A Non-deterministic Approach to Ambiguity Detection in Context Free Grammars

Naveneetha K. Vasudevan
Department of Informatics
King's College London

Thesis submitted in partial fulfilment for the degree of
Doctor of Philosophy

May 2017

Abstract

Context Free Grammars (CFGs) are widely used for describing programming languages. CFGs are often ambiguous, allowing inputs to be parsed in more than one way. While ambiguity has applications in many fields, it is problematic for programming languages—if an input presents a choice of parses, which one should the compiler use? It is thus desirable to identify and remove ambiguity in programming language grammars. However, since CFGs typically describe infinite languages, ambiguity is, in general, undecidable.

Previous approaches to detecting ambiguity have relied on searching a grammar 'in depth' (using small search strings) from its root. In this thesis, I hypothesise that searching a grammar 'in breadth' (using large search strings) is more likely to uncover ambiguity. I introduce the novel concept of search-based ambiguity detection, which is a non-deterministic breadth-based approach to CFG ambiguity detection. Starting from the simplest possible search-based ambiguity algorithm, I show how a high quality algorithm can be constructed using this approach.

In order to evaluate my approach, I also introduce two new techniques for generating large corpuses of random grammars. I use Boltzmann sampling to produce fully random grammars with some statistical guarantees of coverage, and grammar mutation for generating random programming language grammars from an existing set of unambiguous grammars. My grammar corpuses allow me to carry out the largest cross ambiguity detection tool experiment to date. These experiments show that search-based ambiguity detection performs as well as, and generally better, than extant tools.

Finally, I present a novel search-based approach to grammar ambiguity minimisation, showing that it is effective at minimising both the ambiguous input and the portion of the grammar identified as containing the ambiguity.

Acknowledgements

My sincere gratitude goes to Laurence Tratt. His guidance, patience and support throughout the course of my research work has been invaluable. I would also like to thank my fellow team members in the Software Development Team at King's College for their support, guidance and of course friendship.

I am grateful to the Department of Informatics at King's College for giving me extended access to computing facilities. I am much indebted to my employer ThoughtWorks for being generous in giving me the time off to complete my PhD.

I would like to thank Bas Basten, Alexis Darrasse, and Friedrich Schröer for their help during the course of the research. I am much indebted to Edd Barrett for his insightful comments on drafts of this thesis. I am also thankful to the reviewers for their valuable comments and suggestions, which helped me to improve this thesis.

Finally, I am grateful to my family. This thesis would not have been possible without their understanding and support.

Contents

Chapter 1

Introduction

1.1 Parsing

Context Free Grammars (CFGs) are widely used for describing formal languages, including Programming Languages (PLs). The full class of CFGs includes ambiguous grammars—those which can parse inputs in more than one way. Needless to say, ambiguous grammars are highly undesirable. If an input can be parsed in more than one way, which one of those parses should be taken? We would not enjoy using a compiler if it were to continually ask us to choose which parse we want. Unfortunately, we know that, in general, it is undecidable as to whether a given grammar is ambiguous or not [12]. While there are various parsing approaches which allow a user to manually disambiguate amongst multiple parses, one can not in general know if all possible points of ambiguity have been covered. Perhaps because of this, most tools use parsing algorithms such as LL and LR, which limit themselves to parsing a subset of the unambiguous grammars. This leads to other trade-offs: grammars have to be contorted to fit within these subsets; and these subsets rule out the ability to compose grammars [16].

Generalised parsers such as Earley [17], GLL [37], GLR [38] are being used more often as they are able to handle much larger classes of CFGs. Generalised parsers allow a grammar developer to design the grammar in a way that best describes the structure of the intended language. The grammar doesn't have to be contorted to comply with YACC, for instance. The downside in using the entire class of CFGs is the possibility of ambiguity. Therefore, when using the entire class of CFGs to describe programming languages, it becomes necessary to have an ambiguity detection tool.

1.2 Goal and Motivation

Ambiguity detection tools detect ambiguities in a grammar by exploring its search space systematically. Since ambiguity detection is statically undecidable, ambiguity detection

approaches are generally unable to guarantee that a given grammar is unambiguous in finite time. This then leads to different design decisions being taken in order to explore a grammars' search space.

The extant ambiguity detection approaches are deterministic and explore a grammar in 'depth'. A subset of the grammar space is searched in exhaustive detail by generating large numbers of short strings and then checking for ambiguity. My work is based on the idea that exploring a grammar in 'breadth' has a better chance of uncovering ambiguity. I formalise this notion in the following hypothesis:

H1 Covering a grammar in breadth is more likely to uncover ambiguity than covering it in depth.

This hypothesis captures the notion that covering as much of a grammar as possible is more likely to uncover ambiguity. A 'naive' notion of grammar coverage requires that grammar rules are visited at least once [33]; while easily achieved, this makes no guarantees that all interesting combinations of rules are explored. A more 'powerful' notion therefore involves combinations of rules or productions [27]. Although complete coverage under this latter notion may be definable as a finite set, there is no guarantee that such a set is of a tractable size. My aim is therefore to find an approach which falls somewhere between the naive and the powerful notions of grammar coverage.

Understanding the relation between CFGs and their various subsets is key to understanding the motivation for, and the results of, my work. Figure 1.1 shows the relation between the various grammar subsets. Since all the sets involved are infinite, this diagram is necessarily an approximation. An important question to my work is how much of the expressivity of the unambiguous (LL or LR) grammars do PL grammars use? I therefore make the following hypothesis:

H2 PL grammars are only a small step away from being ambiguous.

This hypothesis captures the notion that much of the expressivity of the PL grammars overlaps with that of the unambiguous (i.e. LL or LR) grammars. When defining a grammar for a PL, a PL author typically starts with a grammar that best describes the intended language. It is only when the grammar is fed to YACC (for parser generation), one discovers that the grammar is potentially ambiguous (i.e. contains shift/reduce or reduce/reduce conflicts). The grammar is then gradually adjusted, resolving each of the conflict, to make it unambiguous. My careful observation of several of the PL grammars from grammar archives [2] revealed that most PL grammars reside within the unambiguous subset. My underlying hypothesis is that PL grammars often stretch to the very edge of the class of unambiguous grammars. Stated differently, I suspect that unambiguous PL grammars are only a small step away from being ambiguous.

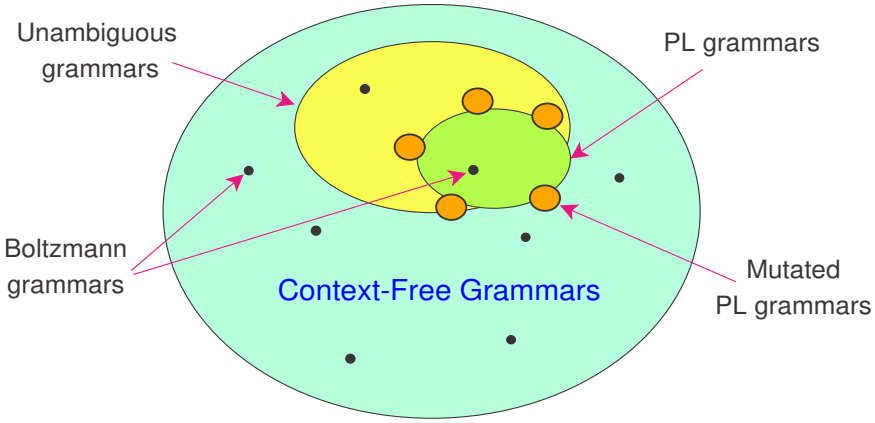

Figure 1.1: A finite approximation of the class of CFGs and the various subsets of grammars contained within. Set of unambiguous grammars are a strict subset of CFGs. Set of PL grammars are a subset of Unambiguous grammars. Mutated PL grammars are on the very edge of Unambiguous grammars. Boltzmann grammars are distributed randomly within the class of CFGs.

1.3 The *SinBAD* Solution

In the light of the above hypotheses, this thesis presents *SinBAD*, a breadth-based ambiguity detection tool. *SinBAD* houses several heuristics, ranging from purely random to semi-non-deterministic, for uncovering ambiguities in grammars. Each of *SinBAD*'s heuristics explore a grammars' search space in breadth, visiting grammar rules semi-randomly, to uncover ambiguous strings. Table 1.1 presents a comparison of the extant ambiguity detection tools against *SinBAD* using Basten's set of characteristics [4].

To evaluate *SinBAD* I generated a large grammar corpus using grammar mutation techniques. My mutation technique involves applying a single change to an unambiguous grammar to generate a random 'PL like' grammar. I have devised five different type of mutations, with which I generated a large corpus of possibly ambiguous PL-like grammars.

SinBAD validates Hypothesis H1: not only does *SinBAD* uncover 16% more ambiguities than existing deterministic approaches, but it uncovers them more quickly.

SinBAD also validates Hypothesis H2: making a single change to an unambiguous PL grammar results in 37% of the grammars becoming ambiguous.

1.4 Overall Thesis Structure

To test my hypotheses, this thesis is organised into five main parts:

1. An analysis and review of extant ambiguity detection approaches.

Tools	ACC_{amb}	ACC_{unamb}	Exhaustive
YACC [24]	N	Y	N
ACLA [10]	Y	Y	N
AMBER [36]	Y	N	Y
NU Test [34]	Y	Y	N
AmbiDexter [7]	Y	Y	Y
SinBAD	Y	N	N

Table 1.1: A comparison of the extant ambiguity detection tools with *SinBAD*. An approach is considered accurate if it can definitely prove a grammar to be ambiguous (denoted as ACC_{amb}) or unambiguous (ACC_{unamb}). An 'Exhaustive' approach explores the search space of a grammar in depth, and therefore, by definition, is non-terminating. *SinBAD*'s approach is subtly different to other approaches in that it explores a previously untried section in the design space.

2. I present a new ambiguity detection tool, *SinBAD* that houses various non-deterministic search-based heuristics for detecting ambiguities in CFGs.
3. I then present two novel ways of generating random grammars.
4. *SinBAD* is evaluated along with three other ambiguity detection tools in the largest cross ambiguity detection experiment.
5. A novel grammar minimisation approach by ambiguity detection is presented.

1.5 Contributions

The main contributions of this thesis are as follows:

- The novel concept of non-deterministic ambiguity detection. I incrementally develop a non-deterministic algorithm to detect ambiguity, showing how different design choices affect results. The implementation of these algorithms in a tool, allowing a large scale experiment to be carried out.
- Two novel ways of generating large grammar corpuses. The use of Boltzmann sampling to generate fully random grammars (with some statistical guarantees of coverage). The use of grammar mutation to produce distinct grammars from an existing set of unambiguous grammars.
- The automated experimental suite containing various grammar corpuses and the setup for carrying out large scale evaluation.

- The novel concept of search-based grammar ambiguity minimisation.

1.6 Detailed Synopsis

Chapter 2 introduces the general concepts of parsing and parsing techniques. An introduction to ambiguity in CFGs is provided. Ambiguity is explained using examples. Several extant ambiguity detection tools are reviewed and analysed.

Chapter 3 introduces a search-based approach to ambiguity detection for CFGs. The motivation for implementing a search-based approach is presented. I then present the *SinBAD* tool. A series of gradually better non-deterministic heuristics for ambiguity detection in CFGs is presented.

Chapter 4 presents two novel ways of generating random grammars. An introduction to Boltzmann sampling is provided. The first Boltzmann sampler for generating CFGs is presented. A mutation based PL grammar generator is presented. Five different ways of mutating a PL grammar are presented.

Chapter 5 presents the dimensioning of the ambiguity detection tools from my experimental suite. The chapter comes in four parts. The first part covers the experiment methodology. An overview of the tools from my experimental suite is provided, and grammar generation is covered. The second part presents a new hill climbing based implementation for uncovering a 'good enough' solution for a given ambiguity detection tool.

Based on my hill climbing implementation, two dimensioning experiments are performed. The third part covers the crude dimensioning experiment, wherein my tools are evaluated on a small grammar corpus to understand their solution landscape. The fourth and final part covers the fine dimensioning experiment, wherein for each tool, the region uncovered by crude dimensioning is explored in detail to determine good run-time options.

Chapter 6 presents the main experiment, which evaluates my tools using the best run-time options discovered by the fine dimensioning experiment on a much larger grammar corpus. Based on the results, the strengths and the weaknesses of each tool are discussed.

Chapter 7 presents a search-based approach to grammar ambiguity minimisation for CFGs. The chapter starts with an introduction to my grammar minimisation approach. I then implement my grammar minimisation approach in a tool. The grammar minimiser is then evaluated on a large grammar corpus. The chapter concludes with a discussion.

Chapter 8 present my conclusions and provides directions for possible future research.

1.7 List of Publications

Several parts of this thesis have appeared in previous publications. The following papers are reproduced in part or full:

1. Chapter 3 was published in the proceedings of the Second Workshop on Imperial College Computing Student Workshop (ICCSW 2012) [39]

2. Chapter 4 was published in the proceedings of the Sixth Conference on the Software Language Engineering (SLE 2013) [40]

Chapter 2

An Overview of Ambiguity Detection

This chapter is intended to provide the background material necessary for the work presented in this thesis. This chapter starts with an overview of the different types of grammars, and the respective languages they describe. The basic concepts that underpin parsing in programming languages are explained. An introduction to ambiguity is given, followed by an overview of the extant ambiguity detection approaches to CFGs. The concepts involving grammars and parsing are provided here for completeness. Readers who are familiar with these concepts may wish to jump straight to the section on ambiguity 2.3.

2.1 Grammars and Languages

A *grammar* is a set of rules that describes the syntax of a language. Each *rule* defines a set of sequence of symbols that it can generate. The rules of a grammar are applied recursively to generate syntactically valid sentences. A *sentence* is defined as a sequence of *words*, and a word is a symbol from the alphabet of the language. The set of valid sentences that can be generated from a grammar is defined as the language of the grammar. An example grammar is shown below.

$$A \rightarrow aB$$
$$B \rightarrow b$$

The symbols contained in the grammar are of two kinds: terminals such as a and b that make up the alphabet of the grammar; and non-terminals such as A and B that act as variables that cannot occur in a sentence. To distinguish non-terminals from terminals, I follow a simple convention: non-terminals are written using the uppercase letters, whereas terminals are written using the lowercase letters. A rule consists of a symbol on the left-hand side (left of $\rightarrow$) and a sequence of symbols on the right-hand side (right of $\rightarrow$). The $\rightarrow$ indicates that the symbol (i.e. non-terminal) on the left-hand side "may be replaced by" the sequence of symbols (terminal or non-terminal) on the right-hand side of a rule. One

of the non-terminals of the grammar is designated as the start symbol of the grammar. If A is designated as the start symbol, then the language (i.e. set of accepted sentences) described by the example grammar is $\{ab\}$.

2.1.1 Formal Definition

Formally, a CFG is defined as a tuple $G = \langle N,T,P,S \rangle$ where N is the set of non-terminals, T is the set of terminals, P is the set of production rules (just rules from now on), and S is the start non-terminal of the grammar. A production rule is denoted as $A : \alpha$, where $A \in N$, and α is a sequence of symbols drawn from $(N \cup T)^*$. A rule with an empty right-hand side is denoted as $A : \epsilon$, where ϵ denotes an empty string. The upper case letters A, B, C and so on will be used to refer to non-terminals and lower case letters a, b, c and so on will be used to refer to terminals. The upper case letters U, V, W and so on will either refer to a terminal or a non-terminal, and greek letters α, β, γ and so on will be used to refer to a sequence of terminals or non-terminals. For a grammar G, the language it describes is denoted as $L(G)$.

2.1.2 Chomsky Hierarchy of Grammars and Languages

Grammars come with varying power of expressivity. In the late 1950's Chomsky [15] formalised the notion of a grammar as a generative device for describing a language, and classified grammars into different classes based on their expressive power. The languages described by the class of grammars with higher expressive powers are a superset of the languages described by those with lower expressive powers. I now describe the various classes of grammars, starting from the least to the most expressive.

The least expressive class of grammars are the *regular grammars*. In a regular grammar, the right-hand side of a rule can either contain one terminal or a terminal followed by a non-terminal. The example grammar in Section 2.1 is a regular grammar.

The class of Context Free Grammars (CFGs) expand on the class of regular grammars. CFGs remove regular grammars' restrictions: the right-hand side of a rule in a CFG can have an arbitrary number of symbols, and the terminals and the non-terminals can occur appear in any order. As a result, CFGs have higher expressive power than regular grammars, and thus describe a larger set of languages.

The set of Context-Sensitive Grammars (CSGs) expand on the set of CFGs. Whereas in the case of CFGs, the left-hand side of a rule always contained a non-terminal, in the case of CSGs, the non-terminal on the left-hand side can be surrounded by an arbitrary number of symbols (terminal or non-terminal). These symbols form the context under which the non-terminal gets replaced by a sequence of symbols on the right-hand side.

A CSG rule replaces only one non-terminal from its left-hand side by its right-hand side. An example rule in a CSG might look as follows:

$$aBC \rightarrow abC$$

That is, the non-terminal B is surrounded by context a and C, and will be replaced by terminal b only in the context of a and C. The ability to apply rules based on the surrounding context on either side of a rule makes them useful for describing natural languages.

The class of Unrestricted grammars expand on the class of CSGs. As the name suggests, the rules in an unrestricted grammar allow any arbitrary (non-zero) sequence of symbols to be replaced by any arbitrary (possibly zero) sequence of symbols. Whereas a CSG rule allows exactly one non-terminal from its left-hand side to be replaced, an Unrestricted grammar rule allows any number of terminals and non-terminals to be replaced. An example rule in an Unrestricted grammar might look as follows:

$$ABC \rightarrow aEF$$

That is, whenever the sequence of symbols ABC is found, replace it with sequence of symbol aEF. Languages described by the set of Unrestricted grammars are the most powerful in the hierarchy; the complicated structure of these grammars also means that their use is limited mostly to theoretical purposes.

Although the class of CSGs and Unrestricted grammars are quite powerful, their use of context for applying rules makes them them quite complex, and thus less suitable for describing programming languages [19].

2.2 CFGs

CFGs are widely used for describing programming languages. The rules of a CFG are typically written in a more condensed form called the Backus-Naur Form or BNF [3]. In this thesis, I use the YACC (BNF variant) notation for writing CFGs. In a BNF notation, a rule takes the following form:

$$\textit{<symbol>} : \ \textit{<expression>}$$

where *<symbol>* is a non-terminal, and *<expression>* consists of one or more sequences of symbols. The sequences are separated by a vertical bar '|' to indicate a choice. Each choice on the right-hand side of a rule is called an alternative. The special symbol ':' acts as a delimiter between the label of the rule on the left and the body of the rule on the right. The ':' symbol indicates that the non-terminal on the left may be replaced by the expression on the right.

2.2.1 Generating Sentences from a Grammar

Generating a sentence from a grammar involves a sequential application of rules, starting with the start rule of the grammar. The alternative of the start rule forms the initial string s. A rule is applied to replace each non-terminal in the string s by its alternatives. Such a symbol replacement is formally called a *derivation step* and is denoted by the $\Rightarrow$ symbol. The rules are applied to the string s until there are no more non-terminals to be replaced, and a *sentence* has been generated. I now explain how a sentence is generated for the following example grammar.

$$
\begin{aligned}
&root: \ expr; \\
&expr: \ expr \ + \ expr \mid expr \ * \ expr \mid num; \\
&num: \ 1 \mid 2 \mid 3;
\end{aligned}
$$

For the example grammar, the sequence of steps involved in generating the sentence *1 + 2* is as follows. The sentence generation commences from the start rule *root* of the grammar. The start string contains the non-terminal *expr*. The symbol *expr* is replaced by the string *expr + expr* by applying the first alternative of the *expr* rule. The derivation step involving the rewrite of the string *expr* as *expr + expr* is shown as:

$$expr \ \Rightarrow \ expr \ + \ expr$$

The string *expr + expr* is rewritten as *num + expr* by replacing the first non-terminal *expr* by applying the third alternative of the *expr* rule. The string *num + expr* is rewritten as *1 + expr* by replacing the non-terminal *num* with the first alternative of the *num* rule. In the subsequent steps, the remaining non-terminal *expr* is replaced in a similar way, yielding the final string *1 + 2*. The process of rewriting of a string, where a non-terminal from the string is replaced by a sequence of symbols, continues until the string contains no non-terminals. Such a string containing just terminals is called a sentence. The intermediate string containing both non-terminals and terminals is called a *sentential form*. The sequence of derivation steps in generating a sentence is called a *derivation*. The derivation for sentence *1 + 2* is as follows:

$$
\begin{aligned}
root \ &\Rightarrow \ expr \\
root \ &\Rightarrow \ expr + expr \\
root \ &\Rightarrow \ num + expr \\
root \ &\Rightarrow \ 1 + expr \\
root \ &\Rightarrow \ 1 + num \\
root \ &\Rightarrow \ 1 + 2
\end{aligned}
$$

The above derivation is usually written in a condensed form:

$$root \ \Rightarrow \ expr \ \Rightarrow \ expr + expr \ \Rightarrow \ num + expr \ \Rightarrow \ 1 + expr \ \Rightarrow \ 1 + num \ \Rightarrow \ 1 + 2$$

The set of sentences derived from a non-terminal A is called the language of A, $L(A)$. The set of sentences derived from the start symbol of a grammar is the called the language of the grammar.

When replacing a non-terminal in a derivation step, there is often a choice: either the leftmost or the rightmost non-terminal can be replaced. The former results in a leftmost derivation (indicated as $\Rightarrow_{\text{lm}}$), whereas the latter results in a rightmost derivation (indicated as $\Rightarrow_{\text{rm}}$). Usually one of the approaches is chosen. The derivation shown above is a leftmost derivation.

2.2.2 Recursion

Grammars often contain rules that reference themselves. A rule that references itself is termed recursive. A grammar is recursive if it contains a recursive rule. An example rule that is recursive is *A: Aa*. A rule *A:* $\alpha A \beta$ is left-recursive, if α can be derived to an empty string in zero or more derivation steps; and right-recursive, if β can be derived to an empty string in zero or more derivation steps.

2.2.3 Parse Tree

The derivation of a sentence can also be represented in the form of a parse tree. A parse tree captures the syntactic structure of a sentence with respect to the given grammar. A parse tree is usually represented as a downward facing tree with the start symbol at the root of the tree. The interior nodes of the tree are made up of non-terminals, and the leaf nodes are made up of terminals. Each derivation step in a sentence generation corresponds to a node in the tree. For a derivation step where a non-terminal A is replaced by a sequence of symbols $A_0 \ldots A_n$, there exists a corresponding interior node in the tree labelled A with $A_0 \ldots A_n$ as its child nodes. Every valid parse tree with respect to a grammar represents a sentence. The parse tree for the sentence `1 + 2` derived using the example grammar in Section 2.2.1 is shown in Figure 2.1.

2.2.4 Parsing Grammars

The process of constructing a parse tree for a sentence using the grammar rules is known as parsing. Tools that execute this process are called parsers. The purpose of a parser is to determine if the sentence can be derived with respect to the grammar whilst also constructing a grammatical structure of the sentence. There are two basic approaches for constructing a parse tree: top-down and bottom-up. This section presents an overview of the top-down and the bottom-up parsing approaches.

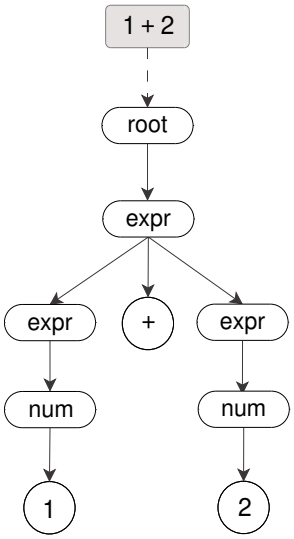

Figure 2.1: The parse tree for the sentence `1 + 2` derived from the example grammar in Section 2.2.1

.

2.2.4.1 Top-down Parsers

A top-down parser attempts to derive a sentence by performing a sequence of derivation steps from the start rule of the grammar. A top-down parser gets its name from the way it constructs the parse tree: nodes are constructed from the top, and as the derivation progresses, child nodes are added to the tree. Examples of top-down parsers include LL and GLL [19].

A top-down parser works as follows. For a given input, the top-down parser begins with the start rule. A root node is created for the start symbol. A non-terminal is derived by predicting one of its alternatives. The symbols of the predicted alternative then form the child nodes of the derived non-terminal. To predict an alternative, the parser performs a *look-ahead*: the next token in the input string is matched with the first symbol of the alternative. If the first symbol of the alternative is a terminal and if the terminal matches the input symbol, then the input symbol is read and the parser moves to the next symbol of the rule. If the first symbol is a non-terminal, then the parser *predicts* the right-hand side of the rule that the non-terminal defines. This way the parser continues to build the parse tree until all of the input has been successfully parsed. In cases where all of the alternatives of a rule begins with a non-terminal, the parser is forced to pick one of the alternatives. If the alternative picked results in a failed parse, then the parser backtracks to the point where the choice was made and picks a different alternative.

Top-down parsers are relatively straightforward to implement and the structure of the parser closely reflects the structure of the grammar. Parsers can also be generated from the grammar specification using a parser generator. ANTLR [32] is one such tool that generates a top-down LL parser. Since top-down parsers perform leftmost derivation, they

cannot handle left-recursive grammars. Consider the following left-recursive grammar: `S: SA | ϵ; A: a`. Given an input string '`aaa`...' of length n, an LL(k) parser (with k being the number of look-ahead symbols) can't make a parsing decision when $n \geq k$. After inspecting k input tokens, it is still not clear whether we are in the middle of a sentence (so keep applying the rule `S: SA`) or at the end of the sentence (so apply rule `S: ϵ`).

2.2.4.2 Bottom-up Parsers

A bottom-up parser attempts to construct a derivation in reverse, effectively deriving the start symbol from the sentence that is being parsed. As the name suggests, in a bottom-up parser, the leaf nodes at the bottom of the tree are constructed first, followed by the interior nodes, and then leading up to the root of the tree. Examples of a bottom-up parsers include LR(k) and GLR [19].

A bottom-up parser works as follows. The parser keeps reading (*shifting*) the input symbols until it finds a sequence of symbols that matches the right-side of a rule. The portion of the sentential form that matches the right-hand side of a rule is called a *handle*. Once the handle is found, the string that matches the handle in the sentential form is replaced (*reduced*) by the non-terminal on the left hand side of the rule. For each terminal shifted, a leaf node in the parse tree is constructed. When a reduction is performed, a new intermediate node is created, and the nodes labelled by the symbols are attached to it as children. This process of shifting and reduction continues until the sentential form contains just the start symbol and a node labelled as start symbol is created, at which point the input has been successfully parsed.

A bottom-up parser uses a stack to determine a handle when parsing a sentence. The symbols from the input string are read and pushed onto the stack. When the symbols on top of the stack match the right-hand side of a rule, a reduction is performed. The symbols matching the right hand side of a rule are popped and the corresponding non-terminal is placed on top of the stack. When the top of the stack contains the start symbol, the input string has been successfully parsed. I now explain bottom-up parsing using an example.

2.2.4.2.1 Bottom-up Parsing – an Example

Consider the following example grammar. The rules of the grammar are annotated by a unique number.

