Ambiguity detection for programming language grammars

Basten, H.J.S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Parse Forest Diagnostics with DR. AMBIGUITY

“A fool sees not the same tree that a wise man sees.”
William Blake

Once an ambiguity detection method finds an unwanted ambiguity, it should be removed from the grammar. However, it is not always straightforward for the grammar developer to see which modifications solve his ambiguity and in which way. In this chapter we present an expert system called Dr. Ambiguity, that can automatically propose applicable cures for an ambiguous sentence. After giving an overview of different causes of ambiguity and ambiguity resolutions, the internals of Dr. Ambiguity are described. The chapter ends with a small experimental validation of the usefulness of the expert system, by applying it on a realistic character-level grammar for Java.

7.1 Introduction

This work is motivated by the use of parsers generated from general context-free grammars (CFGs). General parsing algorithms such as GLR and derivates [Tom85, vdBSVV02, AH99, BG06, Eco06], GLL [SJ10, JS11], and Earley [Ear70, Sch06] support parser generation for highly non-deterministic context-free grammars. The advantages of constructing parsers using such technology are that grammars may be modular and that real programming languages...
Chapter 7. Parse Forest Diagnostics with DR. AMBIGUITY

Figure 7.1: The complexity of a parse forest for a trivial Java class with one method; the indicated subtree is an ambiguous if-with-dangling-else issue (180 nodes, 195 edges).

(often requiring parser non-determinism) can be dealt with efficiently\(^1\). It is common to use general parsing algorithms in (legacy) language reverse engineering, where a language is given but parsers have to be reconstructed [LV01], and in language extension, where a base language is given which needs to be extended with unforeseen syntactical constructs [BTV06].

The major disadvantage of general parsing is that multiple parse trees may be produced by a parser. In this case, the grammar was not only non-deterministic, but also ambiguous. We say that a grammar is ambiguous if it generates more than one parse tree for a particular input sentence. Static detection of ambiguity in CFGs is undecidable in general [Can62, Flo62, CS63].

It is not an overstatement to say that ambiguity is the Achilles’ heel of CFG-general parsing. Most grammar engineers who are building a parser for a programming language intend it to produce a single tree for each input program. They use a general parsing algorithm to efficiently overcome problematic non-determinism, while ambiguity is an unintentional and unpredictable side-effect. Other parsing technologies, for example Ford’s PEG [For04] and Parr’s LL(*) [PF11], do not report ambiguity. Nevertheless, these technologies also employ disambiguation techniques (ordered choice, dynamic lookahead). In combination with a debug-mode that does produce all derivations, the results in this chapter should be beneficial for these parsing techniques as well. It should help the user to intentionally select a disambiguation method. In any case, the point of departure for this chapter is any parsing algorithm that will produce all possible parse trees for an input sentence.

In Chapters 3–5 we present a fast ambiguity detection approach that combines approximative and exhaustive techniques. The output of this method are the ambiguous sentences found in the language of a tested grammar. Nevertheless, this is only an observation that the patient is ill, and now we need a cure. We therefore will diagnose the sets of parse trees produced for specific ambiguous sentences. The following is a typical grammar engineering scenario:

1. While testing or using a generated parser, or after having run a static ambiguity detection tool, we discover that one particular sentence leads to a set of multiple parse trees. This

\(^1\)Linear behavior is usually approached and most algorithms can obtain cubic time worst time complexity [Lan74]
7.1. Introduction

If ( \( < \) ExprName ( Id ( "a" ) ),
IfExists ( IfElse ( IfElse ( > ExprName ( Id ( "a" ) ),
> If ( ExprName ( Id ( "b" ) ), ExprName ( Id ( "b" ) ),
ExprStmt ( ExprStmt ( Invoke ( Invoke ( Method ( MethodName ( Id ( "a" ) ) ), Method ( MethodName ( Id ( "a" ) ) ),
[ [ ] ] ), Method ( MethodName ( Id ( "b" ) ) ), ExprStmt ( ExprStmt ( Invoke ( Invoke ( Method ( MethodName ( Id ( "b" ) ) ), Method ( MethodName ( Id ( "b" ) ) ),
[ [ ] ] ) ) )

Figure 7.2: Using `diff -side-by-side` to diagnose a trivial ambiguous syntax tree for a dangling else in Java (excerpts of Figure 7.1).

set is encoded as a single parse forest with choice nodes where sub-sentences have alternative sub-trees.

2. The parser reports the location in the input sentence of each choice node. Note that such choice nodes may be nested. Each choice node might be caused by a different ambiguity in the CFG.

3. The grammar engineer extracts an arbitrary ambiguous sub-sentence and runs the parser again using the respective sub-parser, producing a set of smaller trees.

4. Each parse tree of this set is visualized on a 2D plane and the grammar engineer spots the differences, or a (tree) diff algorithm is run by the grammar engineer to spot the differences. Between two alternative trees, either the shape of the tree is totally different (rules have moved up/down, left/right), or completely different rules have been used, or both. As a result the output of diff algorithms and 2D visualizations typically require some effort to understand. Figure 7.1 illustrates the complexity of an ambiguous parse forest for a 5 line Java program that has a dangling else ambiguity. Figure 7.2 depicts the output of diff on a strongly simplified representation (abstract syntax tree) of the two alternative parse trees for the same nested conditional. Realistic parse trees are not only too complex to display here, but are often too big to visualize on screen as well. The common solution is to prune the input sentence step-by-step to eventually reach a very minimal example that still triggers the ambiguity but is small enough to inspect.

