

InDubio: A Combinator Library to Disambiguate Ambiguous Grammars

José Nuno Macedo[1,2(✉)] and João Saraiva[1,2]

[1] University of Minho, Braga, Portugal
saraiva@di.uminho.pt
[2] HASLab/INESC TEC, Braga, Portugal
jose.n.macedo@inesctec.pt

Abstract. To infer an abstract model from source code is one of the main tasks of most software quality analysis methods. Such abstract model is called Abstract Syntax Tree and the inference task is called parsing. A parser is usually generated from a grammar specification of a (programming) language and it converts source code of that language into said abstract tree representation. Then, several techniques traverse this tree to assess the quality of the code (for example by computing source code metrics), or by building new data structures (e.g, flow graphs) to perform further analysis (such as, code cloning, dead code, etc). Parsing is a well established technique. In recent years, however, modern languages are inherently ambiguous which can only be fully handled by ambiguous grammars.

In this setting disambiguation rules, which are usually included as part of the grammar specification of the ambiguous language, need to be defined. This approach has a severe limitation: disambiguation rules are not first class citizens. Parser generators offer a small set of rules that can not be extended or changed. Thus, grammar writers are not able to manipulate nor define a new specific rule that the language he is considering requires.

In this paper we present a tool, name *InDubio*, that consists of an extensible combinator library of disambiguation filters together with a generalized parser generator for ambiguous grammars. *InDubio* defines a set of basic disambiguation rules as abstract syntax tree filters that can be combined into more powerful rules. Moreover, the filters are independent of the parser generator and parsing technology, and consequently, they can be easily extended and manipulated. This paper presents *InDubio* in detail and also presents our first experimental results.

Keywords: Parsing · Disambiguation filters · Combinators

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

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12252, pp. 1002–1018, 2020.
https://doi.org/10.1007/978-3-030-58811-3_71

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 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, described by a context-free grammar.

From such grammar, a parser generator 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 [5]. 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.

Listing 1.1. Grammar of arithmetic expressions

```
exp     : exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | '(' exp ')'
        | number ;
```

The ambiguous grammar presented in Listing 1.1 follows closely the syntax nature of arithmetic expressions. For example, the text `1+2+3` can produce two different trees, prioritizing either the left sum or the right sum.

Regular parser generators do not support ambiguous grammars. Thus, ambiguity has to be dealt with by either refactoring the grammar to eliminate ambiguity (which can be complex, and results in a more complex and hard to understand grammar) or by providing disambiguation rules.

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: a) 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; b) 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; c) It is not modular and changing a disambiguation rule results in the generation of a new parser; d) 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.

1.1 Motivation

In the early ages of programming languages, it was usual to purposely include certain symbols in a language's grammar so that the generated parser for said language was more efficient.

The most obvious example is found in the C programming language: the semicolon found at the end of each instruction is a statement terminator [15]. It is used to resolve some ambiguities that could be found in the grammar. However, modern programming languages tend to avoid the use of too many syntactic symbols. This 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 requires complex grammars and corresponding parsers to handle such programs.

This work focuses on generic parsing techniques that generate a parser for any context-free grammar. Due to the ambiguity, these parsers produce various results for the same program, and disambiguation rules are used to select the desired solution, but the existing tools that provide generic parsing techniques are still limited. These rules are to be implemented as combinators, which are simple code tools, that are easy to implement but very powerful when combined with other combinators.

2 State of the Art

There are various different parsing algorithms proposed to solve the parsing problem. The earliest solution to be used was to embed the grammar in the code, where there was no separation between the grammar and the rest of the code. This was a solution that made it very difficult to change the grammar once it was coded in, since it was mixed with the code.

2.1 BNF Notation

The BNF notation is a notation for specifying context-free grammar, generally used for specifying the exact grammar of programming languages. It was proposed by [4], to describe the language of what became known as ALGOL 59.