(1) $S: \ aABe$

(2) $A: \ Abc$

(3) $A: \ b$

(4) $B: \ d$

Stack S	Input I	Action T
	abbcde	shift
a	bbcde	shift
ab	bcde	reduce 3
aA	bcde	shift
aAb	cde	shift
aAbc	de	reduce 2
aA	de	shift
aAd	e	reduce 4
aAB	e	shift
aABe		reduce 1
S		accept

Table 2.1: Bottom-up parsing: the shifts and reductions involved in parsing the sentence *abbcde* using the example grammar (see Section 2.2.4.2.1).

The sequence of shifts and reductions for sentence *abbcde* with respect to the above example grammar is shown in the Table 2.1. The stack S contains the shifted symbols, the input buffer I contains the input string and T denotes the action to be performed.

We start with an empty stack S. The token a is read from the input string (I) and placed on top of the stack S. Since there is no handle, the next token b from I is shifted on to S. We now have a handle, since b forms the right hand side of rule 3. A reduce action is performed, and the non-terminal A is placed on top of the stack S. This way, we continue to shift tokens from input I, and on finding a handle, reduce. When the top of the stack contains the start symbol S, we have successfully parsed the input string (denoted as *accept* action).

Bottom-up parsers are difficult to implement by hand. Fortunately there exists parser generators such as Bison [1] that can generate a bottom-up parser based on an input grammar. The class of grammars parseable by a bottom-up parser is a strict superset of the class of grammars parseable by a top-down parser. Further, bottom-up parsers handle left recursion.

2.3 Ambiguity

Whilst every valid parse tree corresponds to a single sentence, the reverse is not guaranteed: a sentence can easily have more than one parse tree. If a sentence can be parsed in more than one way with respect to a grammar, then the sentence is ambiguous. A grammar is ambiguous if there exists a sentence derivable from the grammar that is ambiguous. The meaning of a sentence is derived from the structure of its parse tree. A

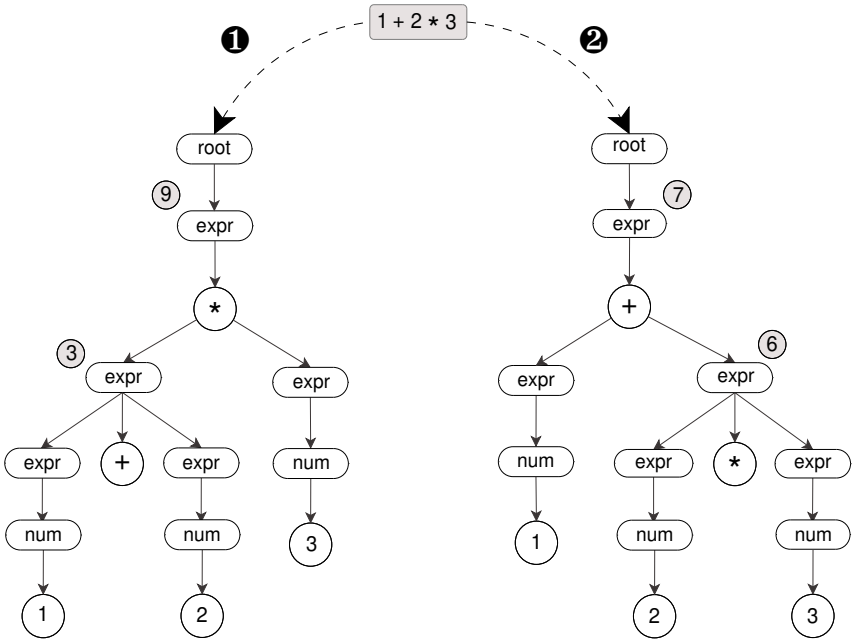

Figure 2.2: Ambiguous parse trees for sentence: `1 + 2 * 3`. ❶ The numbers 1 and 2 are first added, and the result is then multiplied by 3, yielding a value of 9. ❷ The numbers 2 and 3 are first multiplied, and the result is then added to 1, yielding a value of 7. The number in the grey circle adjacent to a node indicate the value of the parse tree at that specific point.

sentence that can be parsed in more than one way implies that there is possibly more than one meaning that can be associated with it. I now show how ambiguity can manifest even in a seemingly harmless looking grammar.

For my example, I use the grammar shown in Section 2.2.1. The grammar is shown below for convenience.

$$
\begin{aligned}
&\mathit{root} : \ \mathit{expr};\\
&\mathit{expr} : \ \mathit{expr} \ + \ \mathit{expr} \mid \mathit{expr} \ * \ \mathit{expr} \mid \mathit{num};\\
&\mathit{num} : \ 1 \mid 2 \mid 3;
\end{aligned}
$$

The example grammar is quite simple, however, the rules it contains are representative of a typical PL grammar (most PL grammars include rules related to arithmetic expressions). A sentence that is ambiguous with respect to the example grammar is `1 + 2 * 3`. The example sentence can be parsed in two ways (see Figure 2.2). The syntactic structure of the parse trees and their meanings are different: whereas the parse tree on the left evaluates to 9, the parse tree on the right evaluates to 7.

From a programming language perspective, if a program can be associated with more than one meaning, then that poses several implications for a compiler. A compiler may choose to interpret a program differently to what the author of the program actually intended. For instance, where there is a choice of parses, a compiler might always choose the first parse (not the parse that the author intended); or in other cases, a compiler might fail on

a seemingly valid program, as it is expecting just one parse but got more than one. It is therefore essential that all the sources of ambiguity are uncovered and resolved prior to using a grammar. A realistic example involving ambiguity in a PL grammar now follows.

2.3.1 Ambiguity in a PL Grammar

To illustrate ambiguity in a PL grammar, I use a simplified version of the SQL grammar from my experimental corpus (see Listing 2.1). The grammar contains a subset of the expression rules from the SQL grammar, and such subsets are typical in PL grammars. Symbols +, (,), ∗ and the symbols in uppercase letters are terminals. Symbols in lowercase letters are non-terminals. The non-terminal `sql` is the start symbol.

```
1 %tokens SELECT, ID, FROM, TABLE, WHERE, STRING;
2
3 sql: select_stmt;
4 select_stmt: SELECT ID FROM TABLE WHERE expr;
5 expr: '(' expr ')' | expr '+' product | product;
6 product: product '*' term | term;
7 term: '(' expr ')' | STRING;
```

Listing 2.1: An Ambiguous SQL grammar

A sentence that is ambiguous with respect to my example SQL grammar is `select id from table where '(' string ')'`. The two parses for the ambiguous sentence is shown in Listing 2.2.

Although the example grammar is fairly small with only a handful of rules and alternatives, the source of the ambiguity may not be immediately obvious. The ambiguity is contained in the substring `'(' string ')'`, and originates within the first and the third alternative of the '`expr`' rule. The ambiguous string can be derived in two ways:

$$expr \Rightarrow (\ expr\ ) \Rightarrow (\ product\ ) \Rightarrow (\ term\ ) \Rightarrow (\ STRING\ )$$
$$expr \Rightarrow product \Rightarrow term \Rightarrow (\ expr\ ) \Rightarrow (\ product\ ) \Rightarrow (\ term\ ) \Rightarrow (\ STRING\ )$$

In the above fragment, the first line corresponds to the parse tree '`TREE 1`' (from Listing 2.2) and the second line corresponds to the parse tree '`TREE 2`'.

The ambiguous string is short, however, the two different parses that make up the ambiguity look far from trivial, the parses transcend multiple rules and subsets of the parses are nested. Ambiguity in PL grammars are characterised by such nested subsets. The parse trees contain the necessary information for the grammar author to diagnose and resolve the ambiguity.

Techniques exist that allow a user to manually disambiguate between multiple parses using operators such as associativity and precedences [19]. In [8], Dr. Ambiguity, a

```
1  Two different ‘‘expr’’ derivation trees for the same phrase.
2
3  TREE 1
4  ------
5
6  expr alternative at line 5, col 7 of grammar {
7    ’(’
8    expr alternative at line 5, col 47 of grammar {
9      product alternative at line 6, col 32 of grammar {
10       term alternative at line 7, col 27 of grammar {
11         STRING
12       }
13     }
14   }
15   ’)’
16 }
17
18 TREE 2
19 ------
20
21 expr alternative at line 5, col 47 of grammar {
22   product alternative at line 6, col 32 of grammar {
23     term alternative at line 7, col 7 of grammar {
24       ’(’
25       expr alternative at line 5, col 47 of grammar {
26         product alternative at line 6, col 32 of grammar {
27           term alternative at line 7, col 27 of grammar {
28             STRING
29           }
30         }
31       }
32       ’)’
33     }
34   }
35 }
```

Listing 2.2: Example of an ambiguity in a PL grammar

parse forest[1] diagnostics tool, was introduced that identifies the cause of an ambiguity and by inspecting the different parse trees that contribute to the ambiguity, proposes a disambiguation technique. Although such techniques manually disambiguate between multiple parses, one can not in general know if all possible points of ambiguity have been covered. Therefore, it is desirable to have an automated tool that would give us some indication as to whether a grammar is ambiguous or otherwise.

[1]A parse forest is the set of parse trees of an ambiguous sentence.

2.4 Ambiguity Detection

Over the last decade, there has been a steady stream of work trying to detect ambiguity in arbitrary grammars, in order to bring most of the benefits of the full class of CFGs without the disadvantages. However, it has been proven that detecting ambiguity in a CFG is, in general, undecidable [12]. Since most CFGs describe infinite languages, ambiguity detection approaches are therefore generally unable to guarantee that a given grammar is unambiguous in finite time. Inevitably, this leads to design choices when devising an ambiguity detection approach. Whilst certain ambiguity detection approaches are accurate, they may run forever and so would require a bound (time or string length) to be put in place. In other cases, the approaches may terminate but may report false positives. Invariably, such design choices influence the practical usability of an approach. A detailed comparative study of the various ambiguity detection approaches is presented in [4]. I now present an overview of the extant ambiguity detection approaches.

2.4.1 LR(k) and LRR

LR(k) parsing [26] is a bottom-up parsing technique that makes a parsing decision based on the next k input symbols. A parse table is central to LR(k) parsing: for each state and for k input symbols of look-ahead, an action is provided. An action may involve either shifting the input symbols or reducing with a grammar rule. For a given state, if there is more than one action for k input symbols of look-ahead, then there is a conflict. A conflict essentially means that there is no deterministic choice at a given point during parsing. A conflict may be of type *shift-reduce* or *reduce-reduce*. The shift-reduce conflict refers to a state where the parser cannot decide whether to shift or reduce. The reduce-reduce conflict refers to a state where there is more than once choice of a grammar rule for reduction. If a LR(k) parse table (without conflicts) can be constructed for a given grammar, then the grammar is deterministically parseable for every string described by the grammar, and therefore the grammar is unambiguous. Therefore LR(k) condition is a powerful test for statically checking unambiguity in grammars.

The class of grammars that can be deterministically parsed using LR(k) is the class of LR grammars. The class of LR grammars is a subset of the class of unambiguous grammars. A bigger class of unambiguous grammars than the class of LR but still a subset of the unambiguous class is the class of LR-Regular (LRR) grammars [23].

The parsing of LRR grammars is similar to the parsing of LR grammars but with one notable exception. Whereas in the case of LR(k) grammars, k look-ahead symbols are allowed to make a parsing decision, in the case of LRR grammars arbitrarily long look-ahead is allowed to make a parsing decision. The basic idea with LRR grammars is that the look-ahead information essential for determining the handle in any right sentential

forms can be represented as a finite number of regular sets. Parsing of LRR grammars is a two-pass process. The first pass involves reading the input from right to left and at each step a label is attached to certain terminals indicating the regular sets to follow. The second pass performs the actual parsing, where an LR(0)-like parser uses the look-ahead labels to make a deterministic choice. The class of LRR grammars are the biggest class of grammars that has proven to be unambiguous [23]. The one major drawback with LRR parsing is that the existence of the regular sets crucial to making a parsing decision is not decidable [22]. As a result, LRR parsing has proven to be impractical. For practical purposes, the class of LR(k) grammars is still the largest testable class of unambiguous grammars.

An example grammar [10] that is unambiguous but that is neither LR(k) (nor LRR) is shown below. The grammar describes the language of palindromes, `{a(a+b)*b, b(a+b)*b}`.

$$S : A$$
$$A : aAa \mid bAb \mid a \mid b \mid \epsilon;$$

I now explain a shift-reduce conflict (i.e. possible source of ambiguity) using an example.

2.4.1.1 LR(1) Parse Table – an Example

My example grammar is ambiguous and is shown below. As is evident from the grammar, the string '`p q r`' can be parsed in two ways.

$$S : AB$$
$$A : p \mid pq$$
$$B : qr \mid r$$

The LR(1) parse table for the example grammar is shown in Table 2.2. The rows in the parse table identify its states and the columns identify the shifting of a terminal or a non-terminal symbol. The grammar is augmented with a special end marker symbol '$' to signal the end of input (see column $ in Table 2.2). The action `acc` denotes the accept action when the end of the start rule is reached. To denote the rule reduced during a reduce action, each alternative of a given grammar is identified by a unique rule number. For my example grammar, the rule number for each alternative is as follows:

(1) $S : AB$
(2) $A : p$
(3) $A : pq$
(4) $B : r$
(5) $B : qr$

state	p	q	r	$	S	A	B
0	s2					g2	
1		s5	s4				g3
2		s6/r1	r1				
3				acc			
4				r3			
5			s7				
6		r2	r2				
7				r4			

Table 2.2: LR(1) parse table for the example grammar (see Section 2.4.1.1).

A shift action from state 'X' to state 'Y' when the next input token is a terminal t, has an entry `sY` at row 'X' in column 't'. A reduce action involving a rule 'N' at state 'X' when the next input token is terminal t, has an entry `rN` at column 't'. A `goto` action involving a non-terminal 'N' from state 'X' to state 'Y' has an entry `gY` at column 'N'. The accept action `acc` is added to the row labelled by the accept state in column $.

The grammar contains one shift-reduce conflict. In state 2, when token p has been processed, the rules are of the form $A : p\bullet$ and $A : p\bullet q$. If the next symbol is q, then we can either reduce by rule 1 (`r1`) or by shift q, and go to state (`s6`). That is, LR(1) parser is unable to make a decision on whether to shift or reduce.

2.4.1.2 Ambiguity Checking with LR(k)

Testing if a grammar is in the LR(k) class involves iteratively generating the parse table for increasing value of k until either the unambiguity condition is satisfied or a certain time limit is reached. If there is a value of k, for which a parse table can be constructed without conflicts, then each string has a unique parse and thus the grammar is unambiguous.

MSTA [29] and Hyacc [13] are two such implementations of an LR(k) test. Although the LR(k) approach is fairly simple to setup as an ambiguity checking tool, they do come with certain drawbacks. In case of a conflict, it is unclear whether the given grammar is ambiguous or that it is simply not LR(k). Each conflict comes with debug information that reveal the type of conflict along with the grammar rules involved. This conflict information is defined in terms of the state of the parse table and the multiple actions involved for that state. The description of a conflict is driven by the parser rather than by the grammar, and therefore the debug report is less intuitive. The main weakness of LR(k) test is that it doesn't provide a concrete example of ambiguity, and so it is difficult to say for sure, if the given grammar is ambiguous or otherwise.

2.4.2 Exhaustive

Exhaustive methods systematically search a grammar space to uncover an ambiguous string. Exhaustive methods are *derivation generators* that generate strings systematically from the start symbol of the grammar and then check the generated strings for duplicates. Although implementations of exhaustive methods are relatively fast, and can check millions of strings in a few seconds, one should be aware that grammars describe infinite languages and therefore it is impossible to definitely prove that they are unambiguous. Exhaustive approaches come in varying forms, from a simple brute-force search to a more sophisticated SAT Solver. An overview of the exhaustive methods now follows.

2.4.2.1 Gorn

Gorn [18] describes a breadth first search method to ambiguity detection in CFGs. The method involves generating strings of bounded length systematically from the start rule of the grammar, and then checking for duplicates in the generated strings. The method is parameterised by 'length' to define the level of derivation during string generation.

Gorn's ambiguity detection method works as follows. Strings are derived from the start symbol of the grammar. At each level of derivation, the completed and the incomplete derivations are tracked. The completed derivations (i.e. sentences) are compared with the previously generated sentences. If a sentence had been previously generated, then it has more than one derivation, and the grammar is therefore ambiguous. The method terminates on finding an ambiguous sentence. Alternatively, the method continues to search for ambiguous sentences by exploring the current set of incomplete derivations. Each incomplete derivation is explored by replacing each of its non-terminal by its set of alternatives.

2.4.2.1.1 Gorn's Method – an Example

I now describe Gorn's ambiguity detection method using the example grammar shown below:

$$S : A;$$
$$A : a \mid b \mid aA \mid Ab;$$

The grammar is ambiguous as the sentence `ab` can be derived in two ways:

$$S \Rightarrow_{\text{lm}} A \Rightarrow_{\text{lm}} aA \Rightarrow_{\text{lm}} ab$$
$$S \Rightarrow_{\text{lm}} A \Rightarrow_{\text{lm}} Ab \Rightarrow_{\text{lm}} ab$$

At level 0, the generated sentences include `{a,b}`, and the incomplete derivations include `{aA,Ab}`. At level 1, the incomplete derivations from level 0 are expanded. The

first derivation `aA` is expanded to generate sentences `{aa,ab}` and incomplete derivations `{aaA,aAb}`. Similarly, the second derivation `Ab` is expanded to generate sentences `{ab,bb}` and incomplete derivations `{aAb,Abb}`. Thus, the sentence `ab` has two derivations, and is therefore ambiguous.

Gorn's method is always correct: if a sentence is reported as ambiguous, then there exists multiple derivations for it. If the ambiguity can be reached within a few level of derivations (that is, if the ambiguity happens to be in the proximity of the start rule), then the method has a good chance of uncovering it. In practice however, ambiguities can lurk deep within a grammar and can be hard to reach. In such cases, the level of derivation required may be too high for the method to be practical.

2.4.2.2 Cheung and Uzgalis

Cheung and Uzgalis's (CandU) method [14] is an optimisation of Gorn's method. The ambiguity checking procedure uses a breadth first search with *pruning* to systematically generate strings and then check for duplicates. Pruning involves excluding string patterns that have already been explored or those that can not result in an ambiguity from further search. Pruning reduces search space of the grammar leading to quicker ambiguity detection.

CandU's ambiguity checking method can be viewed as a search tree. The search for ambiguous sentence starts from the start symbol of the grammar. The search tree is initialised with the root node using the start symbol of the grammar. A node in the tree is identified by a label `label` and a pattern set `ps`. The label denotes the partial sentence that has been derived thus far, and the pattern set `ps` contains one or more string patterns that is being searched. A string pattern denotes the part of the derivation that is yet to be completed. The search from a given node is initiated by expanding each of the pattern from its pattern set `ps`, starting with the leftmost pattern. A pattern is expanded by deriving its leftmost symbol to create child nodes. Prior to expanding the patterns, pruning is applied.

Besides the label and pattern set for each node, the search tree also maintains an additional set, the expanded pattern set `eps`. The string patterns (with terminal prefixes and suffixes removed) are stored in `eps`. A pattern p is excluded from future search if the non-terminal bounded substring $\bar{p}$ (i.e. the remainder of the string that is left after trimming the terminal prefixes and suffixes) has already been expanded (i.e. $\bar{p} \in$ `eps`) and additionally if either of the following is true: p is the only pattern in `ps`; or p is incompatible (two patterns are incompatible if their terminal prefixes or suffixes do not match) with all the other patterns in the pattern set. If, after pruning, the terminal prefixes of the patterns from the pattern set `ps` do not match, then the tree will split into multiple branches. The method continues to build the search tree until a node containing identical string

patterns is reached. At this point the method is said to have found an ambiguous string, and the search stops. I now explain CandU's ambiguity checking method using an example grammar.

2.4.2.2.1 CandU Method – an Example

CandU's ambiguity checking method is explained using a simpler version of the grammar from [14]. The grammar is shown below:

$$S : A;$$
$$A : cB \mid bC;$$
$$B : b \mid cbB \mid ccBBa;$$
$$C : \ c \mid cA \mid bCC;$$

A node is represented as <label,ps,eps>$_i$, where label refers to the partial sentence derived so far, ps refers to the pattern set, eps refers to the set that contains the patterns that have already been expanded, and i refers to the node number.

The search starts with the root node <ϵ,{S},{}>$_1$. Initially the string is empty ϵ and since S $\notin$ eps, it is expanded. Expanding S, a child node <ϵ,{A},{S}>$_2$ is created, with S $\in$ eps. Since A $\notin$ eps, it is expanded to create two child nodes <c,{B},{S,A}>$_3$ and <b,{C},{S,A}>$_4$, with A $\in$ eps. The sequence of expansions thus far is shown below:

<ϵ,{S},{}>$_1$
<ϵ,{A},{S}>$_2$
<c,{B},{S,A}>$_3$ <b,{C},{S,A}>$_4$

For node 3, since B $\notin$ ps, it is expanded to create two child nodes <cb,{},{S,A,B}>$_5$ (by applying the first alternative of B) and <cc,{bB,cBBa},{S,A,B}>$_6$ (by applying the second and the third alternatives of B), with B $\in$ eps. The expansion is shown below:

...
<c,{B},{S,A}>$_3$ ...
<cb,{},{S,A,B}>$_5$ <cc,{bB,cBBa},{S,A,B}>$_6$

Since node 5 has a empty pattern set, it is terminated. In case of node 6, the string pattern bB is incompatible with the other pattern cBBa as the terminal prefixes do not match, and the non-terminal bounded string B $\in$ eps, and therefore this pattern can be excluded from future search.

The search procedure that leads up to ambiguity (i.e. the node whose ps contains two identical string patterns) is as follows. Ambiguity is located on the branch containing node 4. Node 4 is expanded, since C $\notin$ ps, it is expanded to create two child nodes <bc,{A},{S,A,C}>$_7$ (from first and second alternative of C) and <bb,{CC},{S,A,C}>$_8$ (from third alternative of C), with C $\in$ eps. The expansion of node 4 looks as follows:

<ϵ,{S},{}>$_1$
<ϵ,{A},{S}>$_2$
... <b,{C},{S,A}>$_4$
<bc,{A},{S,A,C}>$_7$ <bb,{CC},{S,A,C}>$_8$

Node 7 will be terminated as A is the only symbol in ps and since A ∈ eps. For node 8, since CC ∉ eps, it is expanded by deriving the leftmost C to create two child nodes <bbc,{C,AC},{S,A,C,CC}>$_9$ (from first and second alternative of C) and <bbb,{CCC},{S,A,C,CC}>$_{10}$ (for third alternative of C), with CC ∈ eps. The expansion of node 8 looks as follows:

... <bb,{CC},{S,A,C}>$_8$
<bbc,{C,AC},{S,A,C,CC}>$_9$ <bbb,{CCC},{S,A,C,CC}>$_{10}$

For node 9, although C ∈ eps, it is not the only pattern in the ps. The patterns C and AC are expanded to create two child nodes <bbcc,{A,BC},{S,A,C,CC,AC}>$_{11}$ (from first two alternatives of C and first alternative of A) and <bbcb,{CC,CC},{S,A,C,CC,AC}>$_{12}$ (from third alternative of C and second alternative of A). The ps for node 12 contains two identical string patterns CC, indicating an ambiguity:

<bbc,{C,AC},{S,A,C,CC}>$_9$ ...
<bbcc,{A,BC},{S,A,C,CC,AC}>$_{11}$ <bbcb,{CC,CC},{S,A,C,CC,AC}>$_{12}$

The string bbcbCC therefore has two leftmost derivations:

$$S \Rightarrow_{\text{lm}} A \Rightarrow_{\text{lm}} bC \Rightarrow_{\text{lm}} bbCC \Rightarrow_{\text{lm}} bbcC \Rightarrow_{\text{lm}} bbcbCC$$
$$S \Rightarrow_{\text{lm}} A \Rightarrow_{\text{lm}} bC \Rightarrow_{\text{lm}} bbCC \Rightarrow_{\text{lm}} bbcAC \Rightarrow_{\text{lm}} bbcbCC$$

Pruning optimises the search by visiting certain repetitive string patterns only once, and thus increasing the chances of uncovering ambiguities. CandU's ambiguity checking method is always correct: if a sentence is reported as ambiguous, then the sentence has more than one derivation. Although CandU's method reduces the search space with pruning, the search for ambiguity is still exhaustive as the language described by most grammars is infinite.

2.4.2.3 AMBER

AMBER [36] uses a brute-force search, whereby sentences are generated systematically from the start rule of the grammar and then checked for ambiguity. AMBER relies on the Earley parsing algorithm [17], a top-down parsing technique that accepts any context-free grammar. Earley's recogniser is turned into a sentence generator and then extended for ambiguity detection.

2.4.2.3.1 Earley Parsing

For an input sentence, the Earley recogniser starts deriving from the start rule of the grammar. At each input position, a rule whose next symbol to be derived matches the token, is processed. The token is consumed and for the matched rule, the working position in the rule is advanced past the matched symbol. This process continues until the end of the input is reached, and if the sentence is valid, then the recogniser should have processed the start rule. In Earley parsing, the processing of a rule is denoted using a *dot* notation, where the part that precedes the dot indicate the symbols that have already been recognised and the part following the dot indicate the symbols that are to be expected. For a rule $A : \alpha\ \beta$, the notation $A : \alpha \bullet \beta$ denotes that α has already been parsed and β is to be expected. Such a 'dotted rule' is called an *item*. An item is a tuple containing a dotted rule and the input position at which the matching of the rule began. For a rule $A : \alpha\ \beta$, if the matching began at input position i, and where α has already been recognised, the item is formally written as $(A : \alpha \bullet \beta,\ i)$.

For each input position, the recogniser constructs a set of items, called an *item set*. The *kernel* of an item set is constructed using a *scanner*. For an input token t, the scanner adds those items from the current item set for whom the dot precedes the symbol t to the next item set. The scanning step indicates that the token t has been recognised. Formally, the item set at input position i is denoted as $S(i)$. If a is the next token in the input stream, and if the current item set $S(k)$ contains an item $(A : \alpha \bullet a\beta,\ i)$, then enter an item $(A : \alpha a \bullet \beta,\ i)$ to item set $S(k{+}1)$. The rest of the item set is constructed by the *closure* of the kernel. The closure is obtained by applying the *predictor* and the *completer* steps until no new items can be added to the set.

The predictor is invoked if the symbol following the dot is a non-terminal. A new item is added to the current item set for the non-terminal rule with the dot placed at the beginning of the right-hand side of the rule. For a grammar containing rules $A : \alpha\ B\ \beta$ `and` $B : \gamma$, if the current item set $S(k)$ contains an item $(A : \alpha \bullet B\ \beta,\ j)$, then item $(B : \bullet\gamma,\ k)$ is added to the current set. The completer is invoked if the dot appears at the end of a rule in an item. The item that triggered the processing of this rule is added to the current set with the dot advanced after the non-terminal. For our example grammar, if the current item set contains an item $(B : \gamma \bullet,\ i)$, then find items in $S(i)$ of the form $(A : \alpha \bullet B\ \beta,\ j)$, and add $(A : \alpha\ B \bullet \gamma,\ j)$ to the current set. The completer step indicates that the symbol B has been processed.

