Parse Forest Diagnostics with Dr. Ambiguity

H. J. S. Basten and J. J. Vinju

Centrum Wiskunde & Informatica (CWI)
Science Park 123, 1098 XG Amsterdam, The Netherlands
{Jurgen.Vinju,Bas.Basten}@cwi.nl

Abstract In this paper we propose and evaluate a method for locating causes of ambiguity in context-free grammars by automatic analysis of parse forests. A parse forest is the set of parse trees of an ambiguous sentence. Deducing causes of ambiguity from observing parse forests is hard for grammar engineers because of (a) the size of the parse forests, (b) the complex shape of parse forests, and (c) the diversity of causes of ambiguity.
We first analyze the diversity of ambiguities in grammars for programming languages and the diversity of solutions to these ambiguities. Then we introduce Dr. Ambiguity: a parse forest diagnostics tools that explains the causes of ambiguity by analyzing differences between parse trees and proposes solutions. We demonstrate its effectiveness using a small experiment with a grammar for Java 5.

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 [33,9,3,6,16], GLL [32,20], and Earley [15,30] 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 (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 [23], and in language extension, where a base language is given which needs to be extended with unforeseen syntactical constructs [10].

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 generates more than one parse tree for a particular input sentence. Static detection of ambiguity in CFGs is undecidable in general [14].

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.

[1] Linear behavior is usually approached and most algorithms can obtain cubic time worst time complexity [31]

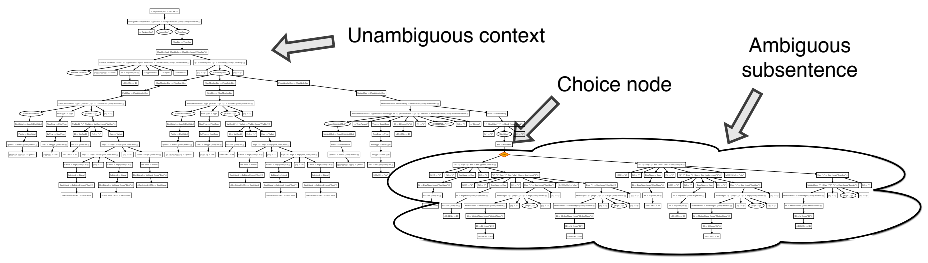


Figure 1. The complexity of a parse forest for a trivial Java class with one method; the indicated subtree is an ambiguous if-with-dangling-else issue (180 nodes, 195 edges).

```
If (                                   <
ExprName ( Id ( "a" ) ),               <
IfElse (                                 IfElse (
                                       > ExprName ( Id ( "a" ) ),
                                       > If (
ExprName ( Id ( "b" ) ),                 ExprName ( Id ( "b" ) ),
ExprStm (                                ExprStm (
Invoke (                                 Invoke (
Method ( MethodName ( Id ( "a" ) ) ),    Method ( MethodName ( Id ( "a" ) ) ),
[                                        [
] ) ),                                 | ] ) ) ),
ExprStm (                                ExprStm (
Invoke (                                 Invoke (
Method ( MethodName ( Id ( "b" ) ) ),    Method ( MethodName ( Id ( "b" ) ) ),
[                                        [
] ) ) ) )                              | ] ) ) )
```

Figure 2. Using `diff -side-by-side` to diagnose a trivial ambiguous syntax tree for a dangling else in Java (excerpts of Figure 1).

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 [17] and Parr's LL(*) [26], 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 paper 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 the current paper is any parsing algorithm that will produce all possible parse trees for an input sentence.

In other papers [4,5] we presented 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 a 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 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 1 illustrates the complexity of an ambiguous parse forest for a 5 line Java program that has a dangling else ambiguity. Figure 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 in this paper, 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 paper 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 paper are an overview of common causes of ambiguity in grammars for programming language (Section 3), an automated tool (Dr. Ambiguity) that diagnoses parse forests to propose one or more appropriate disambiguation techniques (Section 4) and an initial evaluation of its effectiveness (Section 5). In 2006 we published a manual [34] 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.