In the example in Listing 1.1, the BNF notation is used to describe the grammar of arithmetic expressions. This grammar is inherently recursive: an expression (Exp) can be defined as an expression (Exp), an arithmetic sign ('+')

and another expression (Exp). The entirety of a grammar is expressed in this way, which is simple to understand and powerful when compared to embedding the grammar in the code.

The BNF notation is extremely interesting as it laid the foundation to having the grammar separated from the code, such that it would be easy to change the grammar without having to change any of the remaining code.

However, this notation can be ambiguous. A concrete instance of text can be interpreted in different ways, all of them correct according to the specification. As an example, the text *1+2+3* can be considered. There are two possible ways to interpret this text according to the specification, which represent different order of operations, that is, whether the left sum or the right product is processed first.

2.2 Common Parsers

While it is common for programming languages to be expressed as grammars, for example using the BNF notation, there are various ways to generate a parser given such specification. Each alternative method has its own advantages and disadvantages. The first to be relevant were the most powerful in terms of compilation and execution time. As such, the grammars were changed to best fit the method: the developer had to both focus on writing the correct grammar and adapting it to fit the parser generator. The previous example uses left-recursion, which is impossible to parse [2] for some of these algorithms. Therefore, the previous example would have to be changed so as to not have left-recursion, before it could be used in some parser generators.

One of the most well-known parser generators, YACC, is a LALR parser generator [8]: this relies on a lightweight algorithm which was perfect for the time it was developed, that is, 1975, when it was much more necessary to restrict program runtime and memory size.

Another example of a popular parser generator is ANTLR [13], whose development started in 1989 as a LL(*) parser generator. The LL(*) algorithm allows for parsing decisions to be taken by looking at the following tokens in the input stream. ANTLR 4 uses the ALL(*)[14], which is $O(N^4)$ in theory but is shown to consistently perform linearly in practice.

While Yacc generates C code and ANTLR generates code for various programming languages, one of the most popular parser generators for Haskell is Happy [11], which enables the developer to supply a file with the specification of a grammar, and in turn generates a parser, that is, a module of code that can read text according to that grammar's specifications. Happy is part of the Haskell Platform, being one of the most famous Haskell parsing tools. Due to its rather big popularity and regular maintenance, it is a fairly well optimized tool.

2.3 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 could arise.

Ambiguity can be dealth with using GLR parsers, which are slower than their non-generalized counterparts, due to their additional flexibility in dealing with non-determinism: when faced with an input with several different possible outputs, a GLR parser [23] will produce all of the outputs instead of selecting one of them. If no non-determinism is present, a GLR parser will behave just like a LR parser [9], which is efficient. With the constant advances in technology, the limitations that made this technique undesirable are gone and there are parser generators that allow the use of the GLR algorithm, such as Happy.

However, GLR is not the only generalized algorithm. The GLL algorithm [21,22] is also generalized, but much less explored. This algorithm is worst-case cubic in both time and space and there are possible optimizations to it [1].

2.4 Scannerless Parsing

Generally speaking, parser applications are divided in two components: the lexer and the parser. The lexer takes the input and breaks it into a list of tokens [17], and then the parser takes those tokens and matches them with the production rules to produce the actual parsing result.

Scannerless parsing [16] consists of skipping the lexer entirely and treating each character from the input as a token, which is fed directly into the parser. Scannerless parsing removes the necessity of describing the tokens in the grammar specification, allowing the developer to write the grammar without worrying as much with conforming with the parser technology. Scannerless parsers are compositional, therefore allowing for two parsers to be merged without needing to change them.

2.5 Disambiguation Filters for Scannerless Generalized Parsing

In this work, the focus is on scannerless generalized parsers. For such, as they deal with ambiguous inputs, it is expected to get a list 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, instead of all the possibilities.