2.4.2.3.2 Earley Recogniser as Sentence Generator

The Earley recogniser is turned into a sentence generator and extended to form a ambiguity checker. When k input tokens have been processed and item set $S(k)$ has been constructed, items of the form $(A : \alpha \bullet a\ \beta,\ i)$, where the dot is followed by a terminal a,

are valid continuations of the current string. The terminals are added to a list and iterated over. For each terminal, the scanner constructs the next item set as if the terminal was the input token. The closure of this item set is constructed. For each item added to the item set, the algorithm checks for ambiguity. An ambiguity is said to have been found if there exists items in an item set whose dotted rules match, meaning that there exists multiple paths in deriving the current string. This procedure to extend the current string and checking for ambiguity is recursively invoked until strings of a given length are reached.

2.4.2.3.3 Ambiguity Characterisation

Ambiguity characterisation provides an insight into the structure of the ambiguity—the combination of rules and alternatives that cause the ambiguity. AMBER characterises ambiguity as *conjunctive* or *disjunctive*. An ambiguity is of type conjunctive if a string can be split in more than one way by sequence of symbols of a single alternative. An example grammar containing a conjunctive ambiguity is shown below:

$$
\begin{aligned}
S &: \ AB; \\
A &: \ a \mid ab; \\
B &: \ bc \mid c;
\end{aligned}
$$

In the above grammar, the sentence `abc` can be derived in two ways:

$$
\begin{aligned}
S \Rightarrow \ AB \ &\Rightarrow \ abB \ \Rightarrow \ abc \\
S \Rightarrow \ AB \ &\Rightarrow \ aB \ \Rightarrow \ abc
\end{aligned}
$$

In the first case, `A` is derived as `ab` and `B` is derived as `c`, whereas in the second case, `A` is derived as `a` and `B` is derived as `bc`.

An ambiguity is of type disjunctive if the same string can be derived from two different alternatives of a non-terminal. A trivial grammar containing disjunctive ambiguity is shown below:

$$
\begin{aligned}
S &: \ A; \\
A &: \ a \mid a;
\end{aligned}
$$

In the above grammar, the sentence `a` can be derived by either of the alternative of `A`.

In some of the literature, a conjunctive ambiguity is also known as a horizontal ambiguity, and a disjunctive ambiguity is also known as a vertical ambiguity. For consistency, from here on, I refer to conjunctive ambiguity as horizontal ambiguity, and disjunctive ambiguity as vertical ambiguity.

2.4.2.3.4 AMBER Options

AMBER comes with a number of options to influence the search. In *length* mode, strings of up to certain length are generated and checked for ambiguity. In *examples* mode, strings of varying length are generated and checked for ambiguity, however, the search is bound by the number of example strings that is checked. In *ellipsis* mode, non-terminals are also considered as tokens. That is, intermediate strings where non-terminals haven't been fully derived yet, are also checked for ambiguity. Checking for incomplete derivations increases the probability of detecting shorter examples of ambiguity.

2.4.2.3.5 AMBER Method – an Example

Ambiguity detection using AMBER is explained using the grammar from Section 2.4.2.2.1. The grammar contains a horizontal ambiguity, where the sentence '`bbccbc`' has two different derivations, and its substring '`ccbc`' can be split in two ways by the sequence of symbols '`CC`' of the third alternative of the non-terminal '`C`'.

Table 2.3 shows the item sets constructed in deriving the sentence '`bbccbc`'. The sentence that is currently being derived is tracked in s. For each scanned position i of s, the corresponding item set is denoted as S(i). Each item in an item set is numbered; an item at j in S(i) is denoted as S(i)(j). Each block of rows in the table represents an item set. The top of each block shows the item set that is being processed, followed by the sentence s. The bottom of each block shows the list of valid tokens v_t constructed from the item set. The tokens in v_t appear in the order the items were processed. A row in the item set represents an item.

For each item, the table shows the dotted rule it is processing, followed by the reference to the origin set at which the rule matching began, and a comment. The comment indicates the step that added the item to the set. A scanned item is annotated by the item in which the dotted rule precedes the scanned terminal. A predicted item is annotated by the item that created it. A completed item is annotated by the item whose completion triggered the addition of the completed item, and the previous derivation of the completed item where the dot preceded the derived non-terminal. In cases where following a valid token didn't result in an ambiguous string, the item sets related to those tokens are not shown in the table; for S(1) and S(2) the item sets related to the token c have been skipped.

Item	Dotted rule	Origin	Comment
		S(0), $s = \epsilon$	
(1)	YYSTART : • S EOF	0	start rule
(2)	S : • A	0	predict from (1)
(3)	A : • c B	0	predict from (2)

Item	Dotted rule	Origin	Comment
(4)	A : • b C	0	predict from (2)
		v_t ={c,b}	
		S(1), $s = b$	
(1)	A : b • C	0	scan from S(0)(4)
(2)	C : • c	1	predict from (1)
(3)	C : • c A	1	predict from (1)
(4)	C : • b C C	1	predict from (1)
		v_t ={c,b}	
		S(2), $s = bb$	
(1)	C : b • C C	1	scan from S(1)(4)
(2)	C : • c	2	predict from (1)
(3)	C : • c A	2	predict from (1)
(4)	C : • b C C	2	predict from (1)
		v_t ={c,b}	
		S(3), $s = bbc$	
(1)	C : c •	2	scan from S(2)(2)
(2)	C : c • A	2	scan from S(2)(3)
(3)	C : b C • C	1	complete from (1) and S(2)(1)
(4)	A : • c B	3	predict from (2)
(5)	A : • b C	3	predict from (2)
(6)	C : • c	3	predict from (3)
(7)	C : • c A	3	predict from (3)
(8)	C : • b C C	3	predict from (3)
		v_t ={c,b}	
		S(4), $s = bbcc$	
(1)	A : c • B	3	scan from S(3)(4)
(2)	C : c •	3	scan from S(3)(6)
(3)	C : c • A	3	scan from S(3)(7)
(4)	B : • b	4	predict from (1)
(5)	B : • c b B	4	predict from (1)
(6)	B : • c c B B a	4	predict from (1)
(7)	C : b C C •	1	complete from (2) and S(3)(3)
(8)	A : • c B	4	predict from (3)
(9)	A : • b C	4	predict from (3)
(10)	A : b C •	0	complete from (7) and S(1)(1)
(11)	S : A •	0	complete from (10) and S(0)(2)
(12)	YYSTART : S • EOF	0	complete from (11) and S(0)(1)

Item	Dotted rule	Origin	Comment
		v_t ={*b*,*c*,EOF}	
		S(5), $s = bbccb$	
(1)	B : b •	4	scan from S(4)(4)
(2)	A : b • C	4	scan from S(4)(9)
(3)	A : c B •	3	complete from (1) and S(4)(1)
(4)	C : • c	5	predict from (2)
(5)	C : • c A	5	predict from (2)
(6)	C : • b C C	5	predict from (2)
(7)	C : c A •	3	complete from (3) and S(3)(2)
(8)	C : b C • C	1	complete from (7) and S(3)(3)
		v_t ={*c*,*b*}	
		S(6), $s = bbccbc$	
(1)	C : c •	5	scan from S(5)(4)
(2)	C : c • A	5	scan from S(5)(5)
(3)	A : b C •	4	complete from (1) and S(5)(2)
(4)	C : b C C •	1	complete from (1) and S(5)(8)
(5)	A : • c B	6	predict from (2)
(6)	A : • b C	6	predict from (2)
(7)	C : c A •	3	complete from (3) and S(4)(3)
(8)	A : b C •	0	complete from (4) and S(1)(1)
(9)	C : b C C •	1	complete from (7) and S(3)(3)
		v_t ={*c*,*b*}	

Table 2.3: Item sets generated by AMBER for the ambiguous sentence '`bbccbc`'. The identical items S(6)(4) and S(6)(9) means that the derivation of '`C: bCC`' has been completed using two different paths, thus indicating an ambiguity.

AMBER wraps the start rule with the rule `YYSTART: S EOF`, where `YYSTART` and `EOF` indicate the new start symbol and the end of file symbol respectively. The search is seeded from the new start rule. For each position of the string, item sets are created using the scanner, predictor and the completer steps. The first token from v_t is picked to extend the current string. This process is recursively invoked until an ambiguity is found. An ambiguity is detected when an item set contains two identical items. Items S(6)(4) and S(6)(9) are identical, indicating that the rule '`C: bCC`' has been completed using two different paths. By tracing back each of the paths from these two items to the start rule, one obtains the following two leftmost derivations for the string '`bbccbc`':

$$S \Rightarrow_{\text{lm}} bC \Rightarrow_{\text{lm}} bbCC \Rightarrow_{\text{lm}} bbcAC \Rightarrow_{\text{lm}} bbccBC \Rightarrow_{\text{lm}} bbccbC \Rightarrow_{\text{lm}} bbccbc$$

$$S \Rightarrow_{\text{lm}} bC \Rightarrow_{\text{lm}} bbCC \Rightarrow_{\text{lm}} bbcC \Rightarrow_{\text{lm}} bbccA \Rightarrow_{\text{lm}} bbccbC \Rightarrow_{\text{lm}} bbccbc$$

AMBER's ambiguity detection algorithm is always correct: that is, if a string is detected to be ambiguous, then it has more than one derivation. AMBER's implementation of exhaustive search is fast and can check millions of examples in a short time. However, one should be aware that most grammars define infinite languages, and therefore it is impossible to definitively prove that they are unambiguous.

2.4.3 Approximation

Whilst exhaustive methods are fairly simple to implement and are always accurate, for practical grammars, they can run forever. If a grammar can be modified such that the ambiguity problem becomes decidable, then the search can always finish in finite time. Approximation methods transform the search space of a given grammar into an approximated one whose search space is finite. The language described by the approximated grammar is a superset of the original grammar, and within the approximated language, possible ambiguities are sought. The approximations applied are conservative, in the sense that the ambiguities that were present in the original language are also present in the (superset) approximated language. However, the approximations may also introduce potential ambiguities in the superset that didn't exist in the original language. As a result, in certain cases, approximation methods are unable to decide if the grammar is ambiguous or otherwise.

For a given grammar, approximation methods can produce three possible outcomes: the grammar is ambiguous, a string exists in the superset and in the original language; or the grammar is unambiguous, the superset of the language is unambiguous, and therefore the original language must be unambiguous; or it cannot decide, when a string is ambiguous in the superset but not in the original.

Ambiguity Checking with Language Approximations (ACLA) and Noncanonical Unambiguity (NU) are two tools that detect ambiguities in grammars using approximation methods. I now describe both these tools starting with ACLA.

2.4.3.1 ACLA

ACLA [10] is a static grammar ambiguity analyser based on the linguistic characterisation of a grammar. In ACLA, ambiguity is defined not in terms of the grammar but in terms of the language that the non-terminals of a grammar describe.

ACLA categorises ambiguity as vertical or horizontal. Two alternatives of a rule form a vertical ambiguity if the intersection of their languages is not empty. Two languages `L1` and `L2` are vertically ambiguous, if their intersection $\texttt{L1} \cap \texttt{L2} \neq \emptyset$. The sequence of symbols of an alternative form an horizontal ambiguity if they derive a string that can be parsed in two ways. That is, the sequence of symbols can be split in such as way that

the language described by the constituent parts overlap. For example, given a sequence of symbols `P Q`, if the language of `P`, `L(P)` = {`a,ab`} and the language of `Q`, `L(Q)` = {`bc,c`}, then the overlap of their languages `L(P)` ⋈ `L(Q)` = {abc}. Since the intersection and overlap properties is decidable for regular languges, ACLA transforms and extends the original language to an approximated language, a regular superset constructed using the Mohri and Nederhof (MN) transformation [30]. The approximated language is then explored for horizontal or vertical ambiguity.

Formally, for a given grammar G, if the approximation results in a grammar $\widehat{G}$, then $L(G) \subseteq L(\widehat{G})$. For details on the MN algorithm, I refer to [30] I. highlight one of its key properties. Given a grammar G containing rules `A:` α`A`β `and A:` γ, the MN transformation constructs an approximated language of the form `A:` $\alpha^j\gamma\beta^k$, where x^n indicates x appears in succession n times. That is, the approximated language keeps track of the order of the symbols but loses track of the fact α and β must appear in balance.

For a given grammar G, an approximated grammar $\widehat{G}$ is constructed using the MN approximation strategy. Grammar G is explored systematically for vertical or horizontal ambiguity. Each rule in G is explored for vertical ambiguity by computing the intersection between the approximated language of each of its alternatives. The approximated language for an alternative is computed from $\widehat{G}$. For a given rule A: $\alpha \mid \beta \mid \gamma$, intersection is computed between $L(\alpha)$ and $L(\beta)$, $L(\alpha)$ and $L(\gamma)$, and $L(\beta)$ and $L(\gamma)$. For each rule in G, its alternatives are explored for horizontal ambiguity by systematically splitting the alternative into two parts, and computing an overlap on their approximated languages. An alternative can be split between any two adjacent symbols. A split is denoted as $\leftrightarrow$. An alternative $U_1U_2...U_n$ containing n symbols, where $U_i \in N \cup T$, can be split in n-1 ways starting from ($U_1 \leftrightarrow U_2$ ... U_n) to (U_1 ... $U_{n\text{-}1} \leftrightarrow U_n$).

The approximated language of an alternative, or a sequence of symbols, is constructed by computing the language of each of its non-terminals using the rules from the approximated grammar. For a sequence of symbols `aBc`, its approximated language is computed by deriving the non-terminal B using the rules from $\widehat{G}$. Two alternatives are potentially vertically ambiguous if the intersection of their approximated languages is not empty. The shortest example from the intersection is a potential vertical ambiguity. An alternative is potentially horizontally ambiguous if an overlap of the approximated language of its constituent parts is not empty. The shortest example from the overlap is a potential horizontal ambiguity. The potentially ambiguous string is then verified against the original grammar. An Earley parser is used to parse the example string; if the parse is valid, then string is definitely ambiguous.

2.4.3.1.1 ACLA Method – an Example

ACLA's approach to ambiguity detection is now explained using an examples. The examples cover the three possible outcomes of an ACLA run for a given grammar: ACLA is

certain that the grammar is ambiguous; ACLA is not sure if the grammar is ambiguous; or ACLA is certain that the grammar is unambiguous.

For the example grammar shown in Section 2.4.2.1.1, ACLA found an ambiguity. The output indicating the ambiguity is shown below. ACLA correctly identifies the vertical ambiguity located within the third and the fourth alternatives of A (marked as `A[#3]` and `A[#4]` respectively). Both alternatives derive the string `ab`.

```
vertical check: A[#3] vs. A[#4]
*** vertical ambiguity: A[#3] <--> A[#4]
ambiguous string: "ab"
```

For Grammar 2.4.2.2.1, ACLA identified two potential horizontal ambiguities but no certain ones. The output indicating the potential ambiguities is shown below. The potential ambiguities are located within the third alternative of B and C (marked as `B[#3]` and `C[#3]` respectively). In both cases, the shortest example computed from the overlap was not a valid string with respect to the original grammar. In case of `B[#3]`, the alternative `ccBBa` cannot possible derive the string `ccbbba` (there seems to be an extra b in the middle), and quite rightly the parser failed to parse it. Likewise, in the case of `C[#3]`, the alternative `bCC` cannot derive the string `bccc` (there seems to be an extra c towards the end), and so the parser fails to parse it. In such cases, where the example chosen from the overlap or the intersection is not part of the original language, ACLA is unsure if the grammar is ambiguous.

```
 horizontal check: B[#3] at index 3
  a_overlap: "ccbbba"
  parse check: failed
*** potential horizontal ambiguity: B[#3]: "c" "c" B <--> B "a"
...
 horizontal check: C[#3] at index 2
  a_overlap: "bccc"
  parse check: failed
*** potential horizontal ambiguity: C[#3]: "b" C <--> C
the grammar might be ambiguous, but I'm not sure...
```

Since ACLA's approach to ambiguity detection is based on the linguistic properties of the grammar, namely the intersection and overlap of languages, it is able to establish unambiguity for grammars that are not LR(k). An example grammar that is not LR(k) but one that ACLA detects as unambiguous is shown below. The example grammar describes the language of even length palindromes of the form *aaaa*, *abba*, *baab* and so on.

$$S: \ A;$$
$$A: \ aAa \mid bAb \mid \epsilon;$$

The grammar is not LR(k)[2]. The language described by `aAa` is `a(a+b)*a` and the language described by `bAb` is `b(a+b)*b`. Since the two languages are disjoint, there is no vertical ambiguity between these two alternatives, and therefore the grammar is unambiguous. ACLA reports the example grammar as unambiguous.

2.4.3.2 Noncanonical Unambiguity

Noncanonical Unambiguity (NU) Test [34] is another approximation approach for ambiguity detection in CFGs. In NU Test, the grammar is first converted into a bracketed form by introducing two distinct terminal symbols to each alternative. A derivation symbol d_i is placed at the front of the alternative, and a reduction symbol r_i is placed at the end of the alternative, where i denotes a unique number for a rule, and $\{d_i, r_i\} \notin T$. A bracketed grammar is always unambiguous because the terminals d_i and r_i uniquely identify which alternative to apply whilst parsing. If two strings from the bracketed grammar differ only in their d_i and r_i symbols, then the original grammar is ambiguous.

To generate strings from the bracketed grammar, a *position graph* is constructed. The nodes in the position graph are positions in the strings that the bracketed grammar generates. A position is a dotted *item*, where the part of the alternative that precedes the dot has already been derived and the part that follows the dot is yet to be derived. The edges of the position graph represent the evaluation steps in the bracketed grammar, and is one of the following: *derivation*, *reduction* and *shift*. A derivation corresponds to an entry of an alternative, a reduction corresponds to an exit of an alternative, and a shift corresponds to a move over a symbol (terminal or non-terminal) within an alternative. The position graph describes the same language as the bracketed grammar, and every path through the graph corresponds to a parse tree in the original grammar.

Since the language described by programming language grammars is infinite, it is impossible to analyse their position graph in finite time. To obtain a graph that is tractable in size, the position graph is transformed to an approximated one using an equivalence relation. NU Test applies the "$item_0$" equivalence relation to construct the approximated graph. The approximated graph closely resembles an LR(0) parse automaton [26], where each node in the graph corresponds to a LR(0) item, and the edge corresponds to an action. For a grammar with rules $A : \alpha B \beta$, $B : \gamma$, $C : Bc$ identified by rule number i, j, and k respectively, the following three types of transitions are allowed in the approximated graph:

- A derivation transition of the form $A : \alpha \bullet B\beta \xrightarrow{(_j} B : \bullet\gamma$, where $\xrightarrow{(_j}$ indicates the rule j being derived.

[2]For a LR(k) parser, with k look-ahead tokens, for a sentence of length $> k$, when the current state contains `a•Aa`, after inspecting the next k symbols, there is not a clear choice: a reduce is possible using rule `A:` ϵ; or a shift is possible using rule `A: aAa`.

- A reduction transition of the form $B : \gamma\bullet \xrightarrow{\rangle_j} A : \alpha B\bullet\beta$, where $\xrightarrow{\rangle_j}$ indicates the rule j being reduced.
- A shift transition of the form $A : \alpha\bullet B\beta \xrightarrow{B} A : \alpha B\bullet\beta$, where $\xrightarrow{B}$ indicates non-terminal B being shifted.

The derivation and the shift transitions in the approximated graph are similar to those that occur in an LR(0) automaton. The reduction transition however is different: whereas in a LR(0) automaton, a move is made to a state depending on the reduced non-terminal and the state at the top of the parse stack, in an approximated graph there are reduction edges for every item that has the dot after the reduced non-terminal. For our example grammar, the LR(0) automaton in state $A : \alpha\bullet B\beta$ will move to state $A : \alpha B\bullet\beta$ when the non-terminal B is reduced. That is, the automaton will go back to the state from where the derivation of the reduced non-terminal started. In the case of the approximated graph, however, the derivation transition $A : \alpha\bullet B\beta$ can have a non-matching reduction $C : \alpha B\bullet c$. The approximated graph makes it possible to parse the initial part of the string with a rule and then reduce the remaining part of the string with a different rule. Thus, the language described by the approximated graph is a superset of the language described by the original position graph.

To find possible ambiguities, the $item_0$ position graph is traversed using two cursors simultaneously. If the two cursors take different paths through the position graph but shift the same set of tokens, then we have identified a possible ambiguity. Stated differently, if two paths differ only in their d_i and r_i symbols, then the original grammar is ambiguous. An efficient representation of all such simultaneous traversals is a Pair Graph (PG). The nodes of a PG represent the pair of cursors into the $item_0$ graph. The edges in a PG represent the (derive, shift, or reduce) transitions traversed by the cursors. A path in a PG thus describes two potential parse trees of the same string. If the two paths are not identical, then such a path in PG is called an *ambiguous path pair*. An ambiguous path pair denotes a potential ambiguity.

Since the approximated language is a superset of the original, there are potentially ambiguous strings—that is, ambiguous strings that are part of the approximated language but not the original. Thus, an approximated approach never reports false negatives but may report false positives.

2.4.4 AmbiDexter

AmbiDexter [7] uses an hybrid approach by marrying approximation techniques with exhaustive search. AmbiDexter first applies the NU Test to filter out every subset of the grammar that is proven to be unambiguous, before searching exhaustively on the result.

AmbiDexter extends the NU Test to identify rules that are unambiguous. A rule is considered unambiguous if it is not referenced in the ambiguous subsets of a grammar. The uncovering of the unambiguous rules of a grammar is based on the pair graph (PG) constructed by the NU Test. A rule is defined to be unambiguous if its position items do not belong to any of the ambiguous path pairs in a PG. The set of ambiguous path pairs in a PG is an over-approximation of the parse trees of ambiguous strings. Therefore, an alternative that is not referenced in the approximated set also means that it is certainly not used by the parse trees from the original grammar. Once the unambiguous rules have been identified, they are pruned from the PG, and resulting grammar contains only the rules that are potentially ambiguous.

The heuristic to prune a PG is as follows:

- From the set of all pairs in the PG, remove pairs that are not part of any ambiguous path pair.
- For an alternative to be ambiguous, all of its position items have to be used. Identify alternatives in PG where at least one of its items is not part of any ambiguous path pair. From each of the alternatives identified, all pairs that contain the alternatives' items can be safely pruned from the PG. This then triggers further pruning. All the dead ends and unreachable sections of the PG is further pruned.
- Consider that any path in the $item_0$ graph that describes a valid parse tree of the original grammar must both derive and reduce every alternative that it uses. That is, for every derive $\langle_i$ transition from $A : \alpha \bullet B\beta$ to $B : \gamma$, there should be a matching reduce $\rangle_i$ transition from $B : \beta$ to $A : \alpha B \bullet \beta$, and vice versa. Therefore, any cases where a derive transition is not matched by a corresponding reduce transition, and vice versa, then those transitions can be safely pruned from PG. After removing the invalid transitions, further pruning can be performed.

The pruning process is repeated until there are no further invalid paths to be removed from the PG. At the end of the pruning process, the alternatives that are left over are considered to be potentially ambiguous. Post pruning, two further refinements are performed. First, any non-terminals that are unreachable from the start rules are considered to be unambiguous, and are pruned. Second, for those non-terminals for which all the alternatives have been pruned, but are still referenced by other non-terminals, their productivity is restored. For each of those non-terminals, a new alternative containing a terminals only string is constructed from the shortest possible derivation of the non-terminal from the original grammar.

In addition to the LR0 approximation, AmbiDexter provides several other approximation filters, namely, SLR1, LALR1, and LR1. The filters come with varying precision: LR0 (low) to LR1 (high). The more powerful a filter is, the greater the portion of a grammar

it can filter out, but the longer it takes to do so. AmbiDexter has two modes of sentence generation: searching for sentences up to fixed length N, or searching for sentences from an initial length N to ∞. Although hybrid approaches perform better than their constituent approaches, they still rely on an exhaustive search, albeit on a smaller state space.

2.5 Summary

In this chapter, I first provided an overview of the different types of grammars and the languages they describe. I then introduced the basic concepts of parsing and parsing techniques. An introduction to ambiguity was given followed by an example of how ambiguity manifests in programming language grammars. I ended the chapter with a discussion on some of the extant ambiguity detection approaches to CFGs.

Chapter 3

Search-Based Ambiguity Detection

This chapter presents *SinBAD*: a breadth-based implementation for ambiguity detection in CFGs. The extant ambiguity detection approaches are deterministic and explore a grammar in 'depth'. By depth I mean that subset of grammar space is exhaustively searched. My hypothesis is that approaches that explore a grammar in 'breadth' have a greater chance of discovering ambiguity. By breadth I mean that the search space of a grammar is covered as much as a possible without focussing on any specific subset of the grammar. To that end, I have created a number of breadth-based heuristics for detecting ambiguities in grammars.

In this chapter, I first start with a comparison between depth-based and breadth-based approaches to ambiguity detection in CFGs. A case study is presented to demonstrate why depth-based approaches fail for certain grammars. I then present my breadth-based approach to ambiguity detection. A series of non-deterministic heuristics for ambiguity detection in grammars is presented. I start with a simple heuristic and then incrementally improve the heuristic to obtain higher quality results.

3.1 Depth-based Approach

A depth-based approach detects ambiguities by exploring a subset of a grammar in exhaustive detail. Since the search space of the language of a grammar is infinite and a depth-based approach exhaustive, the search has to be restricted in some ways. A bound is put in place for the exhaustive approaches to terminate. For instance, the search is restricted to explore strings of up to a certain length or is restricted to explore only certain subsets of the grammar. Two such ambiguity detection methods that search by depth include: AMBER [36] and CandU's method [14].

AMBER (see Section 2.4.2.3) detects ambiguities by systematically generating strings from the start rule of the grammar and then checking for duplicates. In AMBER, the search is restricted either by the string length or by the number of examples checked.

CandU's method detects ambiguities by generating strings of bounded length. Searching exhaustively on a subset of a grammars' search space means depth-based approaches perform quite well in detecting ambiguities within the subset focussed but not so well outside it. I now explain how a depth-based approach misses out on detecting a fairly trivial ambiguity using an example.

3.1.1 Why Depth-based Approaches Sometimes Fail?

Using the AMBER tool, I now illustrate on why depth-based approaches fail to detect ambiguity in certain cases. For my example, I invoke AMBER to search by examples on the altered Pascal grammar from Basten's grammar collection [5]. The example grammar contains the classic nested *if else* ambiguity. The relevant subset of the Pascal grammar in shown below:

```
1    program: PROGRAM IDENTIFIER external_files ';' block '.' ;
2    external_files: | '(' ident_list ')' ;
3    ident_list: IDENTIFIER |  ident_list ',' IDENTIFIER ;
4    block: opt_declarations compound_stmt ;
5    opt_declarations: | declarations ;
6    declarations: declaration | declarations declaration ;
7    declaration: proc_dec | var_dec | type_dec | label_dec | const_dec ;
8    compound_stmt: SBEGIN statements END ;
9    statements: statement |  statements ';' statement ;
10   statement:
11     | WITH rec_var_list DO statement
12     | FOR ident BECOMES expression direction expression DO statement
13     | REPEAT statements UNTIL expression
14     | WHILE expression DO statement
15     | IF expression THEN statement
16     | IF expression THEN statement ELSE statement
17     ...
18   ;
```

In the above grammar, upper-case symbols are terminals, and lower-case symbols are non-terminals. An experienced human will realise that lines 15 and 16, which represent different `IF` statements (often referred to as the 'dangling else' problem), cause an ambiguity. However, even if run for a significant time (more than an hour), AMBER doesn't detect ambiguity on the above grammar. AMBER is unable to detect the ambiguity due to its 'distance' from the start rule.