5. The grammar engineer hopefully knows that for some patterns of differences there are typical solutions. A solution is picked, and the parser is regenerated.

6. The smaller sentence is parsed again to test if only one tree (and which tree) is produced.
7. The original sentence is parsed again to see if all ambiguity has been removed or perhaps more diagnostics are needed for another ambiguous sub-sentence. Typically, in programs one cause of ambiguity would lead to several instances distributed over the source file. One disambiguation may therefore fix more “ambiguities” in a source file.

The issues we address in this chapter are that the above scenario is (a) an expert job, (b) time consuming and (c) tedious. We investigate the invention of an expert system that can automate finding a concise grammar-level explanation for any choice node in a parse forest and propose a set of solutions that will eliminate it. This expert system is shaped as a set of algorithms that analyze sets of alternative parse trees, simulating what an expert would do when confronted with an ambiguity.

The contributions of this chapter are an overview of common causes of ambiguity in grammars for programming language (Section 7.3), an automated tool (Dr. Ambiguity) that diagnoses parse forests to propose one or more appropriate disambiguation techniques (Section 7.4) and an initial evaluation of its effectiveness (Section 7.5). In 2006 we published a manual [Vin11] to help users disambiguate SDF2 grammars. This well-read manual contains recipes for solving ambiguity in grammars for programming languages. Dr. Ambiguity automates all tasks that users perform when applying the recipes from this manual, except for finally adding the preferred disambiguation declaration.

7.1.1 Preliminaries

In this chapter we will use the following definitions: A context-free grammar \( G \) is defined as a 4-tuple \((T, N, P, S)\), namely finite sets of terminal symbols \( T \) and non-terminal symbols \( N \), production rules \( P \) in \( A \times (T \cup N)^* \) written like \( A \rightarrow \alpha \), and a start symbol \( S \). A sentential form is a finite string in \((T \cup N)^*\). A sentence is a sentential form without non-terminal symbols. An \( \varepsilon \) denotes the empty string. We use the other lowercase greek characters \( \alpha, \beta, \gamma, \ldots \) for variables over sentential forms, uppercase roman characters for non-terminals \((A, B, \ldots)\) and lowercase roman characters and numerical operators for terminals \((a, b, +, -, *, /)\). By applying production rules as substitutions we can generate new sentential forms. One substitution is called a derivation step, e.g. \( \alpha A \beta \Rightarrow \alpha \gamma \beta \) with rule \( A \rightarrow \gamma \). We use \( \Rightarrow^* \) to denote sequences of derivation steps. A full derivation is a sequence of production rule applications that starts with a start symbol and ends with a sentence. The language of a grammar is the set of all sentences derivable from \( S \). In a bracketed derivation [GH67] we record each application of a rule by a pair of brackets, for example \( S \Rightarrow (\alpha E \beta) \Rightarrow (\alpha (E + E) \beta) \Rightarrow (\alpha ((E * E) + E) \beta) \). Brackets are (implicitly) indexed with their corresponding rule.

A non-deterministic derivation sequence is a derivation sequence in which a \( \circ \) operator records choices between different derivation sequences. I.e. \( \alpha \Rightarrow (\beta) \circ (\gamma) \) means that either \( \beta \) or \( \gamma \) may be derived from \( \alpha \) using a single derivation step. Note that \( \beta \) does not necessarily need to be different from \( \gamma \). An example non-deterministic derivation is \( E \Rightarrow (E + E) \circ (E * E) \Rightarrow (E + (E * E)) \circ ((E + E) * E) \). A cyclic derivation sequence is any
sequence \( \alpha \Rightarrow^+ \alpha \), which is only possible by applying rules that do not have to eventually generate terminal symbols, such as \( A \rightarrow A \) and \( A \rightarrow \varepsilon \).

A parse tree is an (ordered) finite tree representation of a bracketed full derivation of a specific sentence. Each pair of brackets is represented by an internal node labeled with the rule that was applied. Each leaf node is labeled with a terminal. This implies that the leafs of a parse tree form a sentence. Note that a single parse tree may represent several equivalent derivation sequences. Namely in sentential forms with several non-terminals one may always choose which non-terminal to expand first. From here on we assume a canonical left-most form for such equivalent derivation sequences, in which expansion always occurs at the left-most non-terminal in a sentential form.

A parse forest is a set of parse trees possibly extended with ambiguity nodes for each use of choice (\( \cdot \)). Like parse trees, parse forests are limited to represent full derivations of a single sentence, each child of an ambiguity node is a derivation for the same sub-sentence. One such child is called an alternative. For simplicity’s sake, and without loss of generality, we assume that all ambiguity nodes have exactly two alternatives.

A parse forest is ambiguous if it contains at least one ambiguity node. A sentence is ambiguous if its parse forest is ambiguous. A grammar is ambiguous if it can generate at least one ambiguous sentence. An ambiguity in a sentence is an ambiguity node. An ambiguity of a grammar is the cause of such aforementioned ambiguity. We define cause of ambiguity precisely in Section 7.3. Note that cyclic derivation sequences can be represented by parse forests by allowing them to be graphs instead of just trees [Rek92].