The task of processing the list of the ambiguous parse trees produced by a parser and removing the undesired is called disambiguation. Typically, such filtering is done on the parser, that is, modifying part of the parser so that the undesired interpretations cannot be produced. Some new rules for disambiguation are needed when dealing with scannerless parsing. In this section, some filters for disambiguation are presented and described, according to the work of van den Brand et al. [5].

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.

The **reject** filter enables the creation of keywords in the grammar. In other words, it rejects some productions from deriving into certain sequences. This filter allows for a clean implementation of reserved keywords. For example, in the C programming language, it shouldn't be allowed for a variable to be named "while", as that is a reserved keyword used in defining loops.

The **follow** filter solves an 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 interpretations of a given input but some are preferred over 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 a part of them. This filter is the go-to filter to remove the dangling else problem, occurs when there are two *if* clauses and only one *else* clause, and thus it is not clear to which *if* clause the *else* clause associates to.

2.6 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 [20] is used as a base for building combinators for filtering syntax trees. The HaGLR tool [7] is a Haskell implementation of a GLR parser generator, which was implemented with pedagogic purposes. It produces as result a pure parse tree forest, which is a list of parse trees. This is not the case for all generalized parsers as some optimizations change the representation of the parse forests to save memory, which use a different, more compact approach, but are less intuitive to work around.

3 The InDubio Combinator Library

In this paper, a new approach for parser disambiguation is described. 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 forest of AST, removing some or all of the ambiguities, according to what the developer specified. There are several advantages and disadvantages in using this process instead of the classical approach. Since the parser that is used is unmodified, it is less efficient, as the classical approach removes parts of the parser reducing thus the number of results the parser has to output. However, while the parser itself is less efficient, the development cycle of the developer is more efficient, as there is no need to constantly produce a new parser after a change 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.

To be able to build complex filters to disambiguate the result of a parser, we use basic combinators 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.

In the following sections, the types of filters described in Sect. 2.5 are implemented using these combinators. However, it is important to note that they apply to the parse trees produced by the 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.

In the remaining of this section, we will use the expression grammar *Filters* (available online at this work's repository) as running example.

Associativity Filter. Given a parser for the *Filters* grammar, and the simple input string *"1+2+3"*, the output consists of two parse trees, as seen in Fig. 1. The left one shall be considered the correct interpretation and the right one the undesired interpretation for this example. As such, to write the filter, it is needed to locate what is the pattern that stands out in this example, and then describe a way to remove it using the combinators.

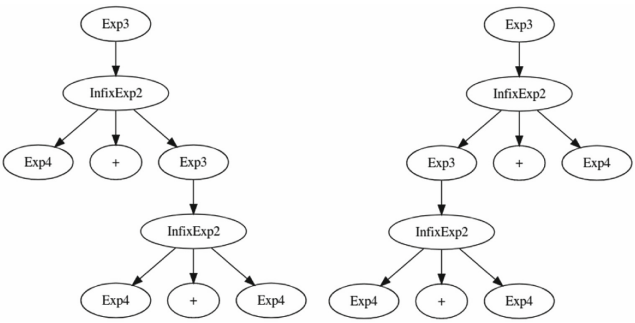


Fig. 1. Simplified parse trees for the string "1+2+3;"

The names of the nodes match with the names of the productions. As an example, the *Exp1* and *Exp2* nodes refer to the *Exp* production. *NTfromT2* refers to a terminal symbol, in this case, the + sign. *InfixExp2* refers to the

InfixExp production, which defines the additions. In this case, the difference between the two parse trees is in whether an *InfixExp2* node is to the left or to the right of their parent *InfixExp2* node. This represents the ambiguity, in which the addition can be left-associative or right-associative. Therefore, the ambiguity can be solved by implementing a filter that removes any tree where there is an *InfixExp2* node to the right (or left) of another *InfixExp2* node.