When AMBER is invoked, it starts generating sentences from the start rule of the grammar. Each of the alternatives of the start rule is explored systematically. An alternative is explored by expanding each of its non-terminals. Each of the non-terminals is then further explored by searching the rules it references exhaustively. This way AMBER continues to explore a grammar in depth by systematically exploring a subset of the grammar rules. If the source of the ambiguity is located near the head of the grammar, then AMBER has a good chance of detecting it. On the other hand, if the source of ambiguity is nested deep within the grammar (i.e. if the source of ambiguity is far away from the head of the grammar) then AMBER is less likely to detect it.

Running through the sequence that AMBER runs through is instructive. AMBER starts by expanding the `program` rule, and exploring each of its non-terminals. It then explores the non-terminal `external_files` by deriving a sequence of `IDENTIFIER`s (note that AMBER naturally limits recursion by the overall limit on sentence length). The non-terminal `block` is expanded by deriving a sequence of `declaration`s and `statement`s. The non-terminal `declaration` is expanded by exploring each of its five alternatives. Likewise the non-terminal `statement` is expanded by exploring each of its alternatives. By this point in the exploration, an infinite number of combinations could have been explored (because of recursion); even when recursion is bounded, the number of combinations is vast. AMBER's systematic and exhaustive search of each of the non-terminal referenced means that it has little chance of getting far enough into these combinations to find this ambiguity.

3.1.2 Breadth-based Approach

Whereas a depth-based approach covers a subset of the grammar in exhaustive detail, a breadth-based approach aims for grammar coverage. The search space of a grammar is explored as much as possible without delving into its specific subsets. By focussing less on any individual part, the search has a better chance of exploring the grammar more equally, potentially increasing the chances of uncovering ambiguity.

The search space of a grammar is defined by the language it describes. Since, for most practical purposes, the language described by a grammar is infinite, detecting an ambiguity within a grammars' language is an impossible task. Therefore, to give myself the best possible chance of finding an ambiguity, I aim to maximise grammar coverage by visiting as many of a grammars' rules and alternatives as I can, without focussing on any of its individual part in exhaustive detail.

Search-based techniques are known to perform reasonably well in finding 'good enough' solutions for problems whose search space is too big to be exhaustively scanned. Search-based techniques use heuristics that allow them to guide the search in seeking good enough solutions. In the following section, I provide an overview of the search-based techniques.

3.2 Search-based Techniques

In engineering disciplines, one often considers problems that involve large search spaces where finding a perfect solution is either theoretically impossible or practically infeasible. Such problems are often characterised by a search space where there is no perfect answer but one where there exist potentially many good solutions. Often the best we can hope for is a 'good enough' solution (i.e. one that falls within an acceptable tolerance), and it is precisely this characteristic that makes search-based techniques readily applicable to

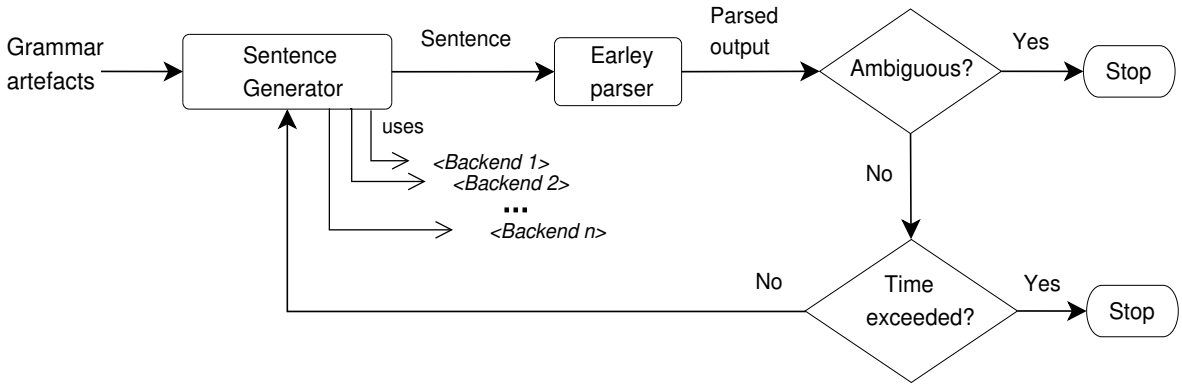

Figure 3.1: *SinBAD*

problems in engineering. As such, many problems in engineering have been adapted and successfully formulated as a search-based optimisation problem [21].

Search-based techniques seek 'good enough' solutions using a variety of algorithms. The simplest form of a search-based technique is a random search. A random search does not use a fitness function, and is thus unguided. The search-based techniques that use fitness function fall into two categories, local search and global search. Local searches operate with one candidate solution, and make moves based on the neighbourhood of that candidate solution. Local searches seek the best solution in the local neighbourhood. Local search techniques include hill climbing, simulated annealing, and tabu search. On the other hand, global searches operate by sampling many solutions in the search space all at once. Global searches seek the best solution (i.e. global optimum) in the search space. An example of a global search technique is genetic algorithms. For a given problem, a local search is applied first to understand the solution landscape before exploring a global search. Since this is the first time search-based techniques have been applied for detecting ambiguities in CFGs, I opt for the simplest form of search-based technique, random search.

3.3 *SinBAD*

To apply search-based techniques for detecting ambiguities in grammars, I created *SinBAD*, a simple tool which houses a number of pluggable ambiguity detection approaches. My ambiguity detection approaches are non-deterministic, and are intended to explore a grammar in breadth rather than depth. The algorithms are extremely simple, with the core of each explained in less than a page. Despite the simplicity of these algorithms, experimental results show that they perform at least as well as, and generally better than, more complex deterministic approaches. Furthermore, good results are found more quickly than by previous approaches.

Figure 3.1 shows *SinBAD*'s architecture. Given a grammar and a lexer, the *Sentence Generator* component generates random sentences using a given *backend*. A backend,

in essence, is an algorithm that governs how sentences are generated. For instance, a backend can use a unique scoring mechanism to favour an alternative when expanding a non-terminal, or one that can generate sentences of bounded length. The generated sentence is then fed to a parser to check for ambiguity (I use ACCENT [35], a fast Earley parser for this). The search stops when an ambiguity is found or when a time limit is exceeded.

3.4 Definitions

Before presenting my algorithms and descriptions, I first introduce some brief definitions (mostly standard) and notations.

A grammar is a tuple $G = \langle N,T,P,S \rangle$ where N is the set of non-terminals, T is the set of terminals, P is the set of production rules over $N \times (N \cup T)^*$ and S is the start non-terminal of the grammar. A 'symbol' is either a non-terminal or a terminal. A grammar rule is denoted as $A : \alpha$, where $A \in N$, and α is a sequence of strings drawn from $(N \cup T)^*$. For a given non-terminal A, the function $\mathcal{R}(A)$ returns the rule associated with A. For a rule r, its alternatives are denoted as $r.alts$. For a given alternative alt, $|alt|$ denotes the number of symbols it contains. For a grammar G, its size is defined as the number of non-terminals, $size(G) = |N|$.

A sentence is defined as a string that has been derived from the start symbol S of the grammar and is composed of only terminals. Formally, a sentence is defined as a string over T^*. A sentence is ambiguous if it can be parsed in more than one way. A grammar is ambiguous if there exists a sentence which is both accepted and ambiguous.

Given a list l, the function $\mathsf{rand}(l)$ returns a random element from l. The function $\mathsf{rand}(0, 1)$ returns a random floating point number between 0 (including) and 1 (excluding). For a list l, and item v, the function $\mathsf{append}(l, v)$ appends v to l. Given a list l, the function $\mathsf{enum}(l)$ enumerates the elements in l and returns a list of pairs, with each pair of the form (i,v), where v is the value from l at index i. For a given list l, the function $\mathsf{join}(l)$ returns a string by concatenating the elements from l with a single space. Given a list of numbers l, the function $\mathsf{min}(l)$ returns the minimum value from l.

For a set s, the function $\mathsf{len}(s)$ returns the length of the set s. For a given set s and an item v, the function $\mathsf{remove}(s, v)$ removes the item v from s. For a given object o, the function $\mathsf{copy}(o)$ returns a deep copy of the object o.

3.5 Search-based Backends

SinBAD houses several types of backends, ranging from purely random to heuristic driven, each of which generates a string by walking over a grammar and generating a string from

Algorithm 1 Algorithm to generate a sentence using the *purerandom* backend

```
1: function START(G)
2:     sen ← [ ]
3:     GENERATE(G, R(S), sen)
4:     return join(sen)
5: end function
6: function GENERATE(G, rule, sen)
7:     alt ← rand(rule.alts)
8:     for sym in alt do
9:         if sym ∈ N then
10:            GENERATE(G, R(sym), sen)
11:        else
12:            append(sen, sym)
13:        end if
14:    end for
15: end function
```

it. It is important to note that, by definition, this means that all backends always generate a valid sentence with respect to the grammar. Whilst in a random backend, sentences are generated by picking alternatives randomly, heuristic driven backends choose alternatives semi-randomly, with a bias towards certain alternatives. In the following subsections, I first explain *purerandom* – the simplest backend – before explaining the heuristic driven backends.

3.6 Purerandom

In the *purerandom* backend, sentences are generated by picking alternatives randomly. Algorithm 1 describes how a sentence is generated using the *purerandom* backend. The function START is initialised with a grammar G. The sentence *sen* is initialised with an empty list. Sentence generation is initiated from the start rule $\mathcal{R}(S)$ of the grammar (line 4). For a given rule *rule*, one of its alternatives is randomly selected (line 7). For each non-terminal symbol *sym* from the selected alternative, the function GENERATE is invoked recursively (line 11). The generated sentence *sen* is returned as a string when the function GENERATE unwinds from recursion.

Despite *purerandom*'s simplicity, it uncovers 52% of ambiguities on my experimental corpus. In a small experiment[1] performed on 5303 grammars from my grammar suite for a time limit of 10s, *purerandom* uncovered 1822 ambiguities. However, it has a significant problem which is that it often continues recursing ever-deeper into the grammar without

[1]Test results can be downloaded from: `https://figshare.com/s/7c86e867ec225a475a13`

unwinding, thus leading to non-termination. The results from my small experiment show that 87%[2] of the grammars failed to terminate. For my small experiment, the *purerandom* backend was restarted if it ran into recursion but the time limit still applied. I now explain how *purerandom* runs into non-termination using an example.

3.6.1 Non-termination in *purerandom*– an Example

To demonstrate the non-termination issue with *purerandom*, I use the SQL grammar from Section 2.3.1. For convenience, it is shown below.

```
1 %tokens SELECT, ID, FROM, TABLE, WHERE, STRING;
2
3 sql: select_stmt;
4 select_stmt: SELECT ID FROM TABLE WHERE expr;
5 expr: '(' expr ')' | expr '+' product | product;
6 product: product '*' term | term;
7 term: '(' expr ')' | STRING;
```

Listing 3.1: An Ambiguous SQL grammar

In case of my example SQL grammar, for the `expr` rule, 2 of the 3 alternatives are recursive. Because each alternative has an equal chance of being picked, this tends to lead to non-termination with *purerandom*. In a tiny experiment performed, where *purerandom* was invoked on the example SQL grammar, 39 out of 50 executions failed to terminate.

My implementation relies on the value of recursion limit set by the underlying stack. The recursion limit[3] for my Python interpreter stack is set to 1000. Running the example grammar with the recursion limit set to a much higher value only leads to much deeper recursive calls and delays the non-termination problem. Running the example SQL grammar with recursion limit set to 10000 resulted in 35 of the 50 executions failing to terminate (restarts were allowed on reaching recursion limit). It is possible to convert the current recursive implementation for sentence generation to a iterative one, by using a stack. During sentence generation, the alternatives (and the pointer within them) that are currently in process can be held in a stack. The stack can then be iteratively processed to generate a sentence. The stack based implementation is likely to resolve the issue with recursion, however it may lead to other issues, such as the underlying interpreters' VM running out of heap space.

[2] 4922 grammars failed to terminate.

[3] The default recursion limit in Python is 1000.

3.7 Heuristic Based Backends

The *purerandom* backend showed me that a random approach has promise, but since it is subject to non-termination, it can never be viable. Therefore, the real challenge is to find a sensible means of ensuring that sentence generation terminates. To that end, I have devised a number of heuristic driven backends. The core of my heuristic based backends is essentially the same; the difference lies in the approach used by each backend to address the issue with the non-termination.

A heuristic based backend generates sentences by picking alternatives randomly in general, but occasionally picking them semi-non-randomly in order to encourage sentence generation to terminate. That is, whilst generating a sentence, when the heuristic has recursed 'sufficiently deep enough' into the grammar, certain alternatives are favoured to encourage termination. Therefore, a heuristic based backend is parameterised by a user-configurable integer D[4]. Once the algorithm has recursed beyond depth D, alternatives are favoured.

Each of my heuristic based backends is specific in its approach to how it picks alternatives for favouritism. My approaches vary from using a simple scoring mechanism where alternatives are assigned scores based on how quickly they can be derived, to a more sophisticated mechanism where alternatives that generate strings of bounded length are picked semi-deterministically. In devising these approaches, I have followed a simple convention: I start with a base heuristic and then look to improve it, fixing any obvious flaws, to create new heuristics. Further, exploring various approaches that are spread across the design space, allows me to understand the strengths and weaknesses of each of them. I now describe each of my heuristic based backend, starting with *dynamic1*.

3.7.1 The *dynamic1* Backend

Picking alternatives purely randomly leads to non-termination. *dynamic1* mitigates this issue by sometimes favouring alternatives which it believes are likely to encourage termination. The favouring of alternatives is triggered when the depth of recursion has exceeded a threshold D. If a rule has an alternative with only terminals (which will clearly terminate immediately), that is picked non-deterministically. If all alternatives contain at least one non-terminal, the alternatives are scored and the alternative with the lowest value picked.

The scoring mechanism is based on two counts for each non-terminal. During sentence generation, as I recurse into the grammar, I keep track of the number of times each non-terminal has been entered and exited. Based on these two counts, a non-terminals' score is defined as the number of incomplete derivations (i.e. (number of times entered)-(number

[4]Setting D to ∞ provides equivalent behaviour to the naive non-terminating approach.

Algorithm 2 Algorithm to generate a sentence using the *dynamic1* backend

```
1: function START(G, D)
2:     for A in N do
3:         R(A).entered = R(A).exited = 0
4:     end for
5:     sen ← [ ]
6:     GENERATE(G, R(S), sen, d=0, D)
7:     return join(sen)
8: end function

9: function GENERATE(G, rule, sen, d, D)
10:    d ← d + 1
11:    rule.entered ← rule.entered + 1
12:    if d > D then
13:        scores ← [ ]
14:        for alt in rule.alts do
15:            score_alt ← CALC-ALT-SCORE(G, alt)
16:            append(scores, score_alt)
17:        end for
18:        alt ← FAVOUR-ALTERNATIVE(G, rule, scores)
19:    else
20:        alt ← rand(rule.alts)
21:    end if
22:    for sym in alt do
23:        if sym ∈ N then                    ▷ If sym is a non-terminal
24:            GENERATE(G, R(sym), sen, d+1, D)
25:        else
26:            append(sen, sym)               ▷ Build sentence
27:        end if
28:    end for
29:    rule.exited ← rule.exited + 1
30:    d ← d − 1
31: end function
```

of times exited)) divided by the times entered. The lower the score, the more suggestive that the rule is quick to derive; conversely the higher the score, the more suggestive that the rule is slow to derive. The score of an alternative is calculated as the sum of the score of each of its constituents' symbol (with terminals being scored as 0).

Algorithm 2 describes the *dynamic1* backend. The function START is initialised with a user-defined grammar G, and a threshold depth D, a depth beyond which alternatives are

Algorithm 3 *dynamic1*'s algorithm to calculate the score of an alternative

```
1: function CALC-ALT-SCORE(G, alt)
2:     score ← 0
3:     for sym in alt do
4:         if sym ∈ N then                          ▷ If sym is a non-terminal
5:             if R(sym).entered > 0 then
6:                 score_sym ← 1 - R(sym).exited/R(sym).entered
7:                 score ← score + score_sym
8:             end if
9:         end if
10:    end for
11:    return score
12: end function
```

favoured. For each rule *rule*, two counts are recorded: the number of times it has been entered (*rule.entered*) and the number of times it has been exited (*rule.exited*). These two counts are initialised to 0 (lines 2–4). The function GENERATE is invoked to start sentence generation from the start rule $\mathcal{R}(S)$ of the grammar, with d=0 (line 6).

Given a rule *rule*, the function GENERATE continues to pick alternatives randomly (line 20) until the heuristic has recursed beyond the threshold depth D. When the sentence generation has recursed beyond D, alternatives are favoured. For each alternative *alt* of the rule *rule*, the function CALC-ALT-SCORE is invoked to calculate its score (line 15). *scores*, containing the scores of the alternatives for *rule* is then fed to the function FAVOUR-ALTERNATIVE to favour an alternative (line 18).

The score of an alternative is the sum of the scores of its constituent symbols. Algorithm 3 shows how the score for an alternative is calculated for a given grammar G and an alternative *alt*. The score of the alternative *score* is initialised as 0 (line 2). For each non-terminal symbol *sym* of the alternative, if *sym* has already been visited (i.e. if $\mathcal{R}(sym).entered > 0$), then the *sym*'s score is calculated as the ratio of the number of incomplete derivations divided by the number of times it has been entered (lines 5 and 6). The score of the *sym* is then added to the score of the alternative (line 7). *score* is returned as the score of the alternative.

Given a rule, once the score of its alternatives are calculated, the favouritism returns the alternative with the lowest score. In the case of a tie, one of the low scoring alternatives is randomly picked.

Algorithm 4 describes how *dynamic1* favours low scoring alternatives for a given grammar G, rule *rule*, and *scores* containing the scores of the alternatives of rule *rule*. The lowest score $score_{min}$ from *scores* is calculated. Alternatives with the lowest score are identified and recorded in $alts_{min}$ (lines 4–7). One of the low scoring alternatives from $alts_{min}$ is

Algorithm 4 The *dynamic1* favour alternative algorithm

```
1: function FAVOUR-ALTERNATIVE(G, rule, scores)
2:     score_min ← min(scores)
3:     alts_min ← [ ]
4:     for i,alt in enum(rule.alts) do
5:         if scores[i] = score_min then
6:             append(alts_min, alt)
7:         end if
8:     end for
9:     return rand(alts_min)
10: end function
```

then randomly picked (line 9).

dynamic1's favouring of low scoring alternatives naively means that in certain cases, it fails to terminate. In a small experiment[5] performed, where 5603 grammars were evaluated, sentence generation did not terminate on 4% of the grammars. My grammar corpus for the small experiment contained 3300 Boltzmann (see Section 4.1) and 2303 mutated grammars (see Section 4.2). *dynamic1* was run with D=9 for the Boltzmann grammars and D=22 for the mutated grammars. The value of D for each grammar set were the best performing values for the *dynamic1* backend from the fine dimensioning experiment (see Section 5.11). *dynamic1* failed to terminate on 211 Boltzmann and 6 mutated grammars. This is due, in an unintended irony, to the one deterministic part of *dynamic1*: the favouring of alternatives. When the favouring of alternatives is triggered, alternatives are scored, and one of the low scoring alternatives is then picked semi-deterministically. If one alternative always has the lowest score, then it will be picked every time. In certain cases, picking the same set of alternatives leads to recursive cycles. I now explain how *dynamic1*'s favouring of alternatives leads to recursive cycles for an example grammar.

3.7.1.1 Non-termination in *dynamic1* – an Example

For a given grammar, if there exists two non-terminals A and B, where the low scoring alternative of A's rule references B, and the low scoring alternative of B's rule references A, then this leads to recursive cycles. *dynamic1*'s issue with non-termination is explained using an example grammar[6] from my experimental corpus. For my example grammar, symbols starting with `TK_` are terminals, and the rest are non-terminals. The relevant subset of the example grammar that contributes to non-termination is shown in the following fragment:

[5] Results can be downloaded from: `https://figshare.com/s/0f73a2b0c4d94f36223a`.

[6] `https://github.com/nvasudevan/experiment/blob/master/grammars/boltzcfg/11/2.acc`

```
D : NXLPR SUHVQ TK_HHUC;
NXLPR : TK_JSI | SUHVQ | CUEV SUHVQ | YRH TK_AWYYX;
YRH : OZZ;
OZZ : TK_HHUC D TK_HWMU TK_LCTO D | TK_HWMU YRH;
```
(3.1)

For the example grammar, whilst the sentence generation is in progress, if at a given point of time, when $d < D$ (d is the current depth of the recursion from Algorithm 2), the number of times entered and exited for all of the rules – `D`, `NXLPR`, `YRH`, and `OZZ` – to be 0 and 0 respectively.

Since $d < D$, for a given rule, alternatives are picked randomly. If rule `D` is entered, its only alternative is selected. Subsequently, the rule `NXLPR` is entered, and is expanded. The random selection picks the last alternative '`YRH TK_AWYYX`'. The rule `YRH` is expanded by picking its only alternative '`OZZ`'. If at this point, $d > D$ (line 12 in Algorithm 2), then the favouring of the alternatives is triggered. Since each of the symbol '`D`', '`NXLPR`', and '`YRH`' has been entered once but not exited, their score will be 1. The score of the first and the second alternative of rule `OZZ` then becomes 2 and 1 respectively. Since *dynamic1*'s scoring mechanism always favours the low scoring alternatives, the second alternative '`TK_HWMU YRH`' will get picked. Deriving '`YRH`' will lead us back to the rule `OZZ`. Since none of the derivations of '`D`' and '`YRH`' are complete yet, the scores of the alternatives of rule `OZZ` remain unchanged. For rule `OZZ`, the second alternative '`YRH TK_AWYYX`' is picked again. Thus, the sentence generation enters a recursive cycle where the `YRH`'s only alternative '`OZZ`' and the `OZZ`'s second alternative '`TK_HWMU YRH`' are picked in a cyclic fashion.

For a sentence generated from the above example grammar, where *dynamic1* (run with D=10) ran into non-termination, the output of the sentence generation is available for download from: `https://figshare.com/s/60ee866be4629e3a1e06`.

For the above grammar, for the sentence generation to unwind from recursion, for rule `OZZ`, its first alternative needs to be picked. But because *dynamic1* scores its alternatives based on the aggregated score of all of its symbols, the first alternative always ends up with a high score, and is thus not favoured. Stated differently, *dynamic1*'s scoring mechanism is biased towards alternatives with fewer non-terminals. A possible solution is to score an alternative, not based on the aggregated score of its symbols but on the individual score from one of its symbols. This then allows a fair comparison between alternatives containing varying number of non-terminals for a given rule. I now explore this idea in my next backend.

3.7.2 The *dynamic2* Backend

When applying favouritism, picking alternatives with fewer non-terminals often leads to non-termination. The motivation for the *dynamic2* backend is to mitigate the bias

Algorithm 5 *dynamic2*'s algorithm to calculate the score of an alternative

```
1: function CALC-ALT-SCORE(G, alt)
2:     score_max ← 0
3:     for sym in alt do
4:         score ← 0
5:         if sym ∈ N then                          ▷ If sym is a non-terminal
6:             if R(sym).entered > 0 then
7:                 score ← 1 − R(sym).exited/R(sym).entered
8:             end if
9:         end if
10:        if score > score_max then
11:            score_max ← score
12:        end if
13:    end for
14:    return score_max
15: end function
```

towards picking alternatives with fewer non-terminals when applying favouritism. To give alternatives with higher numbers of non-terminals an equal chance of selection when applying favouritism, I apply a small tweak to *dynamic1*'s scoring mechanism. Instead of defining an alternative's score to be the aggregate of its non-terminal's scores, I define an alternative's score to be that of the highest non-terminal.

Algorithm 5 describes how the score of an alternative is calculated in *dynamic2* for a given grammar G and an alternative *alt*. The alternative's score starts as 0 (line 2). The algorithm then iterates over each non-terminal symbol *sym*. If the non-terminal has already been visited (i.e. $\mathcal{R}(sym).entered > 0$), then *sym*'s score is calculated as the number of incomplete derivations divided by the number of times the rule is entered (lines 6 and 7). If *sym*'s score is higher than the alternative's current score, then the latter is updated to the former (lines 10 and 11).

Although the *dynamic2* scoring mechanism mitigates the bias towards picking alternatives with fewer non-terminals, it still runs into non-termination on a tiny number of cases. In a small experiment[7] performed, where 5603 grammars were evaluated, whereas in the case of *dynamic1* 4% of the grammars did not terminate, in case of *dynamic2* only 0.66% of the grammar did not terminate. My grammar corpus for the small experiment contained 3300 Boltzmann (see Section 4.1) and 2303 mutated grammars (see Section 4.2). *dynamic2* was run with D=9 for the Boltzmann grammars and D=16 for the mutated grammars. The value of D for each grammar set were the best performing values for the *dynamic2* backend from the fine dimensioning experiment (see Section 5.11). *dynamic2* failed to terminate

[7]Results can be downloaded from: `https://figshare.com/s/c90b637ed9e8c818bfcc`.

on 4 Boltzmann and 33 mutated grammars. *dynamic2* runs into non-termination, as part of the heuristic that favours alternatives, is still, in a small way, deterministic.

The cause of non-termination in *dynamic2* is as follows. When favouring of alternatives is triggered and alternatives are scored, those with low scores are selected and then one of them is non-deterministically picked. For a given rule, if the low scoring alternative also happens to be recursive (direct or indirect), then the part of the heuristic that favours alternatives ends up always picking the recursive alternative. I now explain how *dynamic2* runs into the non-termination on a trivial example grammar.

3.7.2.1 Non-termination in *dynamic2* – an Example

Consider the trivial grammar with rule `P: 'p' P | Q`, where P, and Q are non-terminals and p is a terminal. For the example sentence, the recursion limit of the underlying Python stack is set to 1000. For the rest of this thesis, unless otherwise stated, the recursion limit of the underlying Python stack is 1000.

For the example grammar, assume that the sentence generation is in progress, and at a given point of time when $d > D$, P has a score of 0 (5 of its 5 derivations complete) and Q has a score of 0.8 (1 of 5 derivations complete). Now, if rule P is entered, the favouring of alternatives will always pick the low scoring (recursive) alternative '`'p' P`'. The sentence generator continues to pick the recursive alternative until P's score $\geq$ Q's score. This happens after 20 recursive calls to P, when its score becomes 0.8 (5 of its 25 derivations complete). When P is subsequently entered, P's score becomes 0.807 (5 of its 26 derivation complete), and on this occasion, the second alternative '`Q`' with a lower score (0.8) is picked. Subsequently, all the P's recursive calls unwind and now P's score is 0 (26 of its 26 derivations complete), and Q's score is 0.666 (2 of its 6 derivations complete).

As the sentence generation progresses, if $d > D$ and if P is entered, then favouritism will always pick the low scoring (recursive) alternative '`'p' P`'. The sentence generation continues to pick the recursive alternative until P's score $\geq$ Q's score. This happens after 52 recursive calls to P, when P's score becomes 0.666 (26 of its 78 derivations complete). When P is subsequently entered, P's score becomes 0.67 (26 of its 79 derivations complete), and on this occasion, the second alternative '`Q`' with a lower score (0.666) is picked. Subsequently, all of P's recursive calls unwind and now P's score is 0 (79 of its 79 derivations complete) and Q's score is 0.571 (3 of its 7 derivations complete). Whilst in the former case, only 20 recursive calls were needed for P's and Q's score to reach parity, in the latter case, 52 calls were needed.