We need 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 like $N \to \alpha$ where $\alpha \in (T \cup N)^*$ 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 ϵ denotes the empty sentential form. 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 $A \to \gamma$. We use $\Rightarrow^*$ to denote sequences of derivation steps. A *full derivation* is a sequence of production rule application 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* [18] we record each application of a rule by a pair of brackets, for example $S \Rightarrow (bEe) \Rightarrow (b(E + E)e) \Rightarrow (b((E * E) + E))$. Brackets are (implicitly) indexed with their corresponding rule.

A *non-deterministic derivation sequence* is a derivation sequence in which a $\diamond$ operator records choices between different derivation sequences. I.e. $\alpha \Rightarrow (\beta) \ \diamond \ (\gamma)$ means that either β or γ may be derived from α using a single derivation step. Note that β does not necessarily need to be different from γ. An example non-deterministic derivation is $E \Rightarrow (E+E) \diamond (E*E) \Rightarrow (E+(E*E)) \diamond ((E + E) * E)$. 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 $A \to A$ and $A \to \epsilon$.

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 terminal is a leaf node. This implies 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 $(\diamond)$. 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 3. Note that *cyclic derivation* sequences can be represented by parse forests by allowing them to be graphs instead of just trees [27].

A *recognizer* for G is a terminating function that takes any sentence α 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 α as input and returns an error if the corresponding recognizer would not return true, and otherwise returns a *parse forest* for α. A *disambiguation filter* is a function that takes a parse forest for α and returns a smaller parse forest for α [22]. 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 which take additional disambiguation constructs as input [9]. We use the term *disambiguation* for both disambiguation filters and disambiguators.

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 in the current paper. 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) [19].

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. The current paper does 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 the current paper, 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 [1]. Still, the precise semantics of each method we present here may be specific to the parser frameworks of SDF2 [19,35] and Rascal [21]. In particular, some of these methods are specific to *scannerless* parsing, where a context-free grammar specifies the language down to the character level [35,28]. We recommend [7], 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 [9].
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 $\gamma A \delta \Rightarrow \gamma(\alpha_1 A \alpha_2)\delta \Rightarrow \gamma(\alpha_1(\beta_1 A \beta_2)\alpha_2)$ 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 [9]. Left and right associativity are duals, and *non-assocativity* means no nesting is allowed at all. Formally, if a recursive rule $A \rightarrow A\alpha A$ is defined left associative, then any derivation $\gamma A \delta \Rightarrow \gamma(A\alpha A)\delta \Rightarrow \gamma(A\alpha(A\alpha A))\delta$ 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 [24]. 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 Π 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 $\gamma A \delta \Rightarrow \gamma((\alpha) \diamond (\beta))\delta$ we must remove (β) 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 a language generated from a non-terminal for a certain non-terminal. This may be used to implement *Reserve*, but it is more powerful than that [9]. 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 $\exists(R \Rightarrow^* \alpha)$.

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 `if`, when it would be immediately followed by `else` [9]. Formally, we define follow declaration as follows. Given $A \Rightarrow^* \beta$ and a declaration A **not-follow** α, where α is a sentence, any derivation $S \Rightarrow^* \gamma A \alpha \delta \Rightarrow^* \gamma(\beta)\alpha\delta$ 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 [1]. The semantics is similar to *Reject*, where the set K 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 [12]. 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 [25]. Preference filters are sometimes generalized by counting the number of preferred rules as well [9]. 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 (β) 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.

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 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 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.