The implementation of this filter is rather trivial once the algorithm is defined. This filter will look at the root node, check if it is an *InfixExp2*, and, if it is, check if there is an *InfixExp2* node in the right child of said node. If there is not, then the tree is correct according to the filter.

```
associativity :: TFilter String
associativity = 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"*. Of course, when the root does not match this string, the filter does nothing.

However, this filter does not work as expected on a real parse tree, because the filter only applies to the root. In reality, the ambiguity can exist deep into the tree, and so the *every* combinator is needed to apply the filter to all the tree, and discard the tree if any of the nodes fail to satisfy it.

Finally, we can generalize the resulting filter. This can be done by having the node's name passed as an argument, and not have it hard-coded in the definition.

```
left_assoc :: String -> TFilter String
left_assoc  p = neg (matches p . head . getChildren . (!!2) .
   ↪  getChildren) `when` matches p

assocGeneric :: TFilter String
assocGeneric = every (left_assoc "InfixExp2")
```

Priority Filter. Given a parser for the *Filters* grammar, and the simple input string *"1+2*3"*, the output consists of two parse trees, as seen in Fig. 2.

These trees are rather similar to the ones displayed before, and the node naming conventions are the same. In this case, one of the parse trees represents the + symbol being processed first, while in the other tree is the * symbol that is processed first. To resolve this ambiguity, we use a simple algorithm: for any node in the tree, if it matches the * symbol, then the children nodes must not match the + symbol. If such nodes exist, they represent a situation where the product happens before the sum, which is not the desired behaviour.

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

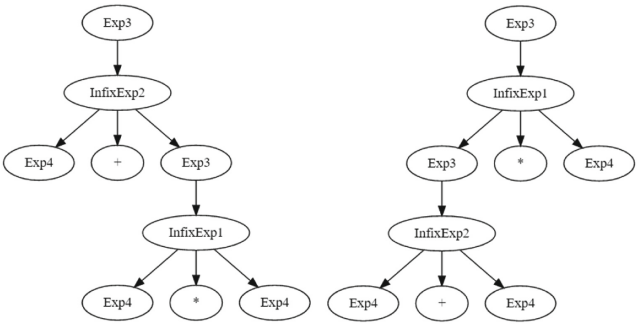


Fig. 2. Simplified parse trees for the string "1+2*3;".

As in previous filters, the *every* combinator is needed to apply the filter to all the nodes in the tree, so that it discards the tree if any of the nodes fail to satisfy it. Finally, it is possible to generalize this filter, so as to allow easier reuse.

```
before :: String -> String -> TFilter String
before x y = neg ((matches  x $$). (concatMap getChildren) .
    ↪ getChildren ) `when` matches  y

priorityGeneric :: TFilter String
priorityGeneric = every (before "InfixExp1" "InfixExp2")
```

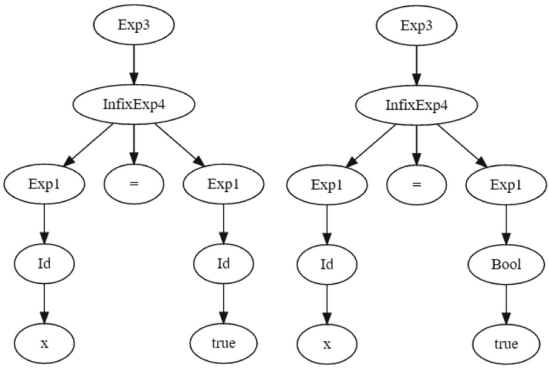


Fig. 3. Simplified parse trees for the string "x = true;".

Reject Filter. Given a parser for the *Filters* grammar, and the simple input string *"x = true"*, the output consists of two parse trees, as seen in Fig. 3. The parser can interpret the string *true* as either a boolean value, that is, a *"Bool"* node, or an identifier, that is, an *"Id"* node. While both are technically

correct, this ambiguity needs to be eliminated by not allowing the use of language keywords as language identifiers, thus the need for reject filters.

Since this parser is a scannerless parser, a string is actually a sequence of tokens, where each token is a character. To recover the string from the sequence of tokens, the *implodeSubTree* function is used. It is part of the tools available in the HaGLR parser, and it converts a sequence of tokens into a string. The resulting string is reversed, and thus the *reverse* function is used to fix the strings and enable adequate comparison between them.

According to the definition of this filter, when a node matches the *"Id"* production, then the string it derives into cannot match any of the desired keywords. The following implementation only considers the keywords *"true"* and *"false"* but it trivially generalizes to more language keywords.

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

As in previous disambiguation filter combinators, the *every* combinator is used to apply this filter to the whole tree. Of course, only the *"Id"* nodes can be affected by it, but they can be located anywhere on the tree, hence why this combinator is needed. One last step is to generalize this filter.