As the sentence generation progresses and d>D, each time Q is picked and the sentence generation unwinds, the number of recursive calls to P that is needed for its score to reach parity with the Q's score, also increases. The number of recursive invocations to P until its score reaches parity with the Q's score at various points during the sentence

generation run are 106, 299, 448, 641, 881 and 1175. In the final case, the number of recursive invocations (1175) needed exceeds the recursion limit (1000) of the Python stack. The non-termination issue with *dynamic2* is described for an Boltzmann grammar in Appendix A.

3.7.2.2 Summary

dynamic2's scoring mechanism significantly mitigates the issue that *dynamic1* had, namely the bias towards picking alternatives with fewer non-terminals. However, *dynamic2*'s favouring of low scoring alternatives is, still in a small way, deterministic. Given a rule, if an alternative contains a symbol that references back to itself, has a lower score, then it will get picked every time. If the recursive alternative happens to have an extremely low score, then this will lead to deep recursion and eventually the sentence generator will run out of stack space. One possible idea to mitigate non-termination even further, is to occasionally pick an alternative other than the lowest scoring alternative whilst still preserving the general approach of *dynamic2*. I now explore this idea in the next section.

3.7.3 The *dynamic2*$_{rws}$ Backend

Whilst the *dynamic2*'s scoring mechanism gets fairly close to mitigating non-termination, there are still cases where the sentence generation struggles to unwind from deep recursion. A case in point is when low scoring alternatives that are recursive are picked every time when applying favouritism. My approach to mitigate non-termination involves a probabilistic selection of low scoring alternatives.

My probabilistic selection involves occasionally picking an alternative other than the one with the lowest score when applying favouritism. During sentence generation, when the heuristic has recursed beyond the threshold depth D, I continue to pick low scoring alternatives. Since picking the low scoring alternatives always leads to non-termination in case of recursive alternatives, I apply a probabilistic weight W to pick an alternative other than the low scoring alternatives. To do so, I source alternatives based on a proportionate selection.

To pick alternatives based on proportionate selection, I use the roulette wheel method. In a roulette wheel selection, alternatives are picked in proportion to their scores. Imagine a roulette wheel with sectors of size proportional to the alternatives' scores. Selecting an alternative is then equivalent to choosing randomly a point on the wheel, and locating the corresponding sector. Since, in my case, I want to favour low scoring alternatives, I convert this minimisation problem to a maximisation one by subtracting each alternatives' score from 1. To the resultant scores, a roulette wheel is applied to pick an alternative. If all of the alternatives of a rule have a resultant score of 0 (i.e. there is at least one

symbol in an alternative for which all of its derivations are incomplete), then a roulette wheel can't be applied. In such cases, an alternative is picked randomly. In summary, by simply not picking the low scoring alternatives occasionally, I mitigate non-termination.

Algorithm 6 describes the *dynamic2*$_{rws}$ backend. The function START is initialised in a similar way to the *dynamic1* backend but with one exception: the function START accepts an additional parameter, a probabilistic weight W on when to trigger proportionate selection.

Given a rule *rule*, the function GENERATE continues to pick alternatives randomly (line 24). When the heuristic has recursed beyond the threshold depth D, alternatives are favoured. For each alternative *alt* of the rule *rule*, the function CALC-ALT-SCORE is invoked to calculate its score (line 15). The scores of the alternatives are then fed to the function FAVOUR-ALTERNATIVE to favour a low scoring alternative (line 21). Picking a low scoring alternative that is also recursive every time leads to non-termination. To mitigate non-termination, low scoring alternatives are probabilistically selected. A probabilistic weight W is applied to pick low scoring alternatives, and instead alternatives are picked in proportion to their scores by invoking the function WEIGHTED-SELECTION (line 19).

Algorithm 7 describes how an alternative is chosen based on proportionate selection for a given rule. The function WEIGHTED-SELECTION is initialised with rule *rule* and *scores*, containing the scores of the alternatives of rule *rule*. The heuristic to pick an alternative based on proportionate selection is as follows. The scores are first converted by calculating the 'complement of 1' for each score. To the resultant scores, the function ROULETTE-WHEEL is applied to select an alternative (line 7). In cases, where all of the alternatives of rule *rule* has a resultant score of 0, an alternative is picked randomly (line 10).

My implementation of the roulette wheel is based on the article [9]. Algorithm 8 describes the heuristic for selecting an alternative based on a roulette wheel for a given rule. The function ROULETTE-WHEEL is initialised with *scores*, containing the scores of the alternatives for a rule. The heuristic for a roulette wheel selection is as follows. A random value $score_{rnd}$ is picked between 0 and *sum(scores)* (line 2). Each score s from *scores* is subtracted from $score_{rnd}$. The item corresponding to the score when $score_{rnd} < 0$ is chosen.

dynamic2$_{rws}$'s weighted approach to favour low scoring alternatives significantly mitigates the non-termination problem. In a small experiment[8] performed, where 5603 grammars were evaluated, whereas in the case of *dynamic2* 0.66% of the grammars did not terminate, in case of *dynamic2*$_{rws}$, only 0.5% of the grammar did not terminate. My grammar corpus for the small experiment contained 3300 Boltzmann (see Section 4.1) and 2303 mutated grammars (see Section 4.2). *dynamic2*$_{rws}$ was run with D=11 and W=0.1334025 for the

[8]Results can be downloaded from: `https://figshare.com/s/8a0bc6afb37b7b74db39`.

Algorithm 6 Algorithm for the *dynamic2*$_{rws}$ backend

```
1: function START(G, D, W)
2:     for A in N do
3:         R(A).entered = R(A).exited = 0
4:     end for
5:     sen ← [ ]
6:     GENERATE(G, R(S), sen, d=0, D, W)
7:     return join(sen)
8: end function

9: function GENERATE(G, rule, sen, d, D, W)
10:    d ← d + 1
11:    rule.entered ← rule.entered + 1
12:    if d > D then
13:        scores ← [ ]
14:        for alt in rule.alts do
15:            score_alt ← CALC-ALT-SCORE(G, alt)
16:            append(scores, score_alt)
17:        end for
18:        if rand(0, 1) < W then
19:            alt ← WEIGHTED-SELECTION(rule, scores)
20:        else
21:            alt ← FAVOUR-ALTERNATIVE(G, rule, scores)
22:        end if
23:    else
24:        alt ← rand(rule.alts)
25:    end if
26:    for sym in alt do
27:        if sym ∈ N then                    ▷ If sym is a non-terminal
28:            GENERATE(G, R(sym), sen, d+1, D, W)
29:        else
30:            append(sen, sym)
31:        end if
32:    end for
33:    rule.exited ← rule.exited + 1
34:    d ← d − 1
35: end function
```

Boltzmann grammars and D=20 and W=0.0363825 for the mutated grammars. The value of D for each grammar set were the best performing values for the *dynamic2*$_{rws}$ backend

Algorithm 7 Algorithm to select an alternative based on proportionate selection

```
1: function WEIGHTED-SELECTION(rule, scores)
2:     scores_wgt ← [ ]
3:     for s in scores do
4:         append(scores_wgt, (1 - s))
5:     end for
6:     if sum(scores_wgt) > 0 then
7:         i ← ROULETTE-WHEEL(scores_wgt)
8:         return rule.alts[i]
9:     end if
10:    return rand(rule.alts)
11: end function
```

Algorithm 8 Algorithm for a roulette wheel selection

```
1: function ROULETTE-WHEEL(scores)
2:     score_rnd ← rand(0,1) * sum(scores)
3:     for i, s in enum(scores) do
4:         score_rnd ← score_rnd - s
5:         if score < 0 then
6:             return i
7:         end if
8:     end for
9: end function
```

from the fine dimensioning experiment (see Section 5.11). $dynamic2_{rws}$ failed to terminate on 1 Boltzmann and 23 mutated grammars. I now explain how $dynamic2_{rws}$ runs into non-termination for an example Boltzmann grammar from my experimental corpus.

3.7.3.1 Non-termination in $dynamic2_{rws}$– an Example

The relevant subset of the Boltzmann grammar[9] from my experimental corpus that runs into non-termination is shown below. Some of the alternatives in my example grammar are long, and so for convenience, I only show the relevant non-terminals that contribute to non-termination. Non-terminal `PFGQ` maps to a terminal and so is easily derived.

[9] `https://github.com/nvasudevan/experiment/blob/master/grammars/boltzcfg/72/9.acc`

Non-terminal GPT is derived to a sequence of terminals in under 10 steps.

```
DXQML : DXQML | K
K : EZ
EZ : PFDMK
PFDMK : K | JVL
JVL : X
X : DXQML | JVL | CPM
CPM : CPM | GPT PFGQ;
```

It is not hard to see that various subsets of the above grammar form a recursive cycle. DXQML's first alternative is recursive. Deriving DXQML with its second alternative K, we have two possibilities:

$$\texttt{DXQML} \Rightarrow \texttt{K} \Rightarrow \texttt{EZ} \Rightarrow \texttt{PFDMK} \Rightarrow \texttt{K}$$
$$\texttt{DXQML} \Rightarrow \texttt{K} \Rightarrow \texttt{EZ} \Rightarrow \texttt{PFDMK} \Rightarrow \texttt{JVL} \Rightarrow \texttt{X}$$

The first possibility results in a recursive cycle. For the second possibility, on following the derivation for X, we get further three possibilities:

$$\texttt{DXQML} \Rightarrow \texttt{K} \Rightarrow \texttt{EZ} \Rightarrow \texttt{PFDMK} \Rightarrow \texttt{JVL} \Rightarrow \texttt{X} \Rightarrow \texttt{DXQML}$$
$$\texttt{DXQML} \Rightarrow \texttt{K} \Rightarrow \texttt{EZ} \Rightarrow \texttt{PFDMK} \Rightarrow \texttt{JVL} \Rightarrow \texttt{X} \Rightarrow \texttt{JVL}$$
$$\texttt{DXQML} \Rightarrow \texttt{K} \Rightarrow \texttt{EZ} \Rightarrow \texttt{PFDMK} \Rightarrow \texttt{JVL} \Rightarrow \texttt{X} \Rightarrow \texttt{CPM}$$

The first possibility results in a recursive cycle (DXQML derives to DXQML). The second possibility also results in a recursive cycle (JVL derives to JVL). The third possibility is the only path that results in a successful derivation. Effectively, during sentence generation, if rule DXQML is entered, the only way to complete its derivation is through X's third alternative 'CPM'. At a given point of time during sentence generation, if CPM has a relatively higher score than that of DXQML or JVL, and if rule DXQML is entered, then the sentence generation will run into non-termination. I now explain how $dynamic2_{rws}$ runs into non-termination for a sample run for the above example grammar.

To illustrate the non-termination problem with $dynamic2_{rws}$, I use the number of recursive calls made to the symbols DXQML, JVL and CPM when favouring an alternative for rule X. The symbols DXQML, JVL and CPM are the hardest to derive symbols for the first, second and the third alternative of rule X respectively. Table 3.1 shows the scores of the symbols DXQML, JVL and CPM at various points during sentence generation for a run that didn't terminate. In the table, each block of lines indicates a recursive phase. The first line in a block marks the beginning of a recursive phase when DXQML's and JVL's score are relatively lower than the CPM's. This means, when $d<D$, the $dynamic2_{rws}$'s favouritism will continue to invoke either DXQML or JVL until their scores reach parity with CPM's

	DXQML		JVL		CPM	
depth (d)	exited/entered	score	exited/entered	score	exited/entered	score
75	2/4	0.5	3/10	0.3	1/6	0.8333
116	2/12	0.8333	3/19	0.8421	1/6	0.8333
			...			
73	10/12	0.166	13/20	0.35	2/8	0.75
229	10/41	0.7560	13/52	0.75	2/8	0.75
			...			
354	59/109	0.4587	79/146	0.4589	5/11	0.5454
482	59/130	0.5461	79/174	0.5461	5/11	0.5454
266	94/131	0.2824	129/175	0.2628	6/12	0.5
423♯	94/155	0.3935	129/213	0.3943	6/12	0.5
308	116/157	0.2611	160/215	0.2558	7/13	0.4615
651♯	116/212	0.4528	160/293	0.4539	7/13	0.4615
563	131/213	0.3849	182/294	0.3809	8/14	0.4285
670	131/231	0.4329	182/319	0.4294	8/14	0.4285
279	194/233	0.1673	274/320	0.1437	9/15	0.4
862	194/324	0.4012	274/459	0.4030	9/15	0.4
457	260/325	0.1999	369/459	0.1960	10/16	0.375
978†	260/409	0.3643	369/580	0.3637	10/16	0.375

† The recursion limit of the Python stack was reached.
♯ Weighted selection was applied.

Table 3.1: Table shows the scores of the relevant symbols DXQML, JVL and CPM for the alternatives of rule X for a sentence that didn't terminate. For a given symbol s, it's score is calculated as: (1-($s.exited/s.entered$)).

score. The second line in a block marks the end of a recursive phase when the scores of the symbols – DXQML, JVL, and CPM – reach parity. For illustration purposes, I only show a subset of the recursive phases from the sentence generation. The first recursive phase and the last 5 recursive phases are shown in the table.

As the sentence generation progresses, the number of recursive calls needed for DXQML's and JVL's score to reach parity with CPM's score steadily increases. At the beginning of the sentence generation, the recursive phase starts at d=73, and the scores reach parity

at d=116 after 8 and 9 recursive calls to `DXQML` and `JVL` respectively. Towards the end of the sentence generation, for the penultimate recursive phase, the recursive phase starts at d=279, and the scores reach parity at d=862 after 91 and 139 recursive calls to `DXQML` and `JVL` respectively. In the final recursive phase, which starts at d=457, the scores are too far apart, and as the sentence generation tries to get the `DXQML`'s and `JVL`'s score to reach parity with `CPM`'s, it runs out of stack. Just before the recursion limit was reached, the (number of exited over entered) for some of the other symbols that were invoked during the sentence generation run were: 205/323 for `K`, 205/323 for `EZ`, 209/329 for `PFDMK`, and 370/581 for `X`. The output of sentence generation run is available for download from: `https://figshare.com/s/a5f2f2e247c7d1ff0ed7`.

3.7.3.2 Summary

Although *dynamic2*$_{rws}$'s approach of occasionally not picking the low scoring alternatives mitigates non-termination, there are still a tiny number of cases where it doesn't terminate. Given a grammar, if there exists a subset of alternatives that is recursive, and if the only alternative that can lead to a successful derivation has a higher score, then the sentence generation runs into non-termination. A possible solution to reduce non-termination is to pick alternatives that guarantee termination when applying favouritism. I now explore this idea in the next section.

3.7.4 The *dynamic3* Backend

My next approach to mitigate non-termination involves a semi-deterministic favouring of alternatives that are more likely to terminate. Given a grammar, if a rule only contains alternatives with non-terminals, then I have no way of statically knowing which one is more likely to terminate. Therefore, I have had to devise an approach that allows me to uncover alternatives with a better chance of guaranteeing termination.

My approach involves noting those alternatives that are likely to guarantee termination during the course of the sentence generation (i.e. it is not determined statically). An alternative is likely to guarantee termination if each of its symbols derives to a finite depth. An alternative is of finite depth if: it contains only terminals; or it references non-terminals that themselves are of finite depth. If a rule has one or more alternatives of finite depth, then one of those is non-deterministically picked and recorded as the rule's alternative of finite length. When sentence generation recurses beyond the threshold depth D, and favouritism applies, I pick alternatives of finite depth if one is available. When a rule is entered, if it contains an alternative with finite depth, it is always picked. Clearly, there will be rules for whom an alternative of finite depth hasn't been uncovered yet. In such cases, I wait until the sentence generator revisits the rule, in the hope that

Algorithm 9 Algorithm for the *dynamic3* backend

```
1: function START(G, D)
2:     for A in N do
3:         R(A).finite_depth = NONE
4:     end for
5:     sen ← [ ]
6:     GENERATE(G, R(S), sen, d=0, D)
7:     return join(sen)
8: end function

9: function GENERATE(G, rule, sen, d, D)
10:    d ← d + 1
11:    if d > D then
12:        if rule.finite_depth ≠ NONE then
13:            alt ← rule.finite_depth
14:        else
15:            alt_fd ←CALC-ALT-FINITE-DEPTH(G, rule)
16:            if alt_fd ≠ NONE then
17:                rule.finite_depth ← alt_fd
18:                alt ← alt_fd
19:            else
20:                alt ← rand(rule.alts)
21:            end if
22:        end if
23:    else
24:        alt ← rand(rule.alts)
25:    end if
26:    for sym in alt do
27:        if sym ∈ N then                    ▷ If sym is a non-terminal
28:            GENERATE(G, R(sym), sen, d+1, D)
29:        else
30:            append(sen, sym)
31:        end if
32:    end for
33:    d ← d − 1
34: end function
```

there is sufficient information to make a decision. This way I continue to find alternatives of finite depth, and favour them as the sentence generation progresses.

Algorithm 9 describes the *dynamic3* backend. The function START is initialised with

Algorithm 10 Algorithm to determine an alternative of finite depth for a rule

```
1: function CALC-ALT-FINITE-DEPTH(G, rule)
2:     for alt in rule.alts do
3:         fd ← TRUE
4:         for sym in alt do
5:             if sym ∈ N then
6:                 if R(sym).finite_depth = NONE then
7:                     fd ← FALSE
8:                     break
9:                 end if
10:            end if
11:        end for
12:        if fd then
13:            return alt
14:        end if
15:    end for
16:    return NONE
17: end function
```

a user-defined grammar G and the threshold depth D beyond which alternatives that derive to a finite depth are favoured. For each rule *rule*, the attribute (*rule.finite_depth*) records the alternative with finite depth, and is initialised to NONE (lines 2–4). The function GENERATE is invoked to start sentence generation from the start rule $\mathcal{R}(S)$ of the grammar (line 6). The current recursion depth d is set to 0 at the start of sentence generation.

Given a rule *rule*, the function GENERATE continues to pick alternatives randomly (line 24). When $d{>}D$, favouritism applies, and alternatives with finite depth are favoured (lines 12 and 13). If *rule* does not have an alternative with finite depth yet, then each of its alternatives is examined for finite depth by invoking the function CALC-ALT-FINITE-DEPTH (line 15). If the rule *rule* contains an alternative of finite depth, then the alternative found alt_{fd} becomes the favoured alternative for rule *rule*, and alt_{fd} is assigned to *rule.finite_depth* (lines 16–18). In case none of rule *rule*'s alternatives is of finite depth, then one of its alternative is randomly picked (line 20).

Algorithm 10 describes the heuristic to determine if there exists an alternative with finite depth for a given rule. The function CALC-ALT-FINITE-DEPTH is initialised with grammar G and the rule *rule* that is being explored for an alternative of finite depth. For the given rule *rule*, each of its alternatives is examined. *fd* tracks if the current alternative is of finite depth or otherwise; *fd* is initialised to TRUE. The alternative *alt* is considered to be of finite depth, if each of its symbols is of finite depth (lines 4–11). If alternative *alt* is of finite depth, it is returned (line 13).

dynamic3's semi-deterministic approach to favour alternatives that guarantee termination significantly reduces non-termination. In a small experiment[10], where 5603 grammars were evaluated, whereas in the case of $dynamic2_{rws}$ 24 grammars did not terminate, in the case of *dynamic3* only one of the mutated grammar[11] did not terminate. My grammar corpus for the small experiment contained 3300 Boltzmann (see Section 4.1) and 2303 mutated grammars (see Section 4.2). *dynamic3* was run with D=16 for the Boltzmann grammars and D=18 for the mutated grammars. The value of D for each grammar set were the best performing values for the *dynamic3* backend from the fine dimensioning experiment (see Section 5.11). I now explain how *dynamic3* runs into non-termination for that one instance of the grammar.

3.7.4.1 Non-termination in *dynamic3*– an Example

For the mutated "Java" grammar that exhibited non-termination, of its 152 rules, *dynamic3* uncovered alternatives of finite depth for only 8 of them for a run that didn't terminate. The mutated grammar runs into non-termination as one of its core non-terminal that is referenced at multiple places within the grammar has become much harder to derive. The subset of the original Java grammar that is relevant to the mutation is shown below:

```
name : qualified_name | simple_name
qualified_name : name DOT_TK identifier
simple_name : identifier
identifier : ID_TK
```

The mutated grammar was generated by mutating the only symbol in the second alternative of the '`name`' rule in the above grammar. In the second alternative, the symbol '`simple_name`' was replaced by symbol '`if_then_else_statement`'. The subset of the mutated Java grammar showing the mutation to the second alternative of the '`name`'s rule is as follows:

```
name : qualified_name | if_then_else_statement
qualified_name : name DOT_TK identifier
if_then_else_statement :
  IF_TK OP_TK expression CP_TK statement_nsi ELSE_TK statement
expression : assignment_expression
statement_nsi : for_statement_nsi | ... | if_then_else_statement_nsi
statement : for_statement | ... | if_then_else_statement
```

[10]Results can be downloaded from: `https://figshare.com/s/885fcd7b2fcc4ad48019`.

[11]https://github.com/nvasudevan/experiment/blob/master/grammars/mutlang/acc/mutate/Java/Java.0_96.acc

Whereas in the original Java grammar, the non-terminal `name` is derivable in three steps (`name` ⇒ `simple_name` ⇒ `identifier` ⇒ `ID_TK`), in the case of the mutated grammar, this is lot harder. The second alternative now references symbol `if_then_else_statement` that in turn references three other non-terminals: `expression`, `statement_nsi` and `statement`. The non-terminals referenced form the core part of the subsets related to the expressions and the statements within the grammar. These subsets are deeply nested, and so *dynamic3* struggles to uncover alternatives of finite depth for this grammar.

One possible solution to mitigate such rare cases of non-termination is to calculate alternatives of finite depth prior to sentence generation and apply them when favouring alternatives.

3.7.5 The *dynamic4* Backend

My third and final approach to mitigate non-termination relies on a deterministic selection of 'finite depth' alternatives when applying favouritism. Whereas in *dynamic3* alternatives of finite depth were calculated during the course of the sentence generation, in *dynamic4*, the alternatives of finite depth are statically calculated prior to sentence generation. Stated differently, *dynamic4*'s heuristic to calculate alternative of finite depth is essentially equivalent to *dynamic3*, when run for long enough.

A fixed-point heuristic is used to calculate the sequence of alternatives that collectively derive to a finite depth for a given grammar. Given a grammar, I iterate over its rules, and for each rule, I check if any of its alternatives derive to a finite depth. If a rule contains such an alternative, then that becomes the favoured alternative for that rule. Every time an alternative of finite depth is found, then that aids in uncovering even more alternatives of finite depth. This process continues until the sequence of alternative of finite depth has been uncovered for all of the rules in the grammar.

Algorithm 11 describes the *dynamic4* backend. The function START is initialised in a similar way to the *dynamic3* backend but with one exception: the function CALC-FINITE-DEPTH is invoked to calculate the alternative with finite depth for each rule (line 5).

Given a rule *rule*, the function GENERATE continues to pick alternatives randomly (line 15). When the heuristic has recursed beyond the threshold depth D, predetermined alternatives that guarantee termination are picked (line 13).

Algorithm 12 describes the heuristic to calculate the alternative of finite depth for each rule for a given grammar. The function CALC-FINITE-DEPTH is initialised with the user defined grammar G. A copy of the grammar G's non-terminals whose rules are to be explored for finite depth is noted in N_c (line 2).

For a given grammar G, alternatives of finite depth are uncovered iteratively. In each iteration, for each grammar rule *rule*, function CALC-ALT-FINITE-DEPTH (see Algo-

Algorithm 11 Algorithm for the *dynamic4* backend

```
1: function START(G, D)
2:     for A in N do
3:         R(A).finite_depth = NONE
4:     end for
5:     CALC-FINITE-DEPTH(G)
6:     sen ← [ ]
7:     GENERATE(G, R(S), sen, d=0, D)
8:     return join(sen)
9: end function

10: function GENERATE(G, rule, sen, d, D)
11:     d ← d + 1
12:     if d > D then
13:         alt ← rule.finite_depth
14:     else
15:         alt ← rand(rule.alts)
16:     end if
17:     for sym in alt do
18:         if sym ∈ N then
19:             GENERATE(G, R(sym), sen, d + 1, D)
20:         else
21:             append(sen, sym)
22:         end if
23:     end for
24:     d ← d − 1
25: end function
```

rithm 10) is invoked to determine if there exists an alternative with finite depth. On finding an alternative *alt* with finite depth, it is assigned to *rule.finite_depth* (line 9). The non-terminal associated with the rule *rule* is removed from N_c, and the rule *rule* is explored no further (lines 10 and 11). The iterative search continues until an alternative with finite depth has been found for every rule (i. e. a path has been found from a given alternative that collectively are of finite depth).

In a small experiment[12] performed, where 5603 grammars were evaluated, none of the grammars ran into the non-termination problem. My grammar corpus for the small experiment contained 3300 Boltzmann (see Section 4.1) and 2303 mutated grammars (see Section 4.2). *dynamic4* was run with D=14 for the Boltzmann grammars and D=26 for the mutated grammars. The value of D for each grammar set were the best performing

[12]Results can be downloaded from: `https://figshare.com/s/8a33288ec46cede9d729`.

Algorithm 12 Algorithm to calculate alternatives with finite depth

```
1: function CALC-FINITE-DEPTH(G)
2:     N_c ← copy(N)
3:     while len(N_c) > 0 do
4:         for A in N do
5:             rule ← R(A)
6:             if rule.finite_depth == NONE then
7:                 alt ← CALC-ALT-FINITE-DEPTH(G, rule)
8:                 if alt ≠ NONE then
9:                     rule.finite_depth ← alt
10:                    remove(N_c, A)
11:                    break
12:                end if
13:            end if
14:        end for
15:    end while
16: end function
```

values for the *dynamic4* backend from the fine dimensioning experiment (see Section 5.11).

3.8 Summary

In this chapter, I presented my ambiguity detection tool, *SinBAD*. *SinBAD* houses several different backends, ranging from pure random to heuristic driven, that each uses a non-deterministic approach to detect ambiguities in grammars. I started with a simple backend, *purerandom*. Although *purerandom* uncovers close to half the number of ambiguities on my grammar corpus, it runs into non-termination on 87% of the grammars. To tackle non-termination, I created various heuristic driven backends. Each of my heuristic driven backends uses a specific approach to tackle non-termination whilst still continuing to perform well in uncovering ambiguities.

In *dynamic1* and *dynamic2*, I showed how using a simple scoring mechanism to favour alternatives that encourage termination significantly mitigates non-termination. However, the deterministic scoring mechanism occasionally leads to recursive cycles. In $dynamic2_{rws}$, I applied a roulette wheel selection of scores. The weighted application of scores mitigated non-termination even further although not fully. In *dynamic3*, I explored a semi-deterministic selection of alternatives that guarantee termination, which failed to terminate in just one case. Finally, in *dynamic4*, I showed that by using predetermined alternatives that guarantee termination, there were no cases of non-termination.

To evaluate my backends properly, I need a larger grammar corpus. Since generating such

a corpus by hand is not feasible, I devised two novel ways of generating grammars. In the following chapter, I present my grammar generation approaches.

Chapter 4

Grammar Generation

In this chapter I introduce new techniques for evaluating the effectiveness of ambiguity detection tools. I believe that evaluating such tools requires much larger input corpuses than previously used: mine contains over 20,000 grammars of various types. In order to generate such a large corpus, one cannot rely on hand-written grammars. I therefore created two large corpuses of random grammars: Boltzmann sampled and mutated PL grammars. The first class of random grammars is generated using Boltzmann sampling, an approach which provides some statistical guarantees about the randomness of the resulting generators. The second class of random grammars is generated through mutation of existing grammars.