A recognizer for \( G \) is a terminating function that takes any sentence \( \alpha \) as input and returns true if and only if \( S \Rightarrow^* \alpha \). A parser for \( G \) is a terminating function that takes any finite sentence \( \alpha \) as input and returns an error if the corresponding recognizer would not return true, and otherwise returns a parse forest for \( \alpha \). A disambiguation filter is a function that takes a parse forest for \( \alpha \) and returns a smaller parse forest for \( \alpha \) [KV94]. A disambiguator is a function that takes a parser and returns a parser that produces smaller parse forests. Disambiguators may be implemented as parser actions, or by parser generators that take additional disambiguation constructs as input [vdBSVV02]. We use the term disambiguation for both disambiguation filters and disambiguators.

### 7.2 Solutions to Ambiguity

There are basically two kinds of solutions to removing ambiguity from grammars. The first involves restructuring the grammar to accept the same set of sentences but using different rules. The second leaves the grammar as-is, but adds disambiguations (see above). Although grammar restructuring is a valid solution direction, we restrict ourselves to disambiguations described below. The benefit of disambiguation as opposed to grammar restructuring is that the shape of the rules, and thus the shape of the parse trees remains unchanged. This allows language engineers to maintain the intended semantic structure of the language, keeping parse trees directly related to abstract syntax trees (or even synonymous) [HHKR89].

Any solution may be language preserving, or not. We may change a grammar to have it generate a different language, or we may change it to generate the same language differently. Similarly, a disambiguation may remove sentences from a language, or simply remove some
ambiguous derivation without removing a sentence. This depends on whether or not the filter is applied always in the context of an ambiguous sentence, i.e. whether another tree is guaranteed to be left over after a certain tree is filtered. It may be hard for a language engineer who adds a disambiguation to understand whether it is actually language preserving. Whether or not it is good to be language preserving depends entirely on ad-hoc requirements. We therefore do not answer this question. Where possible, we do indicate whether adding a certain disambiguation is expected to be language preserving. Proving this property is out-of-scope.

Solving ambiguity is sometimes confused with making parsers deterministic. From the perspective of this chapter, non-determinism is a non-issue. We focus solely on solutions to ambiguity.

We now quote a number of disambiguation methods here. Conceptually, the following list contains nothing but disambiguation methods that are commonly supported by lexer and parser generators [ASU86]. Still, the precise semantics of each method we present here may be specific to the parser frameworks of SDF2 [HHKR89, Vis97] and Rascal [KvdSV11]. In particular, some of these methods are specific to scannerless parsing, where a context-free grammar specifies the language down to the character level [Vis97, SC89]. We recommend [BBV07], to appreciate the intricate differences between semantics of operator priority mechanisms between parser generators.

**Priority** disallows certain direct edges between pairs of rules in parse trees in order to affect operator priority. For instance, the production for the + operator may not be a direct child of the * production [vdBSVV02].

Formally, let a priority relation > be a partial order between recursive rules of an expression grammar. If \( A \rightarrow \alpha_1 A \alpha_2 > A \rightarrow \beta_1 A \beta_2 \) then all derivations \( \gamma A \delta \Rightarrow \gamma (\alpha_1 \alpha_2) \delta \Rightarrow \gamma (\beta_1 \beta_2) \alpha_2 \) are illegal.

**Associativity** is similar to priority, but father and child are the same rule. It can be used to affect operator associativity. For instance, the production of the + operator may not be a direct right child of itself because + is left associative [vdBSVV02]. Left and right associativity are duals, and non-associativity means no nesting is allowed at all. Formally, if a recursive rule \( A \rightarrow A A \) is defined left associative, then any derivation \( \gamma A \delta \Rightarrow \gamma (A A) \delta \Rightarrow \gamma (A \alpha A) \delta \) is illegal.

**Offside** disallows certain derivations using the would-be indentation level of an (indirect) child. If the child is “left” of a certain parent, the derivation is filtered [Lan66]. One example formalization is to let \( \Pi(x) \) compute the start column of the sub-sentence generated by a sentential form \( x \) and let > define a partial order between production rules. Then, if \( A \rightarrow \alpha_1 X \alpha_2 > B \rightarrow \beta \) then all derivations \( \gamma A \delta \Rightarrow \gamma (\alpha_1 X \alpha_2) \delta \Rightarrow^* \gamma (\alpha_1 (\ldots \beta \ldots) \alpha_2) \delta \) are illegal if \( \Pi(\beta) < \Pi(\alpha_1) \). Parsers may employ subtly different offside disambiguators, depending on how \( \Pi \) is defined for each different language or even for each different production rule within a language.

**Preference** removes a derivation, but only if another one of higher preference is present. Again, we take a partial ordering > that defines preference between rules for the same non-terminal. Let \( A \rightarrow \alpha > A \rightarrow \beta \), then from all derivations \( \gamma A \delta \Rightarrow \gamma (\alpha) \delta \Rightarrow \gamma (\alpha) (\beta) \delta \) we must remove \( (\beta) \) to obtain \( A \Rightarrow \gamma (\alpha) \delta \).
Reserve disallows a fixed set of terminals from a certain (non-)terminal, commonly used to reserve keywords from identifiers. Let \( K \) be a set of sentences and let \( I \) be a non-terminal from which they are declared to be reserved. Then, for every \( \alpha \in K \), any derivation \( I \Rightarrow^* \alpha \) is illegal.

Reject disallows the language of a certain non-terminal from that of another one. This may be used to implement Reserve, but it is more powerful than that [vdBSVV02]. Let \((I - R)\) declare that the non-terminal \( R \) is rejected from the non-terminal \( I \). Then any derivation sequence \( I \Rightarrow^* \alpha \) is illegal if and only if \( R \Rightarrow^* \alpha \).