```
reject :: String -> String -> TFilter String
reject w p = neg (matches (reverse w) . head . getChildren .
   ↪ implodeSubTree) `when` matches p

rejGeneric :: TFilter String
rejGeneric=every(reject "true" "Id" `o` reject "false" "Id")
```

Follow Filter. Given a parser for the *Filters* grammar, and the simple input string *"int x [] = [12 3]"*, the output consists of two parse trees, as seen in Fig. 4. Since whitespace is not directly specified in the grammar, if there are not any other separators, the parser cannot distinguish whether the string *"12"* is one or two values. As such, the follow filters are used to solve this ambiguity.

In this situation, it is desired that the *"Values"* production has the longest match in each of its children, that is, tries to incorporate as many characters as possible into each children. In this situation, it is enough to specify that either the left child ends with a character that is not a number, or the right child starts with a value that is not a number. This forces the output to only contain this production when there are two values separated by a whitespace in the input, because if there are no whitespaces, the filter will reject the whole tree.

The implementation of this filter just specifies that, when the root node is a *"Values1"* node, then the first child's processed string must end with a desired character or the second child's processed string must start with one, so that between the two, there is at least one whitespace. It is important to note that the parser assembles whitespaces into various nodes automatically, and it is taken advantage of this fact to describe this filter.

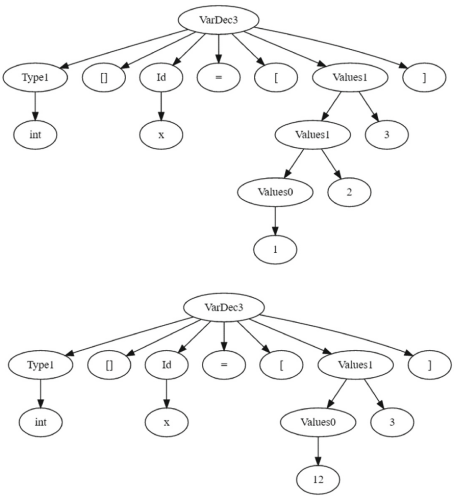


Fig. 4. Simplified parse trees for the string "int [] x = [12 3];".

```
follow :: TFilter String
follow = (p1 `orElse` p2) `when` matches "Values1"
 where p1=isOf$not.isDigit.head.getLastTerm.(!!0).getChildren
       p2=isOf$not.isDigit.head.getFirstTerm.(!!1).getChildren
```

As before, the *every* combinator is used to apply this filter to the whole tree. Of course, only the *"Values1"* nodes can be affected by it, but they can be located anywhere on the tree, hence why this combinator is needed. One last step is to generalize this filter.