This chapter comes in two parts. The first part introduces Boltzmann sampling and then presents the first Boltzmann sampler for CFGs. The second part covers the grammar mutation technique, wherein I present five different types of mutation operators for generating random PL grammars.

4.1 Boltzmann Sampled Grammars

Boltzmann sampling is a framework for random generation of combinatorial structures (see [11] for further details). The basic idea is to give the sampler a class specification of a combinatorial structure and a value to control the size of the generated objects. For a given class C, and size n, the sampler provides approximate-size uniform random generation—objects are generated with approximate size n$\pm\epsilon$, where ϵ is a fixed tolerance, but objects of the same size occur with equal probability. This allows the sampler to generate large objects in linear time. In this section I provide the first Boltzmann sampler for grammars.

Figure 4.1 describes my framework for generating Boltzmann sampled grammars. For a given grammar size (where $|G|$ = number of rules), the *specification generator* generates a Boltzmann class specification. A class specification is a set of definitions that essentially represents the 'style of grammars' that one wishes to generate. For the purposes of my

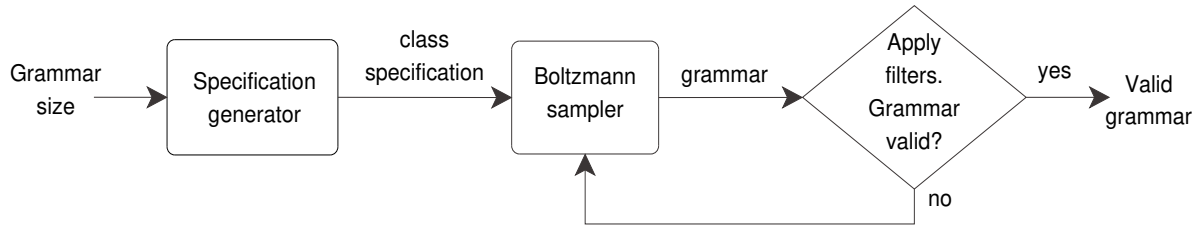

Figure 4.1: Framework for generating Boltzmann grammars.

experiment, I am mostly interested in PL-like grammars, and so by style of grammars, I simply mean 'measurable' aspects of grammars, such as the number of rules a grammar contains, the percentage of empty alternatives and so on. The generated specification is then fed to the *Boltzmann sampler* to generate a grammar. Although Boltzmann samplers are quite efficient at generating randomised tree structures, they are unaware of the semantics of the objects that they generate. Inevitably, my Boltzmann sampler generates PL grammars that are semantically non-sensical. For instance, the generated grammars may contain empty rules or may contain rules with far too many alternatives than one might typically expect in PL grammars. I am therefore, forced to apply post-filters to restrict the grammars generated to those that resemble PL grammars. Such filters are needed if one wishes to generate PL-like grammars. The various parts of my Boltzmann framework are described in the subsequent subsections.

4.1.1 Specification Generator

The primary input to the specification generator is the grammar size. PL grammars come in varying sizes: from JSON (with 6 rules) to Java (with over 200 rules). Although my Boltzmann sampler is quick to generate grammars of various sizes, generating a valid grammar is (roughly) proportional to its size. Whereas generating a valid grammar for lower sizes ($|G| < 10$) took a minute or so, generating a valid grammar for higher sizes ($|G| > 50$) took easily up to an hour. Therefore, for my experiment, I restricted grammar generation to sizes ranging from 10 to 75 inclusive.

4.1.2 Class Specification

A Boltzmann sampler class specification is a grammar containing a set of productions. A production is of the form: `A:` ⟨`rhs`⟩, where `A` is the name of the class being defined and ⟨`rhs`⟩ is a set of definitions. A definition is of the form `DefX Y`, where `DefX` denotes a constructor and `Y` is either: a reference to a definition (if a definition `Y` exists) or a literal otherwise.

Since, as far as I am aware, this is the first time that Boltzmann sampling has been used to generate grammars, I am forced to create a class specification myself. Determining a good

```
Cfg = Cfg Rule ... Rule
Rule = SingleAlt Alt | RuleAlts1 Rule Alt
Alt = EmptyAltSyms | SingleAltSyms1 Symbol | AltSyms1 Alt Symbol
Symbol = NonTerm NonTerm | Term Term
NonTerm = NonTerm1 | NonTerm2 | ... | NonTermN
Term = Term1 | Term2 | ... | TermN
```

Figure 4.2: Tree specification for generating grammars.

class specification is arguably the hardest part of Boltzmann sampling, and is complicated by the fact that grammars do not have a single, obvious specification. Furthermore, since grammars are unbounded in size, I necessarily have to restrict the size of those generated to make using them practical. This immediately led me to a difficult question: what style of grammars do I want? In reality, I am most interested in grammars which somewhat resemble PL grammars. I have therefore crafted my use of Boltzmann sampling to lead to grammars which roughly resemble real PLs. The resulting grammars are close to those that one might reasonably expect to see for PLs. While I do not claim that my specification is perfect, it is the result of careful, and considerable, experimentation.

My class specification is shown in Figure 4.2. Using [31] as a guiding principle, my specification is designed to give me control over three things: the number of empty alternatives, the number of alternatives per rule, and the number of symbols per alternative. `Cfg` denotes a context-free grammar, `Rule` a production rule, `Alt` a production alternative, and `Symbol` denotes either a non-terminal (a `NonTerm`) or a terminal (a `Term`) symbol. A CFG consists of 1 or more production rules (hence the references to multiple Rule definitions). `Rule` has two outcomes: it can either be called recursively to build a list of alternatives; or just build a list with single alternative. `Alt` has three choices: it can either be called recursively to build a sequence of symbols; or just build a sequence with one symbol (middle choice); or an empty string (`EmptyAltSyms`). The specification enforces equal numbers of `NonTerm`s and `Term`s in a grammar, the 1:1 ratio seeming to be a reasonable heuristic based on my observations of real grammars.

It is worth noting that even minor variations to the specification can lead to significantly differing "styles" of grammars being generated. For instance: replacing `SingleAlt Alt` by `EmptyAlt` would cause a much higher percentage of empty alternatives to be generated; or adding a choice `SingleAltSyms2 Symbol` to the `Alt` production would result in higher percentage of single symbol alternatives. Therefore, the design of the specification with its set of choices for each production is important in controlling the style of grammar that gets generated.

4.1.3 Boltzmann Sampler

Given a class specification, the Boltzmann sampler generates random grammars. A Boltzmann sampler is parameterised by two values that control the size of the generated objects: singular precision and value precision. To get an efficient sampler, these two values need to be set as low as possible [11]. However, the lower these values are, the greater the likelihood of large objects being generated. This is a problem, as "large" means rules would have more alternatives and symbols per alternative than one desires. The challenge, then, is to find values that generate large numbers of relevant grammars in reasonable time. After considerable experimentation, I settled on values of $1.0e^{-7}$ and $1.0\text{-}e^{-4}$ for the singular and value precisions respectively.

4.1.4 Filtering

My Boltzmann class specification gets me in the rough neighbourhood of PL grammars, but some obvious differences remain. Alternatives in PL grammars are typically short (number of symbols per alternative roughly varies between 0 and 5). The sampler struggled to generate grammars when I restricted the number of symbols per alternative to 5, so I relaxed this criterion. Approximately 15–20% of alternatives from each grammar generated by the sampler therefore have more than 5 symbols per alternative.

Similarly, the sampler tends to generate a much larger number of empty alternatives than are typical of PL grammars. Using Basten's PL grammar corpus as an example, the proportion of empty alternatives varied between 4% for Java to 19% (CSS). I therefore wrote a filter to remove all grammars that had a proportion of empty alternatives above 5%. Such filters are needed if one wishes to generate PL-like grammars.

In some cases, my sampler generated too many alternatives for a rule. PL grammars usually contain 3–5 alternatives per rule. I therefore wrote a filter which restricts the number of alternatives per rule to 5.

Because the sampler is unaware of the precise semantics of grammars, it can and does produce grammars which are nonsensical or trivially ambiguous. I apply additional filters. Grammars which contain unreachable symbols (i.e. symbols that can't be reached from the start rule) are excluded. Grammars containing non-terminating cycles of the form `A: B` and `B: A` are excluded as they consume no input and generate the empty language. Finally, grammars which contain alternatives with the same sequence of symbols (e.g. A: $X \mid X \mid ...$) are excluded as such grammars are trivially ambiguous.

4.2 Mutated Grammars

The second class of random grammars is generated by mutating existing PL grammars. The class of PL grammars is particularly interesting as I, like most others working in this field, am particularly interested in the ambiguity of PL-like grammars. There is an inevitable problem with generating PL-like grammars. There are only a handful of PL grammars and most PLs are written for approaches such as LR parsing that accept only unambiguous grammars. Basten hand-modified 20 PL grammars to be ambiguous [4] which I reuse in my suite for comparison purposes. However, one can easily, and inadvertently, create a solution which works well for such a small corpus but little beyond it. Thus, by generating a huge number of possibly ambiguous PL-like grammars, I hope to explore a much wider set of possibilities than is practical by hand.

Random grammar generators have one major problem from my perspective: even if they produce grammars in the general style of those used by PLs, it can be reasonably argued that they are never close enough. Of course, exactly what *is* close enough is impossible to pinpoint: it seems unlikely that any metric, or set of metrics, can reliably classify PL vs. non-PL grammars. Instead, I have little choice but to fall back on the intuitive notion that "I know one when I see one". This means that past work has struggled to understand how ambiguity affects PL-like grammars: one simply can't get hold of enough of them to perform adequate studies. The best attempt of which I am aware is the work of Basten, who took 4 unambiguous PL grammars and manually altered them to introduce ambiguity [6]. Manually altering grammars is tedious, hard to scale, and always open to the possibilities of unintentional human bias.

I have therefore devised a simple way of generating arbitrary numbers of 'PL-like' grammars with possible ambiguity. My approach to grammar mutation bears no relation to grammar evolution or grammar recovery. Instead, my basic tactic is inspired by Basten's manual modifications: I take a real (unambiguous) grammar for a PL and perform a single random alteration to a single rule. Although there are numerous possible mutations, I restrict myself to the following five, each of which is applied to a single rule:

Add empty alternative This is only possible if a rule does not already have an empty alternative.

Mutate symbol Randomly select a symbol from an alternative and change it. A non-terminal can be replaced by a terminal and vice versa.

Add symbol Randomly pick an alternative and add a symbol at a random place within it.

Delete symbol Randomly delete a symbol from an alternative. Only non-empty alternatives are considered.

Switch symbols Randomly pick an alternative and switch any two symbols. This is possible only with alternatives with two symbols or more.

My mutated grammars are therefore identical to a real PL grammar, with only a single change. This is the best way that I can imagine of solving the "I know it when I see it" problem. As we will see later, these simple mutations introduce a surprising number of ambiguities.

Algorithm 13 describes each of my mutation operator function. The definitions from Section 3.4 are re-used and some additional notations are defined. For a grammar G, G_{rules} refers to its rules. For $n<N$, the function $\mathsf{randinds}(0, N, n)$ returns a list of n distinct values picked randomly between 0 (including) and N (excluding). For a list l, index i, and value v, the function $\mathsf{insert}(l, i, v)$ inserts value v in l at index i. For a list l, the function $\mathsf{delete}(l, i)$ deletes the element at index i in l. Given a grammar, my mutation operators work on a copy of the grammar. For the purposes of this section, I denote G as the working copy of the grammar.

For the 'add empty alternative' mutation, for a given grammar, rules that do not have empty alternatives are first identified. From these rules, a rule is randomly selected, and an empty alternative is added.

For mutation of type 'add symbol', for a given grammar, a rule is randomly selected. From the selected rule, an alternative is randomly picked. From the selected alternative, a position is randomly picked and a randomly selected symbol from $N \cup T$ is inserted.

For mutation of type 'mutate symbol' and 'delete symbol', for a given grammar, a rule is randomly selected. From the selected rule, one of its non empty alternatives is randomly picked. For 'mutate symbol' mutation, a symbol is randomly picked from the selected alternative, and is replaced with a randomly selected symbol from $N \cup T$. For 'delete symbol' mutation, from the selected alternative, a symbol is randomly picked and deleted.

For mutation of type 'switch symbol', for a given grammar, rules containing alternatives with two or more symbols, are identified. From the identified rules, a rule is randomly picked, and one of its alternatives containing two or more symbols is randomly picked. From the identified alternative, pick two positions randomly and switch the symbols.

4.3 Summary

In this chapter, I presented two novel grammar generating techniques. I implemented the first Boltzmann sampler for CFGs. Using my Boltzmann sampler I then showed how a large corpus of random grammars of various sizes can be generated. I then presented my grammar mutation technique, wherein using five different types of mutation operators, a large corpus of PL grammars can be generated.

Algorithm 13 Algorithm describing the various mutation operators for PL grammars

```
1: function ADD-EMPTY-ALT(G)
2:     rules ← [ ]
3:     for rule ∈ G_rules do
4:         for alt ∈ rule.alts do
5:             if |alt| = 0 then
6:                 append(rules, rule)
7:                 break
8:             end if
9:         end for
10:    end for
11:    r ← rand(G_rules − rules)
12:    append(r.alts, [ ])
13: end function
```

```
1: function ADD-SYMBOL(G)
2:     rule ← rand(G_rules)
3:     alt ← rand(rule.alts)
4:     i ← randinds(0, |alt|, 1)
5:     tok ← rand(N ∪ T)
6:     insert(alt, i, tok)
7: end function
```

```
1: function MUTATE-SYMBOL(G)
2:     rule ← rand(G_rules)
3:     alts ← [ ]
4:     for alt ∈ rule.alts do
5:         if |alt| > 0 then
6:             append(alts, alt)
7:         end if
8:     end for
9:     alt ← rand(alts)
10:    i ← randinds(0, |alt|, 1)
11:    tok ← rand(N ∪ T)
12:    alt[i] = tok
13: end function
```

```
1: function DELETE-SYMBOL(G)
2:     rule ← rand(G_rules)
3:     alts ← [ ]
4:     for alt ∈ rule.alts do
5:         if |alt| > 0 then
6:             append(alts, alt)
7:         end if
8:     end for
9:     alt ← rand(alts)
10:    i ← randinds(0, |alt|, 1)
11:    delete(alt, i)
12: end function
```

```
1: function SWITCH-SYMBOLS(G)
2:     rules ← [ ]
3:     for rule ∈ G_rules do
4:         for alt ∈ rule.alts do
5:             if |alt| ≥ 2 then
6:                 append(rules, rule)
7:                 break
8:             end if
9:         end for
10:    end for
11:    r = rand(rules)
12:    alts ← [ ]
13:    for alt ∈ r.alts do
14:        if |alt| ≥ 2 then
15:            append(alts, alt)
16:        end if
17:    end for
18:    alt = rand(alts)
19:    i, j ← randinds(0, |alt|, 2)
20:    t = alt[i]
21:    alt[i] = alt[j]
22:    alt[j] = t
23: end function
```

To understand how well my dynamic backends work in detecting ambiguities, I perform a large-scale evaluation, comparing my backends with other extant tools. In the following chapter, I present my experimental evaluation of ambiguity detection tools.

Chapter 5

Dimensioning Experiments

The aim of this chapter is to uncover the best performing run-time option for each tool from my experimental suite. This chapter comes in four parts. The first part presents my experimental suite listing the various run-time options and the range of values that each tool supports. The details of my grammar collection used for various experiments is then listed. The second part presents a novel search-based implementation for exploring the solution space of a tool option. The third part covers the crude dimensioning experiment, wherein I apply my search-based implementation to each tool option to uncover promising regions in its solution space. The fourth part covers the fine dimensioning experiment, wherein for each tool option, the promising regions uncovered by the crude dimensioning experiment are exhaustively searched to uncover the best performing run-time value.

5.1 Experimental Suite

The objective of my experiments is to understand how well search-based approaches perform in detecting ambiguities. Since ambiguity is inherently undecidable, it is impossible to evaluate such a tool in an absolute sense. Instead, I evaluated my tool against three others: ACLA, AMBER, and AmbiDexter. Each tool takes a different approach: ACLA uses an approximation technique; AMBER uses an exhaustive search; AmbiDexter uses a hybrid approach where an approximation technique is applied to filter out unambiguous grammar subsets and on the resulting grammar an exhaustive search is used; and *SinBAD*'s backends use a random search-based approach.

All the tools except ACLA have run-time options which adjust the way they operate and thus affect which ambiguities they find. I believe the fairest comparison is between the tools at their best, and that I need to use the "best" run-time option values possible. However, discovering what the best options are by trying all possibilities on my full set of grammars, is prohibitively expensive[1]. Instead, I first perform a "dimensioning"

[1] A reasonable lower bound estimate is at least 3 core months

experiment on a subset of the grammars to determine good tool options.

The dimensioning experiment constitutes of two parts: crude and fine dimensioning. The crude dimensioning run explores the search space of a tool option as much as possible to get a rough idea of the solution landscape. This is then followed by the fine dimensioning run, where the solution space – containing potential good solutions, uncovered by the crude dimensioning run – is explored in finer detail. For the fine dimensioning run, the tools are run on a bigger (approximately twice the size) collection of grammars. Further for the dimensioning runs, the tools are run several times (approximately 30 and 15 times for the crude and fine dimensioning respectively) in exploring the solution space for each tool option. I do not claim that the option values discovered necessarily allow each tool to operate at its maximum potential; rather, I believe that they allow the tool to operate close enough to its maximum potential to make a comparison meaningful.

Using the run-time options and values determined from the fine dimensioning run, I then run the "main" experiment on a large set of grammars (approximately 5 times bigger) with each of the 4 tools. My main experiment explores a subset of the grammars from my corpus. There is a small chance that in evaluating my backends, I may have been biased in choosing my grammars for the main experiment. To check that the proportion of grammars discovered as ambiguities scales up, I run a validation experiment using only the best *SinBAD*'s backend on a much larger set of grammars (just over twice the size used for the main experiment). Note that for my experiments, I am not worried about the particular ambiguous fragments identified: I care only whether a tool uncovers ambiguity in a grammar or not.

My experimental setup is fully repeatable. The grammar generators, the grammar corpus I used, and the results obtained can be downloaded from my experimental suite:

`https://figshare.com/s/990138b4dec10691f3f0`

5.2 Grammar Collection

I evaluated the various tools on three different sets of grammars: Boltzmann sampled grammars, altered PL grammars, and mutated grammars. Boltzmann sampled grammars were described in Section 4.1. The altered PL grammars are taken from [7], where Pascal, SQL, Java, and C grammars were manually modified to produce 5 ambiguous variations of each. The mutated grammars were described in Section 4.2. Table 5.1 shows the size (number of rules) of the grammar sets used in each experiment. For the Boltzmann sampled grammars, each size (10–75) is proportionately represented: for the dimensioning experiment, 10 grammars of each size were used whereas for the main experiment, 50 grammars of each size are used.

	Dimensioning			
	Crude	Fine	Main	Validation
Boltzmann	330	660	3300	6600
Altered PL	20	20	20	-
Mutated	125	500	2303	5941
Total	475	1180	5623	12541

Table 5.1: The size of the grammar sets used for the crude and fine dimensioning, main and validation experiments.

For the mutated grammars, each mutation category (add empty alternative, mutate symbol, add symbol, delete symbol, and switch symbols) is proportionately represented for the dimensioning experiment. I use 20 grammars from the mutation category. However, for the main experiment, the "add empty alternative" mutation is less represented than the others as there are only so many possible rules where one can add an empty alternative. For instance, the SQL grammar has 29 rules with no empty alternatives, thus allowing us to generate at most 29 possible mutations. Therefore, for the add empty alternative, I could only generate a total of 303 mutations (Pascal: 69, SQL: 29, Java: 100, C: 64, and CSS: 41). For the remaining mutation categories, I use 500 grammars from each category (100 from each language).

5.3 Hardware

The crude dimensioning was performed on an i7-2600S CPU 2.80GHz machine with 4 real cores and 8GiB memory; hyperthreading was turned off. The fine dimensioning, the main and the validation experiments were performed on an Intel i7-4790K CPU 4.0GHz machine with 4 real cores and 24GiB memory; turbo mode hyperthreading was turned off. For all my experiments, I used 3 cores per machine. The experiments took around 68 core-days in total, broken down into: 40 core days for the crude dimensioning run; 15 core-days for the fine dimensioning run; 10 core-days for the main experiment; and 3 core-days for the validation experiment.

5.4 Tools and Options

My tools come with options, and several of those options support a broad range of values. Through the dimensioning experiments, I wish to uncover, for each tool, and for each

Tool	Option	Values
AMBER	Search by length	0 to ∞
	Search by example	0 to N
	Ellipsis	Yes / No
AmbiDexter	Length	0 to N
	Incremental length	0 to ∞
	Filter	None, LR0, SLR1, LALR1, LR1
$dynamic_n$	Depth	0 to ∞
$dynamic2_{rws}$	Depth	0 to ∞
	Weight to unfavour alternatives	0 to 1

Table 5.2: Options and the range of run-time values supported by AMBER, AmbiDexter, $dynamic_n$ and the $dynamic2_{rws}$ backends.

grammar collection, the best performing option(s) and the best performing run-time value for that option. I now describe the various options that each tool supports.

ACLA has no options, so does not need to be considered further. For the rest of the tools, the options and the range of values that each tool supports is listed in Table 5.2.

AMBER can search in two modes: by length, where strings up to a fixed length N are checked; or by example, where the search is limited by the number of example strings (of unrestricted length) are checked. The 'ellipsis' option considers non-terminals also as tokens during sentence generation, which increases the chances of finding long ambiguous strings. Each of the main options can be run with ellipsis set to Yes/No.

AmbiDexter comes with two main options: 'search by fixed length', where strings of up to a fixed length N are checked; or by 'search by incremental length', where a search is performed iteratively starting with an initial string length, and at each iteration the string length is incremented by 1. AmbiDexter also supports the use of "filter" that prune out subsets of the grammar proven to be unambiguous. The filter can be applied as a sub-option to the two main options. Generating a filtered version of a grammar is included in the time limit.

SinBAD's dynamic backends are parameterised by a depth option D, a threshold depth beyond which the heuristic starts to favour certain alternatives to terminate sentence generation. A few of the dynamic backends accept an additional option, a probabilistic weight W to unpick low cost alternatives whilst still favouring. My dynamic backends – *dynamic1*, *dynamic2* and $dynamic2_{rws}$– in certain cases, didn't unwind from deep recursion and exited. In such cases, I re-ran my backends (in such cases, the normal time limit still applied).

As shown in Table 5.2, the range of values supported by several of the tool options is broad and finding a reasonably good run-time value is tricky. In [40], I used my personal experience of the tools in question to hand-pick a subset of values to test, using the best for the main experiment. The drawback with such an approach is that one can easily be biased in picking the initial set of values. Search-based [28] techniques have been proven to be effective in finding 'sufficiently good' solutions for problems whose search spaces are too large to find a perfect solution. For dimensioning, where I am interested in uncovering a reasonably good value for a tool option for a given grammar corpus, I apply search-based techniques to optimise each tool option. Search-based techniques were explained in detail in Chapter 3 (Section 3.2). I now describe my search-based implementation for uncovering a good run-time value for my tool options.

5.5 Search-based Techniques

To formulate a given problem as a search-based problem, three key ingredients need to be defined [28]. The ingredients are: the representation of the candidate solution; the move operator that manipulates a solution to generate a neighbour; and the fitness function to measure the quality of a solution. I now provide an overview of each of these ingredients.

5.5.1 Choice of Representation

For a given problem, the representation of a candidate solution defines the shape of the search space. Since each point in the search space corresponds to a feasible solution, picking a suitable representation for a candidate solution is crucial. The most common forms of representing a candidate solution include: numbers (natural numbers or floating point), permutations, and binary strings. The choice of representation of a solution is specific to the problem at hand.

5.5.2 Fitness Function

A search-based technique requires a mechanism by which the quality of the candidate solutions can be measured. The fitness function is the characterisation of what is considered to be a good solution. A fitness function should sufficiently distinguish a good solution from a poor one so as to help guide the search towards good quality solutions. Fortunately, problems in engineering come with a rich set of metrics that naturally form good candidates for fitness functions. The fitness function usually aims to maximise (or minimise) depending on the problem at hand.

5.5.3 Move Operator

For a search-based technique to be effective, picking a 'good' neighbour for a given solution is crucial. Just what constitutes a good neighbour is specific to the problem at hand. Typically, a neighbour is only a small step away for a given solution. To direct the search towards good quality solutions, search-based techniques rely on a 'move' operator to move the search from one solution to another. Given a solution, a move can either produce: a 'near neighbour' by making the smallest possible change to the given solution, and such a solution closely resembles the given solution; or a 'distant neighbour' by mutating the given solution that may not resemble the given solution. The set of available move operations that can be applied to a solution depends on the choice of the representation.

5.6 Formulating Tool Options as a Search Problem

For the dimensioning experiment, I wish to uncover the best performing run-time value for each of the tool options: AMBER length and examples, AmbiDexter length, the $dynamic_n$ backends, and the $dynamic2_{rws}$ backends. Before search-based techniques can be applied for dimensioning my tool options, the three key ingredients outlined in the Section 5.5 need to be defined.

5.6.1 Solution Representation

Since the format of the run-time value (see Table 5.2) that each tool option accepts is different, the choice of representation depends on the tool option that is being optimised.

The tool options for AMBER is either the string length or the number of examples to be explored. For the length option, since the values range from 0 to ∞ and each value is a potential solution, I chose a natural number representation. Similarly for the examples option, the values range from 0 to N with each value being a potential solution, so I choose natural number representation for the examples option too. The sub-option ellipsis for length or examples option is represented as a boolean value (Yes/No).

For AmbiDexter, the length option accepts an integer value between 0 and ∞. So a natural number representation is chosen. The sub-option filter for length is represented as a string.

For $dynamic_n$ backends, for the depth option, the values can range from 0 to ∞ and each value is a potential solution. So depth is represented as a natural number. The $dynamic2_{rws}$ backend accepts a second option, weight W, to pick an alternative other than the low cost alternatives. The value of weight ranges from 0 to 1, where each floating point is a potential solution. So weight is represented as floating point number.

Tool	Option	Values
AMBER	Search by length	0, 10, 20, 50, 100, 500
	Search by example	10^4, 10^6, 10^{10}
	Ellipsis	Yes / No
AmbiDexter	Length	0, 10, 20, 50, 100, 500
	Incremental length	0, 10, 50, 100
	Filter	None, LR0, SLR1, LALR1, LR1
$dynamic_n$	Depth	0, 10, 25, 50
$dynamic2_{rws}$	Depth	0, 10, 25, 50
	Weight to unfavour alternatives	0.01, 0.05, 0.1

Table 5.3: Options and the set of run-time values invoked for AMBER, AmbiDexter, $dynamic_n$ and the $dynamic2_{rws}$ backends.

5.6.2 Fitness Function

For the purposes of my experiment, I am interested in maximising the number of ambiguities detected on a given grammar corpus. Thus the fitness function is simply defined to be the total number of ambiguities detected for a given grammar corpus.