3.1 Classes of Parse Tree Differences

Having precisely 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 5). The upper part of Figure 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 HORIZONTAL ambiguity and a class called VERTICAL ambiguity [8,2,29]. VERTICAL contains all the ambiguity that causes parse forests that have two different production rules directly under a choice node. For

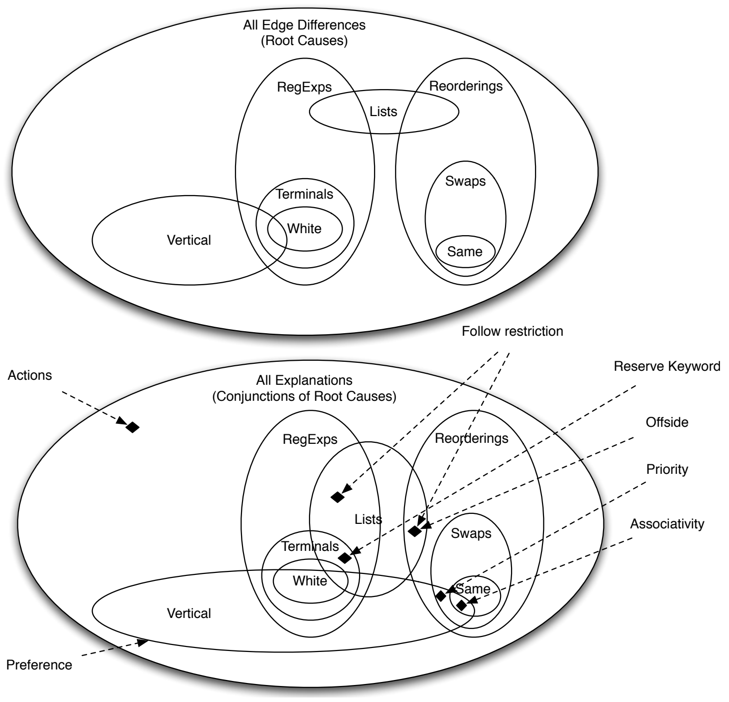


Figure 3. A partial categorization of parse tree differences (Venn diagrams). The categorization is complete for the disambiguation solutions in this paper. Above single causes are categorized. Below conjunctions of causes that form explanations are categorized.

instance, all edges of derivation sequences of form $\gamma A\delta \Rightarrow \gamma((\alpha)\diamond(\beta))\delta$ provided that $\alpha \neq \beta$ are in VERTICAL. VERTICAL clearly identifies a difference class, namely the trees with different edges directly under a choice node.

HORIZONTAL ambiguity is defined to be all the other ambiguity. HORIZONTAL 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 HORIZONTAL/VERTICAL dichotomy is not detailed enough. VERTICAL provides only a partial explanation (a single cause), while HORIZONTAL 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 following categorization is summarized by *the upper part of* Figure 3.

We define the EDGES class to be the universe of all difference classes. In 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 EDGES = HORIZONTAL $\cup$ VERTICAL.

The TERMINALS *class* contains all parse tree edges to non-ϵ leafs 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 nonterminal by a sentential form that generates a regular language, occurring in one derivation but not in the other, i.e. $A \Rightarrow (\rho)$ where ρ is a regular expression over terminals. Of course, TERMINALS $\subset$ REGEXPS. In character level grammars (scannerless [9]), the REGEXPS class 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 that have swapped order. For instance, the lower edges in the parse tree fragment $((E * E) + E) \diamond (E * (E + E))$ 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 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 [1] 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` 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 $L \rightarrow LE$ 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 $L \rightarrow LE$ we find chains of edges for derivation sequences $\alpha L\beta \Rightarrow \alpha LE\beta \Rightarrow \alpha LEE\beta \Rightarrow^* \alpha E^+\beta$, 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 $I \rightarrow a$ and $I \rightarrow I\ a$ may be split up in two shorter identifiers "a" and "a" in another alternative. We can say that LISTS $\cap$ REGEXPS $\neq \emptyset$.

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 ϵ leaf nodes that only occur in one of the alternatives. They correspond to derivation steps $\alpha A\beta \Rightarrow \alpha()\beta$, using $A \rightarrow \epsilon$. 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 subset OPTIONALS *of* 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.

4 Diagnosing Ambiguity

We provide an overview of the architecture and the algorithms of Dr. Ambiguity in this section. In Section 5 we demonstrate its output on example parse forests for an ambiguous Java grammar.

4.1 Architecture

Figure 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.