```
follow :: String -> String -> TFilter String
follow  t r = (p1 `orElse` p2) `when` matches t
 where p1=isOf$flip notElem r.head.getLastTerm.(!!0).
    ↪ getChildren
  p2=isOf$flip notElem r.head.getFirstTerm.(!!1).getChildren

followGeneric :: TFilter String
followGeneric =  every ( follow "Values1" "0123456789")
```

Preference Filter. Given a parser for the *Filters* grammar, and the simple input string *"if true then if false then 1 else 2"*, the output consists of two parse trees, as seen in Fig. 5. This is a rather famous ambiguity problem generally described as the *dangling else* problem. In this input string, there are two *if...then* clauses, and one *else* keyword, but the grammar allows the else to belong to either *if*. Therefore, the parser will generate two interpretations, in which the *else* keyword will associate with either of the *if...then* clauses.

In this situation, both interpretations are correct, but the developer may prefer one over the other. This is where the preference filter comes in.

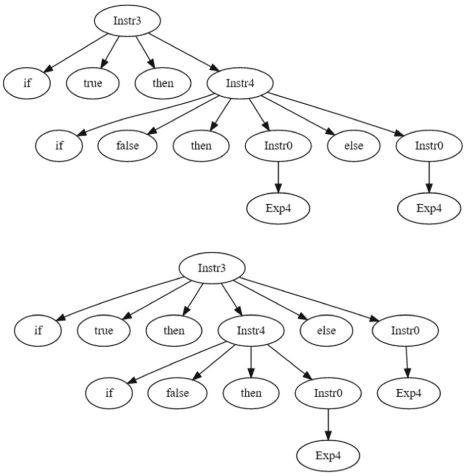


Fig. 5. Simplified parse trees for the string "if true then if false then 1 else 2;".

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 a simple function on filters, and the source code is available at the repository.

3.1 Context Dependent Filters

In our approach, disambiguation rules are embedded in the grammar description. As we presented in the previous sections, they 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 in our running example language, it is a good example of different behaviour implemented into a filter.

```
ff :: TFilter String
ff = iff (matches "Instr3") rAssocAll leftUntil
 where rAssocAll = every (right_assoc "InfixExp2")
       leftUntil = 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 *rightAssocEverything*

portion of the code is run, which just applies the *right_assoc* combinator shown before to all the subtrees from that point onwards. If the matching fails, as seen in the *leftAssocUntil* portion of the code, the *left_assoc* combinator is applied to the current node, and the whole filter is recursively applied to all the subtrees.

3.2 Embedded Filters

To allow an easier usage of these disambiguation filters, an easier way to use the well-known filters was developed. A new data type was developed, where each possible constructor represents a disambiguation rule, and a developer can specify the filters using this data type, coupled with a conversion function that transforms it into a filter that would be produced by using the equivalent combinators. The developer does not need to know the names of the productions to write the filters, and can use the symbols instead, which is generally more intuitive. However, the helper function that does this convertion is computationally intense, and so this approach is less efficient than the pure combinator approach. It is possible, however, to combine this approach with the combinator approach, allowing a developer to write the most trivial filters with this approach, and to write the most complex filters using combinators.

The following filter is an implementation of most of the filters previously described, combined into one filter, using this strategy. The priority filter is changed into a filter that defined priority between the modulus and equality operations, as these filters will be needed in the following section. This change is trivial using this strategy, as there is no need to know the production names, only the symbols that are relevant. The preference filter is not included as it cannot be described using this strategy.

```
filters :: TFilter String
filters = every $ gen_filters [LeftAssoc "+",
    "Values1" `NotFollowedBy` "0123456789",
    Reject_List ["true", "false", "break"] "Id",
    ["=="] `Before` ["%"]  ]
```

4 Disambiguation Filters in Practice

To test and validate our combinators[1], we have defined several grammars in HaGLR and expressed the disambiguation rules using our combinators: we consider the well known ambiguous expression grammar (a fragment of this grammar is shown in Sect. 1) and the Tiger grammar [3] (a real programming language defined with teaching purposes). For these two examples the grammars are pure CFGs: no alien notation for disambiguation rules is included. Moreover, the grammars are ambiguous since no care on expressing operator rules via non-terminals was considered.

[1] The work presented in this paper is available at our repository, at https://bitbucket.org/zenunomacedo/disambiguation-filters/.