5.6.3 Move Operator

My grammar corpus contains grammars of various sizes and types, and whilst a run-time value v for a tool option may be best performing for certain grammars, for others, it might not be so. In [40], I regularly found that even a small change to the run-time value of a tool option can cause it to find or miss ambiguities. Such subtle variations in the run-time behaviour of a tool can cause tiny fluctuations in the fitness values for even the tiniest of changes in the solution space. Therefore, in designing the move operator for my tool options, I first had to understand their solution landscape for each grammar collection. To get an idea of the solution landscape for each tool option, I performed a small experiment. I ran the ambiguity detection tools from my experimental suite for various points spread across the solution landscape. Table 5.3 shows for each tool option the solutions there were invoked for.

My experiment showed that for tool options – AMBER (length and examples) with ellipsis set or otherwise and AmbiDexter (length and incremental length) with filter set or otherwise – the fitness distribution was asymptotic. For $dynamic_n$ backends, the fitness distribution resembled a bell curve, and the $dynamic2_{rws}$ backend had a hill shaped distribution. Since the tool options show different fitness distributions, one has to be careful

in choosing the size of the jump when selecting a neighbour.

My approach for picking a neighbour involves using an adaptive step function. The step function is seeded with two built-in step sizes – small and big – and complemented by a heuristic that acts as a control switch to pick between the two. My heuristic to pick a step size is based on the variation in the fitness values of the last $\mathcal{N}_{\text{nei}}$ solutions. If the variation is negligible (i.e below a certain threshold σ_{nei}), then a big step size is applied, in the hope of exploring the search space beyond the current neighbourhood. On the other hand, if the variation is fairly substantial (i.e. above a certain threshold), then a small step size is applied, with the aim being to explore the current neighbourhood even further to find a better solution.

5.6.4 Local Maximum

A search is said to have found a local maximum when changes to the current solution in the neighbourhood of candidate solutions offers no further improvement. Since my tool options exhibit tiny fluctuations in fitness values for adjacent points in the solution space, to determine local maximum, I do not stop at the first dip in the fitness value but check the last $\mathcal{N}_{\text{lmx}}$ fitness values to be reasonably sure that the fitness can't be improved any further.

5.7 Choosing a Search-based Technique

Since, there are a number of search-based techniques that can be applied to a given problem, I am left with a difficult question: which technique should I use to evaluate my tools? Furthermore, my choice of a technique is complicated by two additional factors. First, as far as I am aware, this is the first time that search-based techniques have been applied to evaluate ambiguity detection tools. Second, I need to evaluate not one, but four tools, and each of those tools comes with various configurable options. To pick a search-based technique, one needs to understand the solution landscape of the problem domain. I therefore ran my ambiguity detection tools for various values spread across the solution space.

Since there is no previous baseline to use as a reference, I have no choice but to opt for a search technique that is simple to implement and one that can find a good enough solution for the problem at hand. A simple search technique, and one that is generally considered to be a good choice for first application to a problem domain, is hill climbing [20]. I chose the hill climbing technique for dimensioning my tool options. I first provide an overview of the hill climbing technique and then discuss its implementation.

Algorithm 14 Hill climbing algorithm

```
1: s ← a randomly chosen initial solution from search space
2: repeat
3:     s′ ← generate a neighbour from s
4:     if (fitness of s′) > (fitness of s) then
5:         s ← s′
6:     end if
7: until stopping condition is reached
8: return s
```

5.7.1 Hill Climbing

The simplest form of a search-based technique that utilises fitness information to guide the search towards better solutions is hill climbing. In hill climbing, a point is selected from the search space at random, and the search is initiated. A candidate solution that is in the neighbourhood of the original, that is, a solution that is a small mutation away from the original is examined. If the neighbour candidate solution has an improved fitness, then the search moves to that new solution. The neighbourhood of the new candidate solution is explored for a better solution, and so on, until a solution is found whose neighbourhood does not offer any further improvement. Intuitively, a 'hill' has been climbed in the search landscape close to the randomly chosen start point. Algorithm 14 shows the pseudo code for the hill climbing technique. The hill located by the hill climbing technique may be a local maxima, and may be far poorer than a global maxima in the search landscape.

The implementation of the hill climbing technique to optimise the run-time options for each tool from Table 5.2 now follows.

5.8 Implementation of Hill Climbing

Several of my tools for dimensioning accept just one option. Only in the case of the $dynamic2_{rws}$ backend, does the tool accept two options depth D and weight W. I have therefore split my hill climbing implementation into two parts: a first implementation that optimises a single option; and a second implementation for the $dynamic2_{rws}$ backend to optimise its twin parameters D and W.

In the sections that follow, I first define some notations, and then present my hill climbing implementation for optimising a single run-time option.

5.8.1 Definitions

For a given list l containing integers, $\mathsf{sdev}(l)$ denotes the standard deviation of items from l, and $\mathsf{max}(l)$ denotes the maximum value from l. For a list of items l, $\mathsf{first}(l, n)$ denotes the first n items from l, and $\mathsf{last}(l, n)$ denotes the last n items from l. To append an item v to a list l, $\mathsf{append}(l, v)$ is used.

For a given solution s, $\mathsf{smallstep}(s)$ and $\mathsf{bigstep}(s)$ denote functions that return a near neighbour and a distant neighbour from s respectively. $\mathsf{fitness}(s)$ denotes the fitness function that returns the number of ambiguities found for solution s. For a given list of pairs p, where each pair is of the form (s, f) with s referring to a solution and f referring to the fitness at s, $\mathsf{getfit}(p)$ returns the list of fitness values from p. For a given tool t that accepts option t_0, t_1 and so on, notation $\mathsf{runtool}(t, (t_0,v_0), (t_1,v_1), \cdots)$ denotes the invocation of tool t with its option t_0 set to v_0, t_1 set to v_1 and so on respectively.

5.8.2 Hill Climbing - Single Option

The tools that I wish to optimise for a single option are: for AMBER, the length and the number of examples option; for AmbiDexter the length option; and for the $dynamic_n$ backends, the depth D option. For a given tool and its option that I wish to optimise, the hill climbing search is started from the initial solution provided. The search continues to explore the solution space iteratively by sampling solutions in the neighbourhood until no better solution can be found. On reaching local maxima, the search returns the solutions with the highest fitness. In cases, where the main option also accepts a sub-option, the hill climbing implementation accepts additional parameters for setting a secondary option and its value. The value for the sub-option is passed as is to the tool.

Algorithm 15 shows the implementation of the hill climbing technique to optimise a single option. The function HC-TOOL-OPTION accepts the following parameters: t is the tool to run; t_{opt} is the run-time option to optimise for tool t with s_0 as the initial solution for t_{opt}; t_{misc} is the additional option that tool t accepts and is set to value v; $\mathcal{N}_{\text{nei}}$ is a constant that refers to the number of fitness values whose standard deviation controls neighbour selection; σ_{nei} is the threshold standard deviation constant that applies for neighbour selection; $\mathcal{N}_{\text{lmx}}$ is the number of fitness values checked to determine if a local maximum has been found.

For a given tool t and its option t_{opt}, the function HC-TOOL-OPTION is invoked with the initial solution of t_{opt} set to s_0. In p, I track each solution sampled along with its fitness value as a pair. p is initialised with an empty list. The current solution is noted in s.

A search iteration works as follows. The tool t is launched for tool option t_{opt} (with its value set to s), along with the additional option t_{misc} (and its value set to v) passed (line

Algorithm 15 The hill climbing algorithm to optimise a single option

```
1: function HC-TOOL-OPTION(t, (t_opt,s_0), (t_misc,v), N_nei, σ_nei, N_lmx)
2:     p ←[ ]
3:     s ← s_0
4:     while True do
5:         runtool(t, (t_opt,s), (t_misc,v))
6:         fit_s ← fitness(s)
7:         p_s ← (s, fit_s)
8:         append(p, p_s)
9:         if LOCALMAX(p, N_lmx) then
10:            return BEST(p)
11:        end if
12:        s ← NEIGHBOUR(s, p, N_nei, σ_nei)
13:    end while
14: end function
```

5). The fitness for s is noted in fit_s (line 6). The solution s and its associated fitness fit_s is noted in the pair p_s (line 7). The pair p_s is then appended to p. The function LOCALMAX is invoked to determine if local maximum has been found (line 9). On finding a local maximum, the search stops and the function BEST is invoked to return a list containing the best performing solutions along with the maximum fitness (line 10). Alternatively, the search continues to explore the neighbourhood of the current solution s to find even better solutions. The function NEIGHBOUR is invoked to select a neighbour solution based on the current solution s, the list of pairs p containing the solution and their fitness values from the run so far, and the two constants $\mathcal{N}_{\text{nei}}$ and σ_{nei} (line 12). The heuristic continues to explore the solution space until no better solution can be found.

My hill climbing implementation relies on certain core functions: the function NEIGHBOUR that returns a neighbour for a given solution; the function LOCALMAX that determines if a local maximum has been found; and the function BEST that returns the best performing solution(s) from the current run. The description of the core functions follows.

5.8.3 Neighbour Selection

My approach for selecting a neighbour is based on the variation in the fitness values from the last $\mathcal{N}_{\text{nei}}$ runs. If the last $\mathcal{N}_{\text{nei}}$ fitness values show a noticeable variation (i.e. above a certain deviation), then a near neighbour is picked, otherwise a distant neighbour is picked. If the search has not sampled enough solutions ($< \mathcal{N}_{\text{nei}}$), then a near neighbour is picked.

Algorithm 16 describes how a neighbour is selected for a given solution. The function

Algorithm 16 Algorithm to select a neighbour for a given solution

```
1: function NEIGHBOUR(s, p, N_nei, σ_nei)
2:     if (|p| < N_nei) then
3:         return smallstep(s)
4:     end if
5:     p_N ← last(p, N_nei)                    ▷ Get last N_nei pairs from p
6:     fitness_N ← getfit(p_N)
7:     if (sdev(fitness_N) > σ_nei) then
8:         return smallstep(s)
9:     end if
10:    return bigstep(s)
11: end function
```

NEIGHBOUR generates a neighbour based on the following parameters: the given solution s; p is the list of pairs, where each pair contains the sampled solution and its respective fitness; and $\mathcal{N}_{\text{nei}}$ denotes the number of fitness values whose standard deviation determines whether a near or a distant neighbour is selected; and σ_{nei} is the threshold standard deviation constant.

When the search has just started and the number of solutions sampled is less than $\mathcal{N}_{\text{nei}}$, a near neighbour is picked by invoking the smallstep function (lines 2–4). The last $\mathcal{N}_{\text{nei}}$ pairs from p are obtained and noted in $p_{\mathcal{N}}$ (line 5). From $p_{\mathcal{N}}$, the last $\mathcal{N}_{\text{nei}}$ fitness values is obtained and noted in $\mathit{fitness}_{\mathcal{N}}$ by invoking the getfit function (lines 6). The standard deviation function sdev is applied to $\mathit{fitness}_{\mathcal{N}}$ to pick a neighbour: if the values in $\mathit{fitness}_{\mathcal{N}}$ have varied beyond threshold value ($> \sigma_{\text{nei}}$), then a near neighbour is picked by invoking the smallstep function (lines 7–9); alternatively, the variation in the fitness values has been negligible, and so a distant neighbour is picked by invoking the bigstep function (line 10).

5.8.4 Local Maxima

To determine if a local maximum has been found, my heuristic does not examine just the fitness value of the last solution sampled but examines the fitness value of the last $\mathcal{N}_{\text{lmx}}$ solutions to be reasonably sure that the fitness can't be improved any further. From the solutions sampled so far, if any of the ones sampled in the last $\mathcal{N}_{\text{lmx}}$ runs has a better fitness than the ones that precedes the last $\mathcal{N}_{\text{lmx}}$ runs, then I predict the current run is still climbing the hill and so I continue to explore the solution space to find even better solutions. Alternatively, if the last $\mathcal{N}_{\text{lmx}}$ runs do not produce a better solution, then the search is stopped and is said to have found a local maximum.

Algorithm 17 describes how the local maximum is found. The function LOCALMAX accepts two parameters: p is the list of pairs, where each pair contains a solution that

Algorithm 17 Algorithm to determine local maximum

```
1: function LOCALMAX(p, N_lmx)
2:     if |p| ≤ N_lmx then
3:         return False
4:     end if
5:     p_f ← first(p, |p|−N_lmx)          ▷ Get all but last N_lmx pairs from p
6:     fitness_f ← getfit(p_f)
7:     maxfit ← max(fitness_f)
8:     p_n ← last(p, N_lmx)               ▷ Get last N_lmx pairs from p
9:     fitness_n ← getfit(p_n)
10:    for fit ∈ fitness_n do
11:        if fit > maxfit then
12:            return False
13:        end if
14:    end for
15:    return True
16: end function
```

has been sampled so far along with its respective fitness; and $\mathcal{N}_{\text{lmx}}$ denotes the number of fitness values that is examined to determine if a local maximum has been found. To determine local maximum, at least $\mathcal{N}_{\text{lmx}}$ solutions should have been sampled (lines 2–4). From p, the list of all but the last $\mathcal{N}_{\text{lmx}}$ pairs is obtained and noted in p_f (line 5). From p_f, the list of fitness values is obtained by invoking the getfit function, and the result is noted in $fitness_f$ (line 6). The maximum fitness value from $fitness_f$ is calculated and noted in *maxfit* (line 7). The last $\mathcal{N}_{\text{lmx}}$ pairs from p are obtained and noted in p_n (line 8). From p_n, the last $\mathcal{N}_{\text{lmx}}$ fitness values are obtained by invoking the getfit function and the result is noted in $fitness_n$ (line 9). The local maximum is said to have been found, if none of the fitness values from $fitness_n$ is $>$ *maxfit* (lines 10–15).

Since the heuristic to determine the local maximum does not stop at the first dip in the fitness value but examines $\mathcal{N}_{\text{lmx}}$ fitness values before deciding on the local maximum, there is a possibility that the search uncovers more than one solution with the best fitness. Consequently, the heuristic picks all the best fit solutions from the search.

Algorithm 18 describes the selection of best performing solutions from a search. The function BEST requires an input: p, a list of pairs, where each pair contains a solution that has been sampled so far along with its respective fitness. The maximum fitness *maxfit* is calculated from p (line 3). Each solution from p whose fitness matches the maximum fitness *maxfit* is noted in *best* (lines 5–9). The list containing the best performing solutions (*best*) along with the maximum fitness is then returned.

With the hill climbing implementation that optimises a single option for a given tool now

Algorithm 18 Algorithm to determine the best solution(s)

```
1: function BEST(p)
2:     fitness ← getfit(p)
3:     maxfit ← max(fitness)
4:     best_s ← [ ]
5:     for s, f ∈ p do
6:         if f = maxfit then
7:             append(best_s, s)
8:         end if
9:     end for
10:    return best_s, maxfit
11: end function
```

defined, the hill climbing setup for AMBER, AmbiDexter, and the $dynamic_n$ backends can be defined.

5.8.5 Hill Climbing – AMBER

AMBER supports ambiguity detection through search by length or by examples. Hill climbing is applied to the AMBER tool to optimise the length or the examples option.

The function HC-TOOL-OPTION (from Algorithm 15) is invoked with tool t=amber and tool option t_{opt} set to either 'length' or 'examples'. s is the initial solution for length or examples to start the search from. The additional tool option t_{misc} is set to 'ellipsis', and its value v is set to Yes/No depending on whether the sub-option ellipsis is being explored or not. $\mathcal{N}_{\text{nei}}$, σ_{nei}, and $\mathcal{N}_{\text{lmx}}$ are hill climbing constants that are set at run-time.

5.8.6 Hill Climbing – AmbiDexter

AmbiDexter supports ambiguity detection by searching for strings of fixed length. Hill climbing is applied to AmbiDexter to optimise the length option.

The function HC-TOOL-OPTION (from Algorithm 15) is invoked with tool t=ambidexter and tool option t_{opt} set to 'length'. s is the initial length to start the search from. The additional tool option t_{misc} is set to 'filter', and its value v is set to the filter that is being optimised (see Table 5.2 for list of filters). $\mathcal{N}_{\text{nei}}$, σ_{nei}, and $\mathcal{N}_{\text{lmx}}$ are hill climbing constants that are set at run-time.

5.8.7 Hill Climbing – Dynamic Backends

SinBAD's dynamic backends supports ambiguity detection through sentence generation. The sentence generation relies on threshold depth D to favour alternatives. For $dynamic_n$ backends, the hill climbing technique is implemented to optimise depth.

The function HC-TOOL-OPTION (from Algorithm 15) is invoked with tool t=$dynamic_n$ (see Table 5.2 for list of backends) and the tool option t_{opt} set to 'depth'. s is the initial depth to start the search from. $dynamic_n$ backends do not support any additional options, so option t_{misc} and its value v are set to null. $\mathcal{N}_{\text{nei}}$, σ_{nei}, and $\mathcal{N}_{\text{lmx}}$ are hill climbing constants that are set at run-time.

5.8.8 Hill Climbing – The $dynamic2_{rws}$ Backend

The $dynamic2_{rws}$ backend works in a similar style to the *dynamic2* backend but relies on an additional parameter W, a probabilistic weight applied to pick alternatives other than the low cost ones to mitigate recursive cycles whilst still preserving the *dynamic2*'s general approach. For the $dynamic2_{rws}$ backend, the hill climbing technique is applied to optimise the combination of depth D and weight W.

My approach to uncover the best performing combination of depth and weight comprises of two routines: a main routine to optimise depth; and an auxiliary routine to find an optimal weight for each depth. The heuristic to optimise depth works in a similar style to the heuristic for the $dynamic_n$ backend (outlined in Algorithm 15) but with one exception: in line 5, instead of invoking the function `runtool` to launch the tool for a given depth, I invoke the weight optimising routine to uncover best performing weight(s) for that depth. Once the value of depth is fixed, the weight optimising routine then becomes a hill climbing heuristic (see Algorithm 15) for optimising a single option, with weight as the main option and depth as the sub-option.

Since the solution space for weight is infinite, to have a reasonable chance of uncovering the best performing weight for a given depth, the weight optimising routine is instantiated at various points in the solution space. The weight with the best fitness then becomes the best performing weight for that depth.

Algorithm 19 describes my heuristic to uncover the best combination of depth and weight for the weighted $dynamic_n$ backends. The function HC-DYNAMIC2-RWS accepts the following parameters: b represents the dynamic backend; d_0 is the initial depth; $wgts_0$ contains a list of initial weights to explore for each depth; $\mathcal{N}_{\text{nei}}$ is a constant that refers to the number of fitness values whose standard deviation controls neighbour selection; σ_{nei} is the threshold standard deviation constant that applies for neighbour selection; $\mathcal{N}_{\text{lmx}}$ is the number of fitness values checked to determine if local maximum has been found.

Algorithm 19 The hill climbing algorithm for the $dynamic2_{rws}$ backend

```
1: function HC-DYNAMIC2-RWS(t, (t_opt,d_0), (t_misc,wgts_0), N_nei, σ_nei, N_lmx)
2:     p ←[ ]
3:     d ← d_0
4:     while True do
5:         w, fit ← OPTIMISE-WGT(t, (t_opt,d), (t_misc,wgts_0), N_nei, σ_nei, N_lmx)
6:         p_d ← ((d, w), fit)
7:         append(p, p_d)
8:         if LOCALMAX(p, N_lmx) then
9:             return BEST(p)
10:        end if
11:        d ← NEIGHBOUR(d, p, N_nei, σ_nei)
12:    end while
13: end function

14: function OPTIMISE-WGT(t, (t_opt,d), (t_misc,wgts_0), N_nei, σ_nei, N_lmx)
15:    p_wgts ← [ ]
16:    for w ∈ wgts_0 do
17:        w, fit_w ← HC-TOOL-OPTION(t, (t_misc,w), (t_opt,d), N_nei, σ_nei, N_lmx)
18:        p_w ← (w, fit_w)
19:        append(p_wgts, p_w)
20:    end for
21:    return BEST(p_wgts)
22: end function
```

For a given backend b, the function HC-DYNAMIC2-RWS is invoked with initial depth set to d_0. p keeps track of the depth that has been sampled so far along with the best performing weight for each depth and its fitness. p is initialised with an empty list (line 2). The current depth is noted in d (line 3). For a given depth d, the function OPTIMISE-WGT explores weight at different intervals to uncover the best performing weight (line 5). The best performing weight w at depth d along with fitness is noted in p_d (line 6). The list of pairs p is updated with p_d (line 7).

The function LOCALMAX is invoked to determine if local maximum has been found (line 8). On finding a local maximum, the search is stopped and the function BEST is invoked to return a list containing the best performing combination of depth and weight along with the maximum fitness (line 9). Otherwise, the search continues to explore the neighbourhood of the current solution d to find even better solutions. The function NEIGHBOUR is invoked to select a neighbour based on the current depth d, the list of pairs p containing the solution and their fitness values from the run so far, and the two

constants $\mathcal{N}_{\text{nei}}$ and σ_{nei} (line 11). In this way, the heuristic continues to explore the search space until no fitter solution can be found.

For a given backend b, depth d and a list of initial weights $wgts_0$, the function OPTIMISE-WGT aims to uncover the best performing weight for depth d. p_{wgts} keeps track of the weight that has been sampled so far along with its fitness. p_w is initialised as an empty list (line 15). For the given depth d, and for each weight from $wgts_0$, the function HC-TOOL-OPTION (from Algorithm 15) is invoked with t=b, t_{opt} set to weight and its value s_0 set to w, t_{misc} set to depth and its value v set to d along with the constants (line 17). The best performing weight w and its respective fitness fit_w is noted in p_w (line 18). p_{wgts} is updated with p_w (line 19). The best performing weight for the given depth d is then returned by invoking the function BEST on p_{wgts} (line 21).

For the dimensioning experiments, the hill climbing functions HC-TOOL-OPTION and HC-DYNAMIC2-RWS are executed for the tool options shown in Table 5.2.

5.9 Crude Dimensioning

The aim of crude dimensioning is to get an overview of the fitness distribution for each tool option and for each grammar set. As such, for each tool option, I try to explore its solution space as much as practical over two runs. First, I use hill climbing with a carefully chosen run-time value for each of its control parameters. Since hill climbing, by definition, is a local optimisation technique, to get an idea of the fitness distribution beyond the neighbourhood covered by hill climbing, I perform additional tool runs. For each tool option, manual runs are performed to sample points at various intervals in the solution space.

5.9.1 Grammar Corpus and Time Limit

The grammar corpus for the crude dimensioning run is show in the Table 5.1). For both the hill climbing run and the additional tool runs, the ambiguity tools were set to run for a time limit of 10s.

5.9.2 Hill Climbing – Run-time Values

The hill climbing run requires a run-time value to be set for each of its control parameters. $\mathcal{N}_{\text{nei}}$ and σ_{nei} control neighbour selection. $\mathcal{N}_{\text{lmx}}$ controls the number of fitness values to checked to determine local maximum. The step size functions $\mathsf{smallstep}(s)$ and $\mathsf{bigstep}(s)$ that determine neighbour selection for a given solution s. The initial solution s_0 for the tool option to start the search from. For the crude dimensioning experiment, the hill

climbing run is seeded with run-time values based on my previous experience of running such experiments from [40].

Using my experience of the tools in question, I set $\mathcal{N}_{\text{lmx}}$, $\mathcal{N}_{\text{nei}}$, and σ_{nei} to be 3 for all the tool options. With regards to the step sizes, for AMBER length, AmbiDexter length, and the depth option for the *SinBAD*'s backends, the small and big step size were set to 1 and 3 respectively. For the AMBER examples option, for a given solution s, the small and big step size increments s by 5% and 10% respectively. For the initial solution s_0, for AMBER length (ellipsis set or otherwise), AmbiDexter length (filter set or otherwise), and *SinBAD*'s backends, I set $s_0 = 1$.

For AMBER examples option, to get a rough idea of the number of examples that it can check in a given time limit, I performed a tiny experiment. AMBER was launched for 'examples' option with N=10^{10} on the grammar corpus used for crude dimensioning for a time limit t=10s. My tiny experiment revealed that the number of examples checked, ranged from: 4K to 6.7M for the Boltzmann grammar set; 1K to 5.8M for the altered PL grammar set; and 2K to 5.5M for the mutated grammar set. The low value in each case corresponds to the cases when the AMBER finds ambiguity quickly and exits. For the crude dimensioning experiment, since I wish to understand how ambiguity scales up as N increases, for all the three grammar sets, I set $s_0 = 100K$.

5.9.3 Invoking Hill Climbing Functions

For AMBER length, since I need to explore with ellipsis set and unset, I invoked two instances of the function HC-TOOL-OPTION with t=amber, t_{opt} ='length', s=1, t_{misc} ='ellipsis', and v set to Yes or No. Likewise, for AMBER examples, two instances of the function HC-TOOL-OPTION were invoked with t=amber, t_{opt} ='examples', s=100K, t_{misc} ='ellipsis', and v set to either Yes or No.

For AmbiDexter length, five instances of the function HC-TOOL-OPTION were invoked with t=ambidexter, t_{opt} ='length', s=1, t_{misc} ='filter' and for v set to one of None, LR0, SLR1, LALR1, or LR1.

For the $dynamic_n$ backends, four instances of the function HC-TOOL-OPTION were invoked for each of t=*dynamic1*,*dynamic2*,*dynamic3*, and *dynamic4*; t_{opt} ='depth'; and s=1. The $dynamic2_{rws}$ backend requires an additional parameter weight W to probabilistically unpick low cost alternatives for mitigating non-termination. My careful observation of the run-time behaviour of $dynamic2_{rws}$ for the Boltzmann, altered PL, and mutated grammar sets suggests that a low weight is sufficient to significantly reduce non-termination. I therefore used W=0.01. For the $dynamic2_{rws}$ backend, the function HC-DYNAMIC2-RWS was invoked with t=$dynamic2_{rws}$, t_{opt} ='depth', s=1, t_{misc} ='weight', and v=0.01.

5.9.4 Additional Tool Runs

Additional tool runs were performed to sample solution regions unexplored by the hill climbing run. For both Boltzmann and mutated grammar sets, additional tool runs were performed for the following tool options: for AMBER length (with ellipsis set or otherwise) len=50, 75, 100, 150, 200, and 500; for AmbiDexter length (for unfiltered and for each filter) len=50, 75, 100, 150, 200, and 500; for AMBER examples (ellipsis set or otherwise) N=2×10^6, 5×10^6, 10^7, 10^8, 10^9, and 10^{10}; and for the $dynamic_n$ depth option D=25, 50, 75, 100, 150, 200, and 500.

For the $dynamic2_{rws}$ backend, additional tool runs were performed for the following combination of depth D and weight W: for Boltzmann grammars, for depths D=25 and 50 and weights W=0.005, 0.05, 0.075, 0.1, and 0.2; and for mutated grammars, for depths D=30 and 50 and weights W=0.005, 0.05, 0.075, 0.1, 0.2, 0.3, 0.4 and 0.5.

5.9.5 Crude Dimensioning Results

I now present the results of the crude dimensioning run for each tool option for the Boltzmann and the mutated PL grammar set. Since the altered PL grammar set is tiny, the dimensioning results are less interesting. The results for the altered PL grammar set are shown in Appendix B. The results of the crude dimensioning run can be downloaded from: `https://figshare.com/s/0aac8b6f2b581a4f0326`.