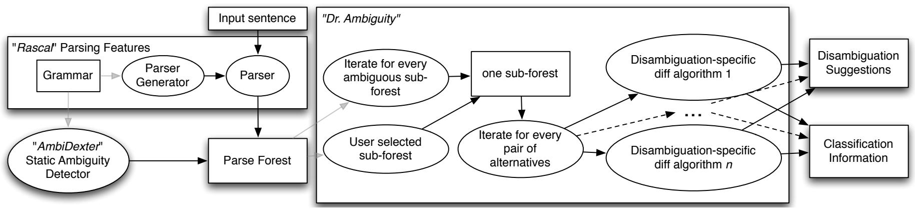

Input sentence
"Rascal" Parsing Features
Grammar
Parser Generator
Parser
"AmbiDexter" Static Ambiguity Detector
Parse Forest
"Dr. Ambiguity"
Iterate for every ambiguous sub-forest
User selected sub-forest
one sub-forest
Iterate for every pair of alternatives
Disambiguation-specific diff algorithm 1
...
Disambiguation-specific diff algorithm n
Disambiguation Suggestions
Classification Information

Figure 4. Contextual overview (input/output) of "Dr. Ambiguity".

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 2). However, since some methods may share intermediate analyses there is some additional intermediate stages and some data-dependency that is not depicted in Figure 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 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 [21]. 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 [35]):

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

[2] We use Eclipse IMP [13] as a platform for generating editors for programming languages defined using Rascal [21]. IMP provides contextual pop-up menus.

[3] The current implementation of Rascal lacks many trivial optimizations.

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];
}
```

The following functions each implement one of the diff algorithms from Figure 4. Intuitively they identify one of the spots from the lower Venn diagram in Figure 3. The following two (slightly simplified[4]) functions detect opportunities to apply priority or associativity disambiguations.

```
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 (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:

```
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 [];
```

[4] We have removed references to location information that facilitates IDE features.

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

The function `danglingCauses` detects re-orderings of arbitrary depth, after which the outermost productions are compared by `danglingFollowRestrictions` 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].

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 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 [11], 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

[5] The source code is available at `http://svn.rascal-mpl.org/rascal/trunk/src/org/rascalmpl/library/Ambiguity.rsc`.

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 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.

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 [11,10]. 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 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.

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 of precision and recall. 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".

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) fail; }`
9 associativity groups	`Expr = left ( 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 1. Disambiguations applied in the Java 5 grammar [11]

- 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 Dr. Ambiguity for the parse forests of each grammar version, and compute the precision and recall.

Recall is computed by $\frac{|\textsc{FoundDisambiguations} \cap \textsc{RemovedDisambiguations}|}{|\textsc{RemovedDisambiguations}|} \times 100\%$. 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.
Precision is computed by $\frac{|\textsc{FoundDisambiguations} \cap \textsc{RemovedDisambiguations}|}{|\textsc{FoundDisambiguations}|} \times 100\%$. 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.

5.2 Results

Table 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 parsing of a single Java program file that triggers the introduced ambiguity[6].

[6] **Note to reviewers:** We intend to add more experiments with this grammar for the camera-ready version of this paper.

	Diagnoses								
Experiment	**P**	**A**	**R**	**F**	**c**	**v**	**O**	**Precision**	**Recall**
1. Remove priority between "*" and "+"	**1**	1	0	0	0	1	0	33%	100%
2. Remove associativity for "+"	0	**1**	0	0	0	0	0	100%	100%
3. Remove reservation of `true` keyword from `ID`	0	0	**1**	0	0	1	0	50%	100%
4. Remove longest match for identifiers	0	0	0	**6**	0	0	0	16%	100%
5. Remove package name vs. field access priority	0	0	0	0	**6**	1	0	14%	100%
6. Remove vertical preference for dangling else	0	0	0	1	14	**1**	1	7%	100%
7. *All the above changes at the same time*	**1**	**2**	**1**	**7**	**20**	**4**	1	17%	100%

Table 2. Precision/Recall results for each experiment, including (P)riority, (A)ssociativity, (R)eject, (F)ollow restrictions, A(c)tions filtering edges, A(v)oid/prefer suggestions, and (O)ffside rule. For each experiment, the figures of the removed disambiguation are highlighted.