Not Follow/Precede declarations disallow derivation steps if the generated sub-sentence in its context is immediately followed/preceded by a certain terminal. This is used to affect longest match behavior for regular languages, but also to solve “dangling else” by not allowing the short version of \( \text{if} \), when it would be immediately followed by \( \text{else} \) [vdBSVV02]. Formally, we define follow declaration as follows. Given \( A \Rightarrow^* \alpha \) and a declaration \( A \text{ not-follow } \beta \), where \( \beta \) is a sentence, any derivation \( S \Rightarrow^* \gamma A \beta \delta \Rightarrow^* \gamma(\alpha) \beta \delta \) is illegal. We should mention that Follow declarations may simulate the effect of “shift before reduce” heuristics that deterministic — LR, LALR — parsers use when confronted with a shift/reduce conflict.

Dynamic Reserve disallows a dynamic set of sub-sentences from a certain non-terminal, i.e. using a symbol table [ASU86]. The semantics is similar to Reject, where the set \( R \) is dynamically changed as certain derivations (i.e. type declarations) are applied.

Types removes certain type-incorrect sub-trees using a type-checker, leaving correctly typed trees as-is [BVVV05]. Let \( C(d) \) be true if and only if derivation \( d \) (represented by a tree) is a type-correct part of a program. Then all derivations \( \gamma A \delta \Rightarrow \gamma(\alpha) \delta \) are illegal if \( C(\alpha) \) is false.

Heuristics There are many kinds of heuristic disambiguation that we bundle under a single definition here. The preference of “Islands” over “Water” in island grammars is an example [Moo01]. Preference filters are sometimes generalized by counting the number of preferred rules as well [vdBSVV02]. Counting rules is used sometimes to choose a “simplest” derivation, i.e. the most shallow trees are selected over deeper ones. Formally, Let \( C(d) \) be any function that maps a derivation (parse tree) to an integer. If \( C(A \Rightarrow \alpha) > C(A \Rightarrow \beta) \) then from all derivations \( A \Rightarrow^* (\alpha) \diamond (\beta) \) we must remove \( (\beta) \) to obtain \( A \Rightarrow (\alpha) \).

Not surprisingly, each kind of disambiguation characterizes certain properties of derivations. In the following section we link such properties to causes of ambiguity. Apart from Types and Heuristics (which are too general to automatically report specific suggestions for), we can then link the causes explicitly back to the solution types.
7.3 Causes of Ambiguity

Ambiguity is caused by the fact that the grammar can derive the same sentence in at least two ways. This is not a particularly interesting cause, since it characterizes all ambiguity in general. We are interested in explaining to a grammar engineer what is wrong for a very particular grammar and sentence and how to possibly solve this particular issue. We are interested in the root causes of specific occurrences of choice nodes in parse forests.

For example, let us consider a particular grammar for the C programming language for which the sub-sentence "\{ S * b; \}" is ambiguous. In one derivation it is a block of a single statement that multiplies variables \( S \) and \( b \), in another it is a block of a single declaration of a pointer variable \( b \) to something of type \( S \). From a language engineer’s perspective, the causes of this ambiguous sentence are that:

- "\*" is used both in the rule that defines multiplication, and in the rule that defines pointer types, and
- type names and variable names have the same lexical syntax, and
- blocks of code start with a possibly empty list of declarations and end with a possibly empty list of statements, and
- both statements and declarations end with "\;".

The conjunction of all these causes explains us why there is an ambiguity. The removal of just one of them fixes it. In fact, we know that for C the ambiguity was fixed by introducing a disambiguator that reserves any declared type name from variable names using a symbol table at parse time, effectively removing the second cause.

We now define a cause of an ambiguity in a sub-sentence to be the existence of any edge that is in the parse tree of one alternative of an ambiguity node, but not in the other. In other words, each difference between two alternative parse trees in a forest is one cause of the ambiguity. For example, two parse tree edges differ if they represent the application of a different production rule, span a different part of the ambiguous sub-sentence, or are located at different heights in the tree.

We define an explanation of an ambiguity in a sentence to be the conjunction of all causes of ambiguity in a sentence. An explanation is a set of differences. We call it an explanation because an ambiguity exists if and only if all of its causes exist. A solution is any change to the grammar, addition of a disambiguation filter or use of a disambiguator that removes at least one of the causes.

Some causes of ambiguity may be solvable by the disambiguation methods defined in Section 7.2, some may not. Our goals are therefore to first explain the cause of ambiguity as concisely as possible, and then if possible propose a palette of applicable disambiguations. Note that even though the given common disambiguations have limited scope, disambiguation in general is always possible by writing a disambiguation filter in any computationally complete programming language.
7.3. Causes of Ambiguity

Figure 7.3: Euler diagram showing the categorization of parse tree differences.

7.3.1 Classes of Parse Tree Differences

Having defined ambiguity and the causes thereof, we can now categorize different kinds of causes into classes of differences between parse trees. The difference classes are the theory behind the workings of Dr. Ambiguity (Section 7.5). Figure 7.3 summarizes the cause classes that we will identify in the following.