Since the parsers produced by HaGLR for these two ambiguous grammars produce a forest of ASTs, we used our library to express the disambiguation rules for the two languages in order to select a single correct AST. Moreover, to validate that the selected tree is the correct one, we also express both languages via an equivalent non-ambiguous grammar. Those equivalent grammars were obtained by using grammar adaptation rules [10]. Thus, we compare the single AST produced by the non-ambiguous grammar, with the select AST produced by our filters from the AST forest produced by the HaGLR parser.

To automate this validation we used the QuickCheck [6] test case generator and property-based test framework: we define generators for random expressions, and used a set of pre-defined Tiger programs as input for our parsers. Then, we express a property to confirm that the resulting ASTs are equivalent: they produce semantically equivalent representations. For the expression grammars we developed an evaluator (that computes the result of the expression), and for the Tiger language we defined a translator to low-level code. Next, we show the QuickCheck property for the expression example.

```
prop_valid :: String -> Property
prop_valid x = forAll genExpr $ \e ->
 let rAmbig = disambiguation_small $ AP.glr_parser_aterm e
     rUnambig = UP.glr_parser_aterm e
 in singletonList rAmbig && singletonList rUnambig &&
 eval (head $ rAmbig) == eval (head $ rUnambig)
```

QuickCheck is then able to randomly generate inputs for the expression language and test this property for all of them: it states that both the ambiguous and non-ambiguous parsers produce a single result for the input, and that the semantic result of evaluating both expressions is equal. A similar property is defined for Tiger with the only difference of not randomly generating inputs, but instead selecting them from a pre-defined list. Thus, Quickcheck was used to confirm that both grammars produce equivalent semantic results.

4.1 Performance

As stated before, the HaGLR was developed with pedagogic purposes only. For example, it uses Haskell pre-defined lists to model parsing tables which is easy to understand but it is not the best data structure if we focus on performance. Thus, the resulting parsers are not optimized at all (unlike for example the parser produced by Happy [11]). Nevertheless, we performed some preliminary benchmarks to measure the cost of applying the disambiguation filters to the forest produced by the HaGLR parsers. For the two considered grammars we have the following results produced by Criterion [12] (a Haskell benchmark framework) (Table 1):

This table contains a single input for each grammar: an expression with 11 operators, and the Tiger program modelling the n-Queens problem (both available from our repository).

For the Expression grammar, applying the disambiguation filters reduces the parsing time by 17.6%, and for the Tiger grammar, applying the filters increases

Table 1. Comparison of the runtime without and with filters, in seconds, for the Expression and Tiger parsers.

	Expression	Tiger
No filters	5.811	2.119
With filters	4.789	2.829

the parsing time by 33.5%. We can conclude that using the filters improves the parsing performance when the grammar is highly ambiguous, because of Haskell's lazy evaluation, as the filters cut off the generation of unnecessary ASTs. However, for less ambiguous grammars such as Tiger's, the filters perform unnecessary verifications of non-ambiguous parts of the ASTs, which has a negative impact on performance.

5 Conclusions

In this paper, disambiguation rules are first-class citizens: new rules can be defined by combining existing ones and they can be also passed has input to parsers. As a result, grammar writers are not limited to a set of pre-defined and fixed 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 InDubio: a combinator library of such disambiguation rules and we have defined rules to disambiguate both the well-known ambiguous expression language, and the Tiger language. We validate our library 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. We presented our first preliminary benchmark results showing that although the generated parsers are non-optimized, for highly ambiguous grammars and due to the lazy evaluation of Haskell, by applying the filters, we actually produce faster parsers: lazy evaluation avoids the creation of unnecessary and redundant ASTs.