5.9.5.1 Boltzmann Grammars

Figure 5.1 shows the results from the crude dimensioning run for AMBER length and examples options, AmbiDexter length and the *SinBAD*'s backends (*dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*) for the Boltzmann grammar set. Table 5.4 shows the results from the crude dimensioning run for the *dynamic2rws* backend for the Boltzmann grammar set.

From Figure 5.1, we can see that the fitness distribution for the tool options – AMBER length (ellipsis set or otherwise), AMBER examples (ellipsis set or otherwise), AmbiDexter length (for unfiltered and for each filter) – is asymptotic (i.e. plateauing as it converges on a value). In case of the *SinBAD*'s backends – *dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*– the fitness distribution is a bell curve. That is, as the value of the solution increases, the fitness increases for a while and then starts to dip. For the $dynamic2_{rws}$ backend, the fitness distribution is hill shaped. That is, as the depth increases, the fitness seems to ramp up for a while and then starts to dip. For each depth, as the value of weight increases, the fitness seems to ramp up for a while and then starts to dip.

For each tool option, the solution space containing potential good solutions (indicated by gray shaded region in Figure 5.1) is explored in detail in the fine dimensioning experiment.

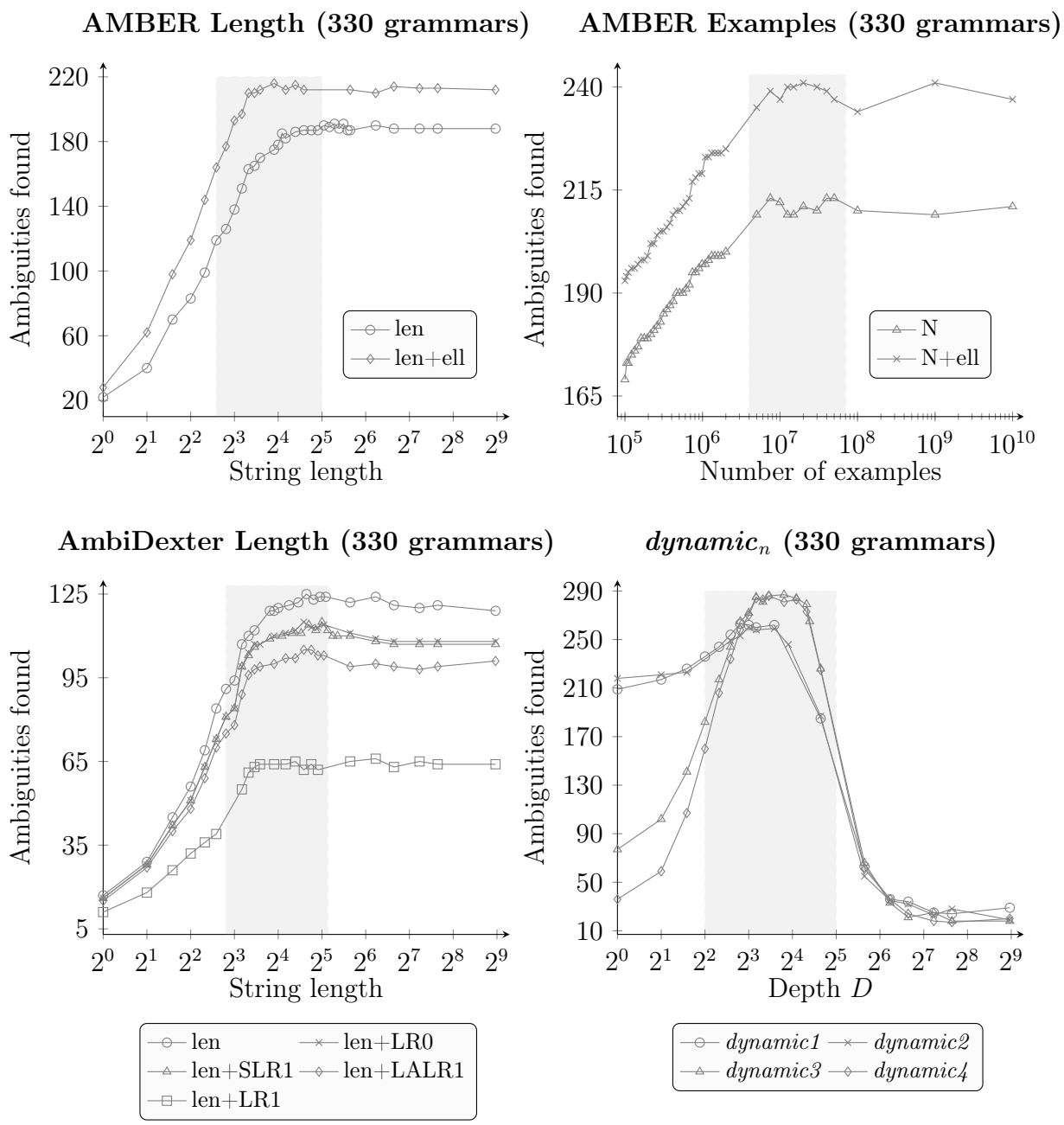

Figure 5.1: Crude dimensioning: the number of ambiguities (out of 330 grammars) found by tool options, AMBER and AmbiDexter length, AMBER examples, and the *dynamic*$_n$ backends for Boltzmann grammars.

5.9.5.2 Mutated Grammars

Figure 5.2 shows the results from the crude dimensioning run for AMBER length and examples options, AmbiDexter length and the *SinBAD*'s backends (*dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*) for the mutated grammars. Table 5.5 shows the results from the crude dimensioning run for the *dynamic2rws* backend for the mutated grammars. In certain cases, my hill climb run terminated early. For AMBER examples, the hill climbing run required several restarts. In both cases, with ellipsis set or otherwise, hill climbing

W[♭]\D	1	2	3	6	7	8	9	10	13	25	50
0.005	236	238	241	257	258	263	263	260	250	181	59
0.01	238	239	241	255	256	261	258	259	254	178	57
0.0105	242	237	241	250	254	261	258	258	252		
0.011025	234	244	239	254	249	258	256	260	248		
0.011576	241										
0.012127		243	239	254	255	264[†]	258	259	247		
0.012155	241										
0.012733		243									
0.013340						259		261			
0.014007		236									
0.014674						254		256			
0.015407						255					
0.016141								256			
0.017755								259			
0.05	245	247	249	249	257	260	254	252	247	179	57
0.075	243	247	248	254	256	257	259	251	244	170	55
0.1	245	249	248	247	248	252	247	252	240	172	53
0.2	222	226	224	225	224	226	227	223	222	160	53

[♭] Weights are approximated to the nearest hundred thousandth.
[†] Maximum number of ambiguities found.

Table 5.4: Crude dimensioning for *dynamic2*$_{rws}$ backend: the number of ambiguities found (out of 330 grammars) for various values of D and W for Boltzmann grammars.

was restarted âĽĹ 5 times.

From Figure 5.2, we can see that the fitness distribution for the tool options – AMBER length, AmbiDexter length, and AMBER examples – is asymptotic. For the *SinBAD*'s backends, the fitness distribution is a bell curve. For all the *dynamic*$_n$ backends, as D increases, the fitness drops. For the *dynamic2*$_{rws}$ backend, the fitness distribution is a hill shaped. That is, as the depth increases, the fitness ramps up for a while and then starts to dip. For each depth, as the value of weight increases, the fitness ramps up for a while and then start to dip.

For each tool option, the solution space containing potential good solutions (indicated by gray shaded region in Figure 5.2) is explored in detail in the fine dimensioning experiment.

5.10 Crude Dimensioning – Summary

The results from the crude dimensioning experiment show that for each tool the shape of the fitness distribution is consistent across the different grammar corpuses. This means that we can choose a single value from the potential good regions of the solution space

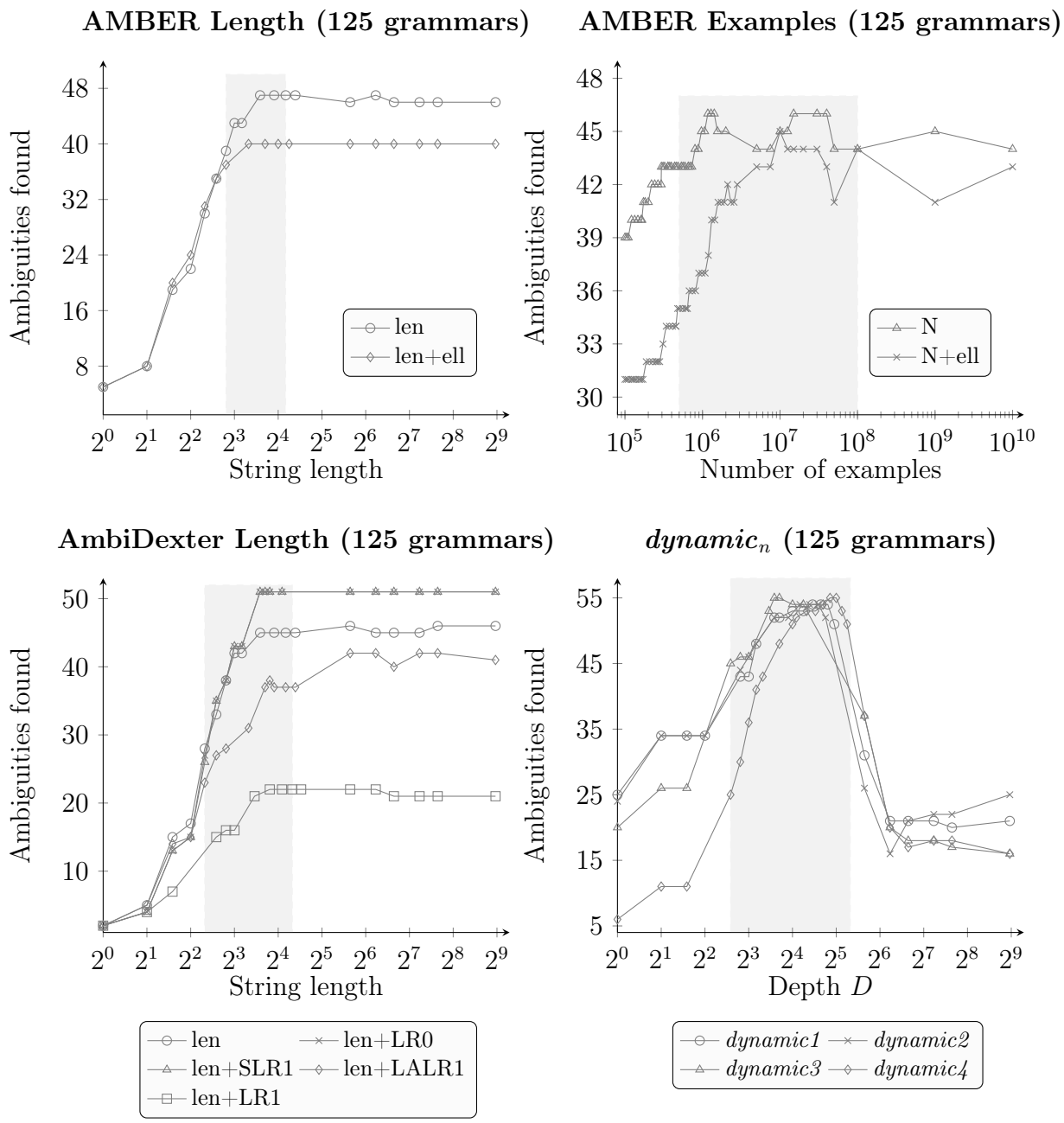

Figure 5.2: Crude dimensioning: the number of ambiguities (out of 125 grammars) found by tool options, AMBER and AmbiDexter length, AMBER examples, and the *dynamic*$_n$ backends for mutated grammars.

and expect it to do fairly well across my grammar corpuses.

My choice of values for the constants ($\mathcal{N}_{\text{nei}}$, $\mathcal{N}_{\text{lmx}}$, and σ_{nei}), and the small and big step sizes were good enough for most of the tool options, and I went with the same values for the fine dimensioning run. Only in the case of AMBER examples for the altered PL grammar set, since the hill climbing required several restarts (AMBER was restarted 10 times), I felt a small change was necessary. Since the altered PL grammar set is fairly small, I chose to let the hill climbing run for as long as it was needed, and then terminate

W[♭]\D	1	2	3	6	9	12	15	18	21	30	50
0.005	43	45	48	48	51	54	56	52	53	52	27
0.01	42	49	47	52	51	53	53	55	54	54	24
0.0105	42	47	50	49	52	55	54	53	53		
0.011025	44	50	49	52	54	55	55	53	54		
0.012127	42	46	48	50	51	55	55	53	53		
0.013340	41	48	48		50	55	55				
0.014674	45	47			51		54				
0.016141	46										
0.017755	46										
0.019531	41										
0.021484	45										
0.03	46	50	51	51	51	52	54	55	53	50	28
0.04	48	50	50	52	52	54	55	55	55	54	31
0.05	49	50	53	53	53	54	53	56	54	50	29
0.075	51	52	53	54	53	54	55	57[†]	55	54	26
0.1	51	54	54	55	53	55	56	55	56	51	27
0.2	55	55	53	55	56	54	54	54	54	46	30
0.3	54	52	52	54	53	54	54	51	49	32	24
0.4	37	40	41	37	35	37	38	34	36	32	26
0.5	36	37	37	37	34	33	31	30	30	27	25

[♭] Weights are approximated to the nearest hundred thousandth.
[†] Maximum number of ambiguities found.

Table 5.5: Crude dimensioning: the number of ambiguities (out of 125 grammars) found by *dynamic2rws* for various D and W for the mutated grammars.

it manually. That is, for AMBER examples option for the altered PL grammar set, lines 9–11 were skipped in function HC-TOOL-OPTION (see Algorithm 15). In the case of AMBER examples for the mutated grammar set, which also required several hill climbing restarts (AMBER was restarted 5 times), I felt that the increased size of the grammar collection should help in alleviating the early termination problem.

Tables 5.6, 5.7, 5.8, 5.9, 5.10 list the initial solution used for the fine dimensioning run for tool options AMBER length, AMBER examples, AmbiDexter length, AmbiDexter search by incremental length, *SinBAD*'s backends (*dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*), and the $dynamic2_{rws}$ backend respectively.

Table 5.11 lists the step sizes (small and big) applied for each tool option for all grammar sets.

I now perform the fine dimensioning of the tool options based on the run-time values chosen from the crude dimensioning run.

	Initial length		
Ellipsis	Boltzmann	Altered PL	Mutated
-	9	6	6
Yes	9	6	6

Table 5.6: AMBER: initial solution for length for Boltzmann, Altered PL and Mutated grammar sets.

	Initial number of examples		
Ellipsis	Boltzmann	Altered PL	Mutated
-	2×10^6	1×10^6	6×10^5
Yes	2×10^6	3×10^6	2×10^6

Table 5.7: AMBER: initial solution for examples for Boltzmann, Altered PL and Mutated grammar sets.

	Initial length		
Filter	Boltzmann	Altered PL	Mutated
-	8	6	5
LR0	8	6	7
SLR1	8	9	8
LALR1	9	9	7
LR1	9	9	7

Table 5.8: AmbiDexter: initial solution for option length for Boltzmann, Altered PL and Mutated grammar sets.

	Initial Depth D		
Backend	Boltzmann	Altered PL	Mutated
dynamic1	5	6	7
dynamic2	3	6	7
dynamic3	7	1	6
dynamic4	7	9	8

Table 5.9: $dynamic_n$ backends: initial solution for option depth.

	Initial D and W					
	Boltzmann		Altered PL		Mutated	
Backend (b)	D	W	D	W	D	W
dynamic2rws	6	0.01	6	0.01	9	0.03

Table 5.10: The $dynamic2_{rws}$ backend: initial solution for depth D and weight W for the Boltzmann, altered PL, and the mutated grammar sets.

Tool	Option	smallstep(s)	bigstep(s)
AMBER	Length	$s + 1$	$s + 3$
	Example	$s \times 1.05$	$s \times 1.1$
AmbiDexter	Length	$s + 1$	$s + 3$
$dynamic_n$	Depth	$s + 1$	$s + 3$
$dynamic2_{rws}$	Depth	$s + 1$	$s + 3$
	Weight	$s \times 1.05$	$s \times 1.1$

Table 5.11: Neighbour selection: step sizes (small and big) for tool options for all grammar sets.

5.11 Fine Dimensioning

The aim of fine dimensioning is to find the best performing solution for each tool option from Table 5.2. Fine dimensioning for each tool option involves two runs: a hill climbing run is performed based on the run-time values from Section 5.10, followed by additional hill climbs and tool runs to improve on the solution uncovered by the hill climbing run. For AmbiDexter 'search by incremental length' option, since the tool already searches incrementally from an initial length, no hill climbing was performed; instead, AmbiDexter was invoked to search from length 0. Additionally, for this option, AmbiDexter was invoked with each of the filter set too.

5.11.1 Grammar Corpus and Time Limit

For each tool option, the experiment is invoked for each grammar collection (see Table 5.1). Since grammars can specify infinite languages, grammar ambiguity tools can run forever. In [40], I observed that for a time limit of 30s, most tool options produced reasonable quality results. Only in the case of AMBER, for the PL grammar set and the mutated PL grammar set (add symbol and mutate symbol), was there a noticeable difference between

30s and 60s. In all three cases, AMBER found roughly 10% more ambiguities for 60 seconds. Since choosing a 60s time limit will significantly increase the running time of the fine dimensioning run, I settled for a time limit of 30s. I don't believe this is an issue, as for the main experiment the tools are evaluated for much extended time limits.

5.11.2 Results

I now present the results from the fine dimensioning run for the Boltzmann, and the mutated grammar sets. The results of the fine dimensioning run for the altered PL grammar set are shown in Appendix C. The results of the fine dimensioning run can be downloaded from:`https://figshare.com/s/798f37cdba7898f839da`.

5.11.2.1 Boltzmann Grammars

Figure 5.3 shows the results from the fine dimensioning run for the tool options AMBER length, AMBER examples, AmbiDexter length, and the *SinBAD*'s backends – *dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*– for the Boltzmann grammar set.

AmbiDexter *ilen*				
-	LR0	SLR1	LALR1	LR1
260	240	236	219	160

Table 5.12: Fine dimensioning for the AmbiDexter incremental length *ilen* option: the number of ambiguities (out of 660 grammars) found for the unfiltered and the filtered options for the Boltzmann grammar set.

Table 5.12 shows the results of the fine dimensioning run for the AmbiDexter search by incremental length *ilen* option for the unfiltered and the filtered options for the Boltzmann grammar set. Table 5.13 shows the results of the $dynamic2_{rws}$ backend for the Boltzmann grammar set.

Table 5.14 shows the additional tool runs for the AMBER length and the examples option for the Boltzmann grammar set. For AMBER length (with ellipsis not set), the additional tool runs did not uncover a better solution. For AMBER length (with ellipsis set), the additional runs uncovered a better solution for len=24 (ambiguities found=438).

For AMBER examples, additional runs were performed on ±5% of the best solutions from the hill climbing run. For the AMBER examples (with ellipsis set or otherwise) option, the additional runs did not uncover a better solution.

W[♭]\D	6	7	8	9[‡]	10[‡]	11	12[‡]	13[‡]	14	15	16
0.01	539	548	541	547	547	544	540	539	542	531	524
0.0105	545	544	549	545	547	545	537	544	536	533	524
0.011025	535	541	548	545	551	549	546	540	534	527	525
0.011576	536		542				542		540		
0.012127		547		548	547	543		542		531	515
0.012155	533		548				542				
0.012673[‡]						549					
0.012733											530
0.013340				540	544	554[†]		537		529	
0.013370							543				525
0.014007				542		546					
0.014039											524
0.014674					545						
0.014707				539		542					
0.015443						548					522

[♭] Weights are approximated to the nearest hundred thousandth.
[†] Maximum number of ambiguities found.
[‡] Additional hill climb and tool runs.

Table 5.13: Fine dimensioning for *dynamic2*$_{rws}$ backend: the number of ambiguities (out of 660 grammars) found for various values of D and W for Boltzmann grammars.

AMBER			
len	len+ell	N	N+ell
(36, 421)	(23, 435)	(28907584, 421)	(75320034, 484)
		(31798343, 421)	
(34, 363)	(21, 441)	(27462204, 421)	(71554032, 483)
(35, 365)	(22, 434)	(30308425, 420)	(79086035, 481)
(37, 357)	(24, 438)[♯]	(33388260, 420)	
(38, 364)	(25, 437)		

[♯] Better solution found from the additional tool runs.

Table 5.14: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the AMBER length and the examples option for the Boltzmann grammar set. Pair value (l, f) denote the length l sampled and its fitness value. Pair value (N, f) denote the number of examples N sampled and its fitness value.

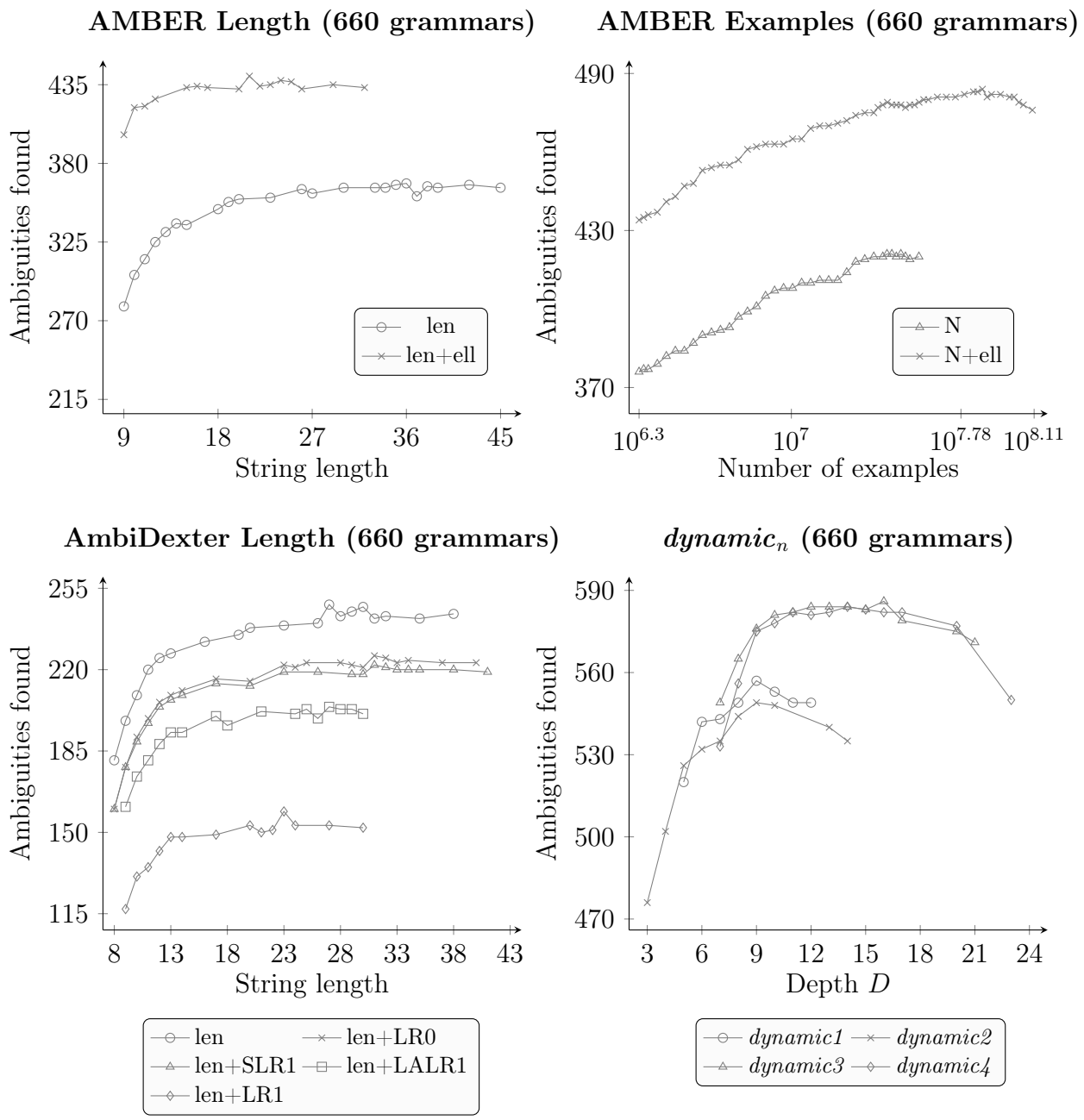

Figure 5.3: Fine dimensioning for the Boltzmann grammar set: number of ambiguities (out of 660 grammars) found by tool options, AMBER length, AMBER examples, AmbiDexter length and the $dynamic_n$ backends.

Table 5.15 shows the additional tool runs performed for the AmbiDexter length option for the Boltzmann grammar set. For AmbiDexter length option for filters LR0, LALR1, and LR1, the additional runs did not uncover a better solution. For unfiltered and for filter SLR1, the additional runs uncovered a better solution for len=27 (ambiguities found=248) and len=31 (ambiguities found=222) respectively.

Table 5.16 shows the additional tool runs performed for the $dynamic_n$ backends depth option for the Boltzmann grammar set. For the *dynamic1* and the *dynamic2* backends,

AmbiDexter Length				
unf≙	LR0	SLR1	LALR1	LR1
(29, 245)	(31, 226)	(32, 221)	(27, 204)	(23, 159)
(27, 248)♯	(29, 222)♯	(30, 218)	(25, 203)	(21, 150)
(28, 243)	(30, 221)	(31, 222)	(26, 199)	(22, 151)
(30, 247)	(32, 225)	(33, 220)	(28, 203)	
(31, 242)	(33, 223)	(34, 220)	(29, 203)	

♯ Better solution found from the additional tool runs.
≙ AmbiDexter invoked with filter not set.

Table 5.15: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the AmbiDexter length option for the Boltzmann grammar set. Pair value (l, f) denote the length l sampled and its fitness value.

there were no potential neighbouring solutions to explore, so additional tool runs were not performed. For the *dynamic3* backend, the additional runs uncovered a better solution for D=16 (ambiguities found=586). For the *dynamic4* backend, the additional runs did not uncover a better solution.

dynamic$_n$ Depth			
dynamic1	*dynamic2*	*dynamic3*	*dynamic4*
(9, 557)	(9, 549)	(14, 584)	(14, 584)
		(12, 584)	(12, 581)
		(13, 584)	(13, 582)
		(15, 583)	(15, 583)
		(16, 586)♯	(16, 582)

♯ Better solution found from the additional tool runs.

Table 5.16: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the *dynamic*$_n$ depth option for the Boltzmann grammar set. Pair value (D, f) denote the depth D sampled and its fitness value.

For the additional runs for *dynamic2*$_{rws}$ for the Boltzmann grammar set, see Table 5.13. For *dynamic2*$_{rws}$, since I need to explore the neighbourhood of the local maximum for both depth D and weight W, I had to perform two sets of runs: hill climb runs to explore depth in the proximity of the best performing depth; and tool runs to explore weight in the proximity of the best performing weight. The additional hill climb runs and the tool runs did not uncover a better solution.

5.11.2.2 Mutated Grammars

Figure 5.4 shows the results of the fine dimensioning run for the tool options AMBER length, AMBER examples, AmbiDexter length, and the $dynamic_n$ backends for the mutated grammar set. Table 5.17 shows the results of the fine dimensioning run for the AmbiDexter incremental length option for the unfiltered and filtered options for the mutated grammar set. Table 5.18 shows the results of the fine dimensioning run for the $dynamic2_{rws}$ backend for the mutated grammar set.