For completeness we should explain that ambiguity of CFGs is normally bisected into a class called \texttt{HORIZON TAL} ambiguity and a class called \texttt{VERTICAL} ambiguity [BGM10, AL90, Sch01]. \texttt{VERTICAL} contains all the ambiguity that causes parse forests that have two different production rules directly under a choice node. For instance, all edges of derivation sequences of form $\gamma A \delta \Rightarrow \gamma (\alpha) \circ (\beta) \delta$ provided that $\alpha \neq \beta$ are in \texttt{VERTICAL}. \texttt{VERTICAL} clearly identifies a difference class, namely the trees with different edges directly under a choice node.

\texttt{HORIZON TAL} ambiguity is defined to be all the other ambiguity. \texttt{HORIZON TAL} does not identify any difference class, since it just implies that the two top rules are the same. Our previous example of ambiguity in a C grammar is an example of such ambiguity. We conclude that in order to obtain full explanations of ambiguity the \texttt{HORIZON TAL/VERTICAL} dichotomy is not detailed enough. \texttt{VERTICAL} provides only a partial explanation (a single cause), while \texttt{HORIZON TAL} provides no explanations at all.

We now introduce a number of difference classes with the intention of characterizing differences which can be solved by one of the aforementioned disambiguation methods. Each element in a different class points to a single cause of ambiguity. A particular disambiguation method may be applicable in the presence of elements in one or more of these classes.

The \texttt{EDGES} class is the universe of all difference classes. In \texttt{EDGES} are all single derivation
steps (equivalent to edges in parse forests) that occur in one alternative but not in the other. If no such derivation steps exist, the two alternatives are exactly equal. Note that \(\text{EDGES} = \text{HORIZONTAL} \cup \text{VERTICAL}\).

The **TERMINALS** class contains all parse tree edges to non-\(\varepsilon\) leaves that occur in one alternative but not in the other. If an explanation contains a difference in TERMINALS, we know that the alternatives have used different terminal tokens—or in the case of scannerless, different character classes—for the same sub-sentences. This is sometimes called *lexical ambiguity*. If no differences are in TERMINALS, we know that the terminals used in each alternative are equal.

The **WHITESPACE** class (\(\subset\) TERMINALS) simply identifies the differences in TERMINALS that produce terminals consisting of nothing but spaces, tabs, newlines, carriage returns or linefeeds.

The **REGEXPS** class contains all edges of derivation steps that replace a non-terminal by a sentential form that generates a regular language, occurring in one derivation but not in the other, i.e. \(A \Rightarrow (\rho)\) where \(\rho\) is a regular expression over terminals. Of course, TERMINALS \(\subset\) REGEXPS. In character level grammars (scannerless [vdBSVV02]), the REGEXPS class often represents lexical ambiguity. Differences in REGEXPS may point to solutions such as Reserve, Follow and Reject, since longest match and keyword reservation are typical solution scenarios for ambiguity on the lexical level.

In the **SWAPS** class we put all edges that have a corresponding edge in the other alternative of which the source and target productions are equal but have swapped order. For instance, the lower edges in the parse tree fragment \(((E + E) + E) \cdot (E + (E + E))\) are in SWAPS. If all differences are in SWAPS, the set of rules used in the derivations of both alternatives are the same and each rule is applied the same number of times—only their order of application is different.

The **SAME** class is the subset of edges in SWAPS that have the same source and target productions. In this case, the only difference between two corresponding edges are the substrings they span. For instance, the lower edges in the parse tree fragment \(((E + E) + E) \cdot (E + (E + E))\) are in SAME. Differences in this class typically require Associativity solutions.

The **REORDERINGS** class generalizes SWAPS with more than two rules to permute. This may happen when rules are not directly recursive, but mutually recursive in longer chains. Differences in REORDERINGS or SWAPS obviously suggest a Priority solution, but especially for non-directly recursive derivations Priority will not work. For example, the notorious “dangling else” issue [ASU86] generates differences in application order of mutually recursive statements and lists of statements. For some grammars, a difference in REORDERINGS may also imply a difference in VERTICAL, i.e. a choice between an if with an else and one without. In this case a Preference solution would work. Some grammars (e.g. the IBM COBOL VS2 standard) only have differences in HORIZONTAL and REORDERINGS. In this case a Follow solution may prevent the use of the if
7.4. Diagnosing Ambiguity

without the else if there is an else to be parsed. Note that the Offside solution is an alternative method to remove ambiguity caused by REORDERINGS. Apparently, we need even smaller classes of differences before we can be more precise about suggesting a solution.

The LISTS class contains differences in the length of certain lists between two alternatives. For instance, we consider rules $L \to LE$ and observe differences in the amount of times these rules are applied by the derivation steps in each alternative. More precisely, for any $L$ and $E$ with the rule $L \to LE$ we find chains of edges for derivation sequences $\alpha L \beta \Rightarrow \alpha LE \beta \Rightarrow \Rightarrow^{*} \alpha LE^{+} \beta$, and compute their length. The edges of such chains of different lengths in the two alternatives are members of LISTS. Examples of ambiguities caused by LISTS are those caused by not having “longest match” behavior: an identifier “aa” generated using the rules $I \to a$ and $I \to Ia$ may be split up in two shorter identifiers “a” and “a” in another alternative. We can say that LISTS $\cap$ REGEPS $\neq \emptyset$.

Note that differences in LISTS $\cap$ REORDERINGS indicate a solution towards Follow or Offside for they flag issues commonly seen in dangling constructs. On the other hand a difference in LISTS $\setminus$ REORDERINGS indicates that there must be another important difference to explain the ambiguity. The “‘(S * a)’” ambiguity in C is of that sort, since the length of declaration and statement lists differ between the two alternatives, while also differences in TERMINALS are necessary.