Because HaGLR was developed in a pedagogic setting, it does not use the most efficient data structures. Thus, we are optimizing HaGLR so that it uses a better parse-table representation, and we are also considering to use shared packed parse forests [24] and data structure free compilation [18], which both use sharing to reduce runtime and memory consumption. We are also integrating the disambiguation filters with BiYacc [25], a tool for generating both a parser and a reflective printer [26,27] 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. Moreover, we also plan to combine our approach with Generic attribute grammars [19].

References

1. Afroozeh, A., Izmaylova, A.: Faster, practical GLL parsing. In: Franke, B. (ed.) CC 2015. LNCS, vol. 9031, pp. 89–108. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46663-6_5
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, San Francisco (2006)
3. Appel, A.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
4. Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In: IFIP Congress, pp. 125–131 (1959)
5. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_12
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. ACM Sigplan Not. **46**(4), 53–64 (2011)
7. Fernandes, J.P., Saraiva, J., Visser, J.: Generalised LR Parsing in Haskell (2004)
8. Johnson, S.C.: Yacc: yet another compiler-compiler (1979)
9. Johnstone, A., Scott, E., Economopoulos, G.: Generalised parsing: some costs, March 2004
10. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_32
11. Marlow, S., Gil, A.: Happy user guide (2001)
12. O'Sullivan, B.: Criterion: a Haskell microbenchmarking library (2009). http://www.serpentine.com/criterion/
13. Parr, T., Fisher, K.: Ll(*): the foundation of the ANTLR parser generator. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 425–436. ACM, New York (2011)
14. Parr, T., Harwell, S., Fisher, K.: Adaptive ll (*) parsing: the power of dynamic analysis. ACM SIGPLAN Not. **49**(10), 579–598 (2014)
15. Perlis, A.J., Shaw, M., Sayward, F. (eds.): Software Metrics: An Analysis and Evaluation. MIT Press, Cambridge (1981)
16. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI 1989), pp. 170–178. ACM (1989)
17. Saraiva, J.: HaLeX: a Haskell library to model, manipulate and animate regular languages. In: Hanus, M., Krishnamurthi, S., Thompson, S. (eds.) Proceedings of the ACM Workshop on Functional and Declarative Programming in Education, pp. 133–140. University of Kiel Technical report 0210, September 2002
18. Saraiva, J., Swierstra, D.: Data Structure Free Compilation. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 1–16. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-49051-7_1
19. Saraiva, J., Swierstra, D.: Generic attribute grammars. In: Parigot, D., Mernik, M. (eds.) 2nd Workshop on Attribute Grammars and their Applications, WAGA 1999, pp. 185–204. INRIA Rocquencourt, March 1999
20. Schmidt, U., Schmidt, M., Kuseler, T.: HXT: a collection of tools for processing xml with Haskell (2016). https://github.com/UweSchmidt/hxt

21. Scott, E., Johnstone, A.: Gll parsing. Electron. Notes Theoret. Comput. Sci. **253**(7), 177–189 (2010)
22. Scott, E., Johnstone, A.: Gll parse-tree generation. Sci. Comput. Program. **78**(10), 1828–1844 (2013)
23. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell (1985)
24. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, vol. 8. Springer Science & Business Media, New York (1985)
25. Zhu, Z., Ko, H., Martins, P., Saraiva, J., Hu, Z.: Biyacc: roll your parser and reflective printer into one. In: Proceedings of the 4th International Workshop on Bidirectional, L'Aquila, Italy, 24 July 2015, pp. 43–50 (2015)
26. Zhu, Z., Ko, H.-S., Zhang, Y., Martins, P., Saraiva, J., Hu, Z.: Unifying parsing and reflective printing for fully disambiguated grammars. New Gener. Comput. **38**(3), 423–476 (2020). https://doi.org/10.1007/s00354-019-00082-y
27. Zhu, Z., Zhang, Y., Ko, H.S., Martins, P., Saraiva, J.A., Hu, Z.: Parsing and reflective printing, bidirectionally. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, pp. 2–14. ACM (2016)