Bidirectionally. In *Proc. of the 2016 ACM SIGPLAN Int. Conf. on Software Language Engineering (SLE 2016).* ACM, New York, NY, USA, 2–14.