The EPSILONS class contains all edges to $\varepsilon$ leaf nodes that only occur in one of the alternatives. They correspond to derivation steps $\alpha A \beta \Rightarrow \alpha() \beta$, using $A \to \varepsilon$. All cyclic derivations are caused by differences in EPSILONS because one of the alternatives of a cyclic ambiguity must derive the empty sub-sentence, while the other eventually loops back. However, differences in EPSILONS may also cause other ambiguity than cyclic derivations.

The OPTIONALS class (\subset EPSILONS) contains all edges of a derivation step $\alpha A \beta \Rightarrow \alpha() \beta$ that only exist in one alternative, while a corresponding edge of $\delta A \zeta \Rightarrow \delta(\gamma) \zeta$ only exists in the other alternative. Problems that are solved using longest match (Follow) are commonly caused by optional whitespace for example.

7.4 Diagnosing Ambiguity

We provide an overview of the architecture and the algorithms of DR AMBIGUITY in this section. In Section 7.5 we demonstrate its output on example parse forests for an ambiguous Java grammar.

7.4.1 Architecture

Figure 7.4 shows an overview of our diagnostics tool: DR AMBIGUITY. We start from the parse forest of an ambiguous sentence that is either encountered by a language engineer or produced by a static ambiguity detection tool like AMBIDEXTER. Then, either the user
points at a specific sub-sentence\(^2\), or DR. AMBIGUITY finds all ambiguous sub-sentences (e.g. choice nodes) and iterates over them. For each choice node, the tool then generates all unique combinations of two children of the choice node and applies a number of specialized diff algorithms to them.

Conceptually there exists one diff algorithm per disambiguation method (Section 7.2).

\(^2\) We use Eclipse IMP [CFJ09] as a platform for generating editors for programming languages defined using RASCAL [KvdSV11]. IMP provides contextual pop-up menus.
However, since some methods may share intermediate analyses there is some additional intermediate stages and some data-dependency that is not depicted in Figure 7.4. These intermediate stages output information messages about the larger difference classes that are to be analyzed further if possible. This output is called “Classification Information” in Figure 7.4. The other output, called “Disambiguation Suggestions” is a list of specific disambiguation solutions (with reference to specific production rules from the grammar).

If no specific or meaningful disambiguation method is proposed the classification information will provide the user with useful information on designing an ad-hoc disambiguation.

DR. AMBIGUITY is written in the RASCAL domain specific programming language [KvdSV11]. This language is specifically targeted at analysis, transformation, generation and visualization of source code. Parse trees are a built-in data-type which can be queried using (higher order) pattern matching, visiting and set, list and map comprehension facilities. To understand some of the RASCAL snippets in this section, please familiarize yourself with this definition for parse trees (as introduced by [Vis97]):

```
data Tree = appl(Production prod, list[Tree] args) // production nodes
         | amb(set[Tree] alternatives) // choice nodes
         | char(int code); // terminal leaves

data Production = prod(Symbol lhs, list[Symbol] rhs, Attributes atts); // rules
```

DR. AMBIGUITY, in total, is 250 lines of RASCAL code that queries and traverses terms of this parse tree format. The count includes source code comments. It is slow on big parse forests\(^3\), which is why the aforementioned user-selection of specific sub-sentences is important.

### 7.4.2 Algorithms

Here we show some of the actual source code of DR. AMBIGUITY.

First, the following two small functions iterate over all (deeply nested) choice nodes (amb) and over all possible pairs of alternatives. This code uses deep match (/), set matching, and set or list comprehensions. Note that the match operator (::=) iterates over all possible matches of a value against a pattern, thus generating all different bindings for the free variables in the pattern. This feature is used often in the implementation of DR. AMBIGUITY.

```
list[Message] diagnose(Tree t) {
    return [findCauses(x) | x <- {a | /a:amb(_) := t}];
}
list[Message] findCauses(Tree a) {
    return [findCauses(x, y) | {x, y, _} := a.alternatives];
}
```

\(^3\)The current implementation of RASCAL lacks many trivial optimizations.
The following functions each implement one of the diff algorithms from Figure 7.4. The following two (slightly simplified⁴) functions detect opportunities to apply priority or associativity disambiguations.

```plaintext
list[Message] priorityCauses(Tree x, Tree y) {
    if ((/appl(p,[appl(q,_),_*]) := x,
         /t:appl(q,[_*,appl(p,_)]) := y, p != q) {
        return [error("You might add this priority rule: <p>|<q>"),
                error("You might add this associativity group:
                        + "left:<p>|<q>")];
    }
    return [];
}

list[Message] associativityCauses(Tree x, Tree y) {
    if ((/appl(p,[appl(p,_),_*]) := x,
         /Tree t:appl(p,[_*,appl(p,_)]) := y) {
        return [error("You might add this associativity declaration:
                        + "left:<p>")];
    }
    return [];
}
```

Both functions “simultaneously” search through the two alternative parse trees p and q, detecting a vertical swap of two different rules p and q (priority) or a horizontal swap of the same rule p under itself (associativity).