Table 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.

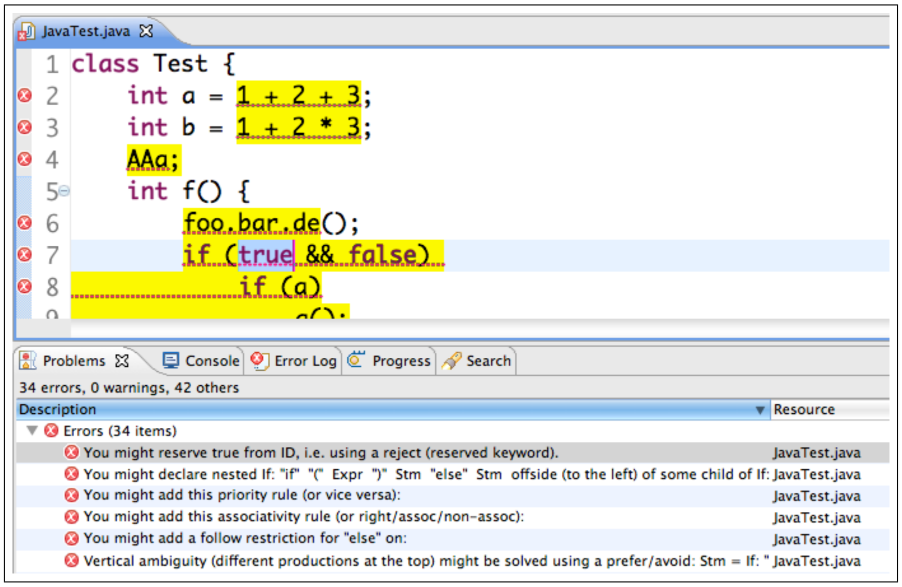
The final experiment tests whether the simultaneous analysis of different choice nodes that are present in a parse forest may lead to a loss of precision or recall. The results show that we find exactly the same suggestions. Also, as expected the precision of such an experiment is very low. Note however, 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 5 depicts how Dr Ambiguity may report its output.

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 ques-



```
class Test {
    int a = 1 + 2 + 3;
    int b = 1 + 2 * 3;
    AAa;
    int f() {
        foo.bar.de();
        if (true && false)
            if (a)
```

```
34 errors, 0 warnings, 42 others
Description                                                                                    Resource
Errors (34 items)
    You might reserve true from ID, i.e. using a reject (reserved keyword).                      JavaTest.java
    You might declare nested If: "if" "(" Expr ")" Stm "else" Stm offside (to the left) of some child of If: JavaTest.java
    You might add this priority rule (or vice versa):                                           JavaTest.java
    You might add this associativity rule (or right/assoc/non-assoc):                           JavaTest.java
    You might add a follow restriction for "else" on:                                           JavaTest.java
    Vertical ambiguity (different productions at the top) might be solved using a prefer/avoid: Stm = If: " JavaTest.java
```

Figure 5. Dr. Ambiguity reports diagnostics in the Rascal language workbench.

tion. 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.

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).

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers. Principles, Techniques and Tools. Addison-Wesley (1986)