This slightly more involved function detects dangling-else and proposes a follow restriction as a solution:

```plaintext
list[Message] danglingCauses(Tree x, Tree y) {
    if (appl(p,/appl(q,_)) := x, appl(q,/appl(p,_)) := y) {
        return danglingOffsideSolutions(x, y)
               + danglingFollowSolutions(x, y);
    }
    return [];
}

list[Message] danglingFollowSolutions(Tree x, Tree y) {
    if (prod(_, rhs, _) := x.prod,
        prod(_, [prefix*, _, l:lit(_), more*], _) := y.prod,
        rhs == prefix) {
        return [error("You might add a follow restriction on:
                        + "<x.prod*>")];
    }
    return [];
}
```

⁴We have removed references to location information that facilitates IDE features.
The function `danglingCauses` detects re-orderings of arbitrary depth, after which the outermost productions are compared by `danglingFollowSolutions` to see if one production is a prefix of the other.

**DR. AMBIGUITY** currently contains 10 such functions, and we will probably add more. Since they all employ the same style — (a) simultaneous deep match, (b) production comparison and (c) construction of a feedback message — we have not included more source code\(^5\).

### 7.4.3 Discussion on Correctness

These diagnostics algorithms are typically wrong if one of the following four errors is made:

- no suggestion is given, even though the ambiguity is of a quite common kind;
- the given suggestion does not resolve any ambiguity;
- the given suggestion removes both alternatives from the forest, resulting in an empty forest (i.e., it removes the sentence from the language and is thus not language preserving);
- the given suggestion removes the proper derivation, but also unintentionally removes sentences from the language.

We address the first threat by demonstrating DR. AMBIGUITY on Java in Section 7.5. However, we do believe that the number of detection algorithms is open in principle. For instance, for any disambiguation method that characterizes a specific way of solving ambiguity we may have a function to analyze the characteristic kind of difference. As an “expert tool”, automating proposals for common solutions in language design, we feel that an open-ended solution is warranted. More disambiguation suggestion algorithms will be added as more language designs are made. Still, in the next section we will demonstrate that the current set of algorithms is complete for all disambiguations applied to a scannerless definition of Java 5 [BVdGD11], which actually uses all disambiguations offered by SDF2.

For the second and third threats, we claim that no currently proposed solution removes both alternatives and all proposed solutions remove at least one. This is the case because each suggestion is solely deduced from a *difference* between two alternatives, and each disambiguation removes an artifact that is only present in one of the alternatives. We are considering to actually prove this, but only after more usability studies.

The final threat is an important weakness of DR. AMBIGUITY, inherited from the strength of the given disambiguation solutions. In principle and in practice, the application of rejects, follow restrictions, or semantic actions in general renders the entire parsing process stronger than context-free. For example, using context-free grammars with additional disambiguations we may decide language membership of many non-context-free languages. On the one hand, this property is beneficial, because we want to parse programming languages that have no or awkward context-free grammars. On the other hand, this property is cumbersome, since we

\(^5\)The source code is available at http://svn.rascal- mpl.org/rascal/trunk/src/org/rascalmpl/library/Ambiguity.rsc.
Chapter 7. Parse Forest Diagnostics with DR. AMBIGUITY

Disambiguations | Grammar snippet (Rascal notation)
--- | ---
7 levels of expression priority | Expr = Expr 
> "++" Expr
1 father/child removal | MethodSpec = Expr callee 
"." TypeArgs? 
Id { if (callee is ExprName) filter; }
9 associativity groups | Expr = left ( Expr 
| Expr "+" Expr 
| Expr "-" Expr )
10 rejects | ID = ( [$A-Z_a-z] [$0-9A-Z_a-z]* ) \ 
Keywords
30 follow restrictions | "+" = [\+] !>> [\+]
4 vertical preferences | Stm = @prefer "if" (" Expr ")" Stm 
| "if" (" Expr ")" Stm "else" Stm

Table 7.1: Disambiguations applied in the Java 5 grammar [BVdGD11]

can not easily predict or characterize the effect of a disambiguation filter on the accepted set of sentences.

Only in the SWAPS class, and its sub-classes we may be (fairly) confident that we do not remove unforeseen sentences from a language by introducing a disambiguation. The reason is that if one of the alternatives is present in the forest, the other is guaranteed to be also there. The running assumption is that the other derivation has not been filtered by some other disambiguation. We might validate this assumption automatically in many cases. So, application of priority and associativity rules suggested by DR. AMBIGUITY are safe if no other disambiguations are applied.

7.5 Demonstration

In this section we evaluate the effectiveness of DR. AMBIGUITY as a tool. We applied DR. AMBIGUITY to a scannerless (character level) grammar for Java [BVdGD11, BTV06]. This well tested grammar was written in SDF2 by Bravenboer et al. and makes ample use of its disambiguation facilities. For the experiment here we automatically transformed the SDF2 grammar to RASCAL’s EBNF-like form.

Table 7.1 summarizes which disambiguations were applied in this grammar. RASCAL supports all disambiguation features of SDF2, but some disambiguation filters are implemented as libraries rather than built-in features. The @prefer attribute is interpreted by a library function for example. Also, in SDF2 one can (mis)use a non-transitive priority to remove a direct father/child relation from the grammar. In RASCAL we use a semantic action for this.

7.5.1 Evaluation Method

DR. AMBIGUITY is effective if it can explain the existence of a significant amount of choice nodes in parse forests and proposes the right fixes. We measure this effectiveness in terms
Table 7.2: Precision/Recall results for each experiment, including (P)riority, (A)ssociativity, (R)eject, (F)ollow restrictions, (A)c(t)ions filtering edges, (A)void/prefer suggestions, and (O)ffside rule. For each experiment, the figures of the removed disambiguation are highlighted.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Diagnoses</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Remove priority between &quot;*&quot; and &quot;+&quot;</td>
<td>1 1 0 0 0 1 0</td>
<td>33%</td>
<td>100%</td>
</tr>
<tr>
<td>2. Remove associativity for &quot;+&quot;</td>
<td>0 1 0 0 0 0 0</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>3. Remove reservation of \texttt{true} keyword from \texttt{ID}</td>
<td>0 0 1 0 0 1 0</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>4. Remove longest match for identifiers</td>
<td>0 0 0 6 0 0 0</td>
<td>16%</td>
<td>100%</td>
</tr>
<tr>
<td>5. Remove package name vs. field access priority</td>
<td>0 0 0 0 6 1 0</td>
<td>14%</td>
<td>100%</td>
</tr>
<tr>
<td>6. Remove vertical preference for dangling else</td>
<td>0 0 0 1 14 1 1</td>
<td>7%</td>
<td>100%</td>
</tr>
<tr>
<td>7. All the above changes at the same time</td>
<td>1 2 1 7 20 4 1</td>
<td>17%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 7.2: Precision/Recall results for each experiment, including (P)riority, (A)ssociativity, (R)eject, (F)ollow restrictions, (A)c(t)ions filtering edges, (A)void/prefer suggestions, and (O)ffside rule. For each experiment, the figures of the removed disambiguation are highlighted.

of precision and recall. \textsc{Dr. Ambiguity} has high precision if it does not propose too many solutions that are useless or meaningless to the language engineer. It has high recall if it finds all the solutions that the language engineer deems necessary. Our evaluation method is as follows:

- The set of disambiguations that Bravenboer applied to his Java grammar is our “golden standard”.
- The disambiguations in the grammar are selectively removed, which results in different ambiguous versions of the grammar. New parsers are generated for each version.
- An example Java program is parsed with each newly generated parser. The program is unambiguous for the original grammar, but becomes ambiguous for each altered version of the grammar.
- We measure the total amount and which kinds of suggestions are made by \textsc{Dr. Ambiguity} for the parse forests of each grammar version, and compute the precision and recall.

Precision is computed by \[ \frac{|\text{Found Disambiguations} \cap \text{Removed Disambiguations}|}{|\text{Found Disambiguations}|} \times 100\%. \] We expect low precision, around 50%, because each particular ambiguity often has many different solution types. Low precision is not necessarily a bad thing, provided the total amount of disambiguation suggestions remains human-checkable.

Recall is computed by \[ \frac{|\text{Found Disambiguations} \cap \text{Removed Disambiguations}|}{|\text{Removed Disambiguations}|} \times 100\%. \] From this number we see how much we have missed. We expect the recall to be 100% in our experiments, since we designed our detection methods specifically for the disambiguation techniques of SDF2.

### 7.5.2 Results

Table 7.2 contains the results of measuring the precision and recall on a number of experiments. Each experiment corresponds to a removal of one or more disambiguation constructs and the
Chapter 7. Parse Forest Diagnostics with DR. AMBIGUITY

Figure 7.5: DR. AMBIGUITY reports diagnostics in the RASCAL language workbench.

parsing of a single Java program file that triggers the introduced ambiguity.

Table 7.2 shows that we indeed always find the removed disambiguation among the suggestions. Also, we always find more than one suggestion (the second experiment is the only exception).

The dangling-else ambiguity of experiment 6 introduces many small differences between two alternatives, which is why many (arbitrary) semantic actions are proposed to solve these. We may learn from this that semantic actions need to be presented to the language engineer as a last resort. For these disambiguations the risk of collateral damage (a non-language preserving disambiguation) is also quite high.

The final experiment tests whether the simultaneous analysis of different choice nodes that are present in a parse forest may lead to a loss of precision or recall. The results show that we find exactly the same suggestions. Also, as expected the precision of such an experiment is very low. Note however, that DR. AMBIGUITY reports each disambiguation suggestion per choice node, and thus the precision is usually perceived per choice node and never as an aggregated value over an entire source file. Figure 7.5 depicts how DR. AMBIGUITY may report its output.
7.5.3 Discussion

We have demonstrated the effectiveness of DR. AMBIGUITY for only one grammar. Moreover this grammar already contained disambiguations that we have removed, simultaneously creating a representative case and a golden standard.

We may question whether DR. AMBIGUITY would do well on grammars that have not been written with any disambiguation construct in mind. We may also question whether DR. AMBIGUITY works well on completely different grammars, such as for COBOL or PL/I. More experimental evaluation is warranted. Nevertheless, this initial evaluation based on Java looks promising and does not invalidate our approach.

Regarding the relatively low precision, we claimed that this is indeed wanted in many cases. The actual resolution of an ambiguity is a language design question. DR. AMBIGUITY should not a priori promote a particular disambiguation over another well known disambiguation. For example, reverse engineers have a general dislike of the offside rule because it complicates the construction of a parser, while the users of a domain specific language may applaud the sparing use of bracket literals.

7.6 Conclusions

We have presented theory and practice of automatically diagnosing the causes of ambiguity in context-free grammars for programming languages and of proposing disambiguation solutions. We have evaluated our prototype implementation on an actively used and mature grammar for Java 5, to show that DR. AMBIGUITY can indeed propose the proper disambiguations.

Future work on this subject includes further extension, further usability study and finally proofs of correctness. To support development of front-ends for many programming languages and domain specific languages, we will include DR. AMBIGUITY in releases of the RASCAL IDE (a software language workbench).