2. Altman, T., Logothetis, G.: A note on ambiguity in context-free grammars. Inf. Process. Lett. 35(3), 111–114 (1990)
3. Aycock, J., Horspool, R.: Faster generalized LR parsing. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 32–46. Springer-Verlag (1999)
4. Basten, H.J.S.: Tracking down the origins of ambiguity in context-free grammars. In: Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC 2010). LNCS, vol. 6255, pp. 76–90. Springer (2010)
5. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Brabrand, C., Moreau, P.E. (eds.) Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010). pp. 5:1–5:9. ACM (2010)
6. Begel, A., Graham, S.L.: XGLR–an algorithm for ambiguity in programming languages. Science of Computer Programming 61(3), 211 – 227 (2006), Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004)
7. Bouwers, E., Bravenboer, M., Visser, E.: Grammar engineering support for precedence rule recovery and compatibility checking. ENTCS 203(2), 85 – 101 (2008), proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)
8. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Sci. Comput. Program. 75(3), 176–191 (2010)
9. van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Horspool, R.N. (ed.) Compiler Construction, 11th International Conference, CC 2002. LNCS, vol. 2304, pp. 143–158. Springer (2002)
10. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ. SIGPLAN Not. 41, 209–228 (October 2006)
11. Bravenboer, M., Vermaas, R., de Groot, R., Dolstra, E.: Java-front: Java syntax definition, parser, and pretty-printer. Tech. rep., `http://www.program-transformation.org` (2011), `http://www.program-transformation.org/Stratego/JavaFront`
12. Bravenboer, M., Vermaas, R., Vinju, J.J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Glück, R., Lowry, M.R. (eds.) Generative Programming and Component Engineering, 4th International Conference, GPCE 2005. LNCS, vol. 3676, pp. 157–172. Springer, Tallinn, Estonia (2005)
13. Charles, P., Fuhrer, R.M., Jr., S.M.S., Duesterwald, E., Vinju, J.: Accelerating the creation of customized, language-specific IDEs in eclipse. In: Arora, S., Leavens, G.T. (eds.) Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009 (2009)
14. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Braffort, P. (ed.) Computer Programming and Formal Systems, pp. 118–161. North-Holland, Amsterdam (1963)
15. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13, 94–102 (February 1970)
16. Economopoulos, G.R.: Generalised LR parsing algorithms. Ph.D. thesis, Royal Holloway, University of London (August 2006)
17. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. SIGPLAN Not. 39, 111–122 (January 2004)
18. Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. Journal of Computer and System Sciences 1(1), 1–23 (1967)

19. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices 24(11), 43–75 (1989)
20. Johnstone, A., Scott, E.: Modelling GLL parser implementations. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Software Language Engineering, LNCS, vol. 6563, pp. 42–61. Springer Berlin / Heidelberg (2011)
21. Klint, P., van der Storm, T., Vinju, J.: EASY meta-programming with Rascal. In: Fernandes, J.a., Lämmel, R., Visser, J., Saraiva, J.a. (eds.) Generative and Transformational Techniques in Software Engineering III, LNCS, vol. 6491, pp. 222–289. Springer Berlin / Heidelberg (2011)
22. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In: Pighizzini, G., San Pietro, P. (eds.) Proc. ASMICS Workshop on Parsing Theory. pp. 1–20. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy (1994)
23. Lämmel, R., Verhoef, C.: Semi-automatic grammar recovery. Softw. Pract. Exper. 31, 1395–1448 (December 2001)
24. Landin, P.J.: The next 700 programming languages. Commun. ACM 9, 157–166 (March 1966)
25. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001). pp. 13–. WCRE 2001, IEEE Computer Society, Washington, DC, USA (2001)
26. 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. pp. 425–436. PLDI 2011, ACM, New York, NY, USA (2011)
27. Rekers, J.: Parser Generation for Interactive Environments. Ph.D. thesis, University of Amsterdam (1992)
28. 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. pp. 170–178. PLDI 1989, ACM (1989)
29. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), see `http://accent.compilertools.net/Amber.html`
30. Schröer, F.W.: ACCENT, a compiler compiler for the entire class of context-free grammars, second edition. Tech. rep., compilertools.net (2006), see `http://accent.compilertools.net/Accent.html`
31. Scott, E.: SPPF-style parsing from earley recognisers. ENTCS 203, 53–67 (April 2008)
32. Scott, E., Johnstone, A.: GLL parsing. ENTCS 253(7), 177 – 189 (2010), proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)
33. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1985)
34. Vinju, J.J.: SDF disambiguation medkit for programming languages. Tech. Rep. SEN-1107, Centrum Wiskunde & Informatica (2011), `http://oai.cwi.nl/oai/asset/18080/18080D.pdf`
35. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, Universiteit van Amsterdam (1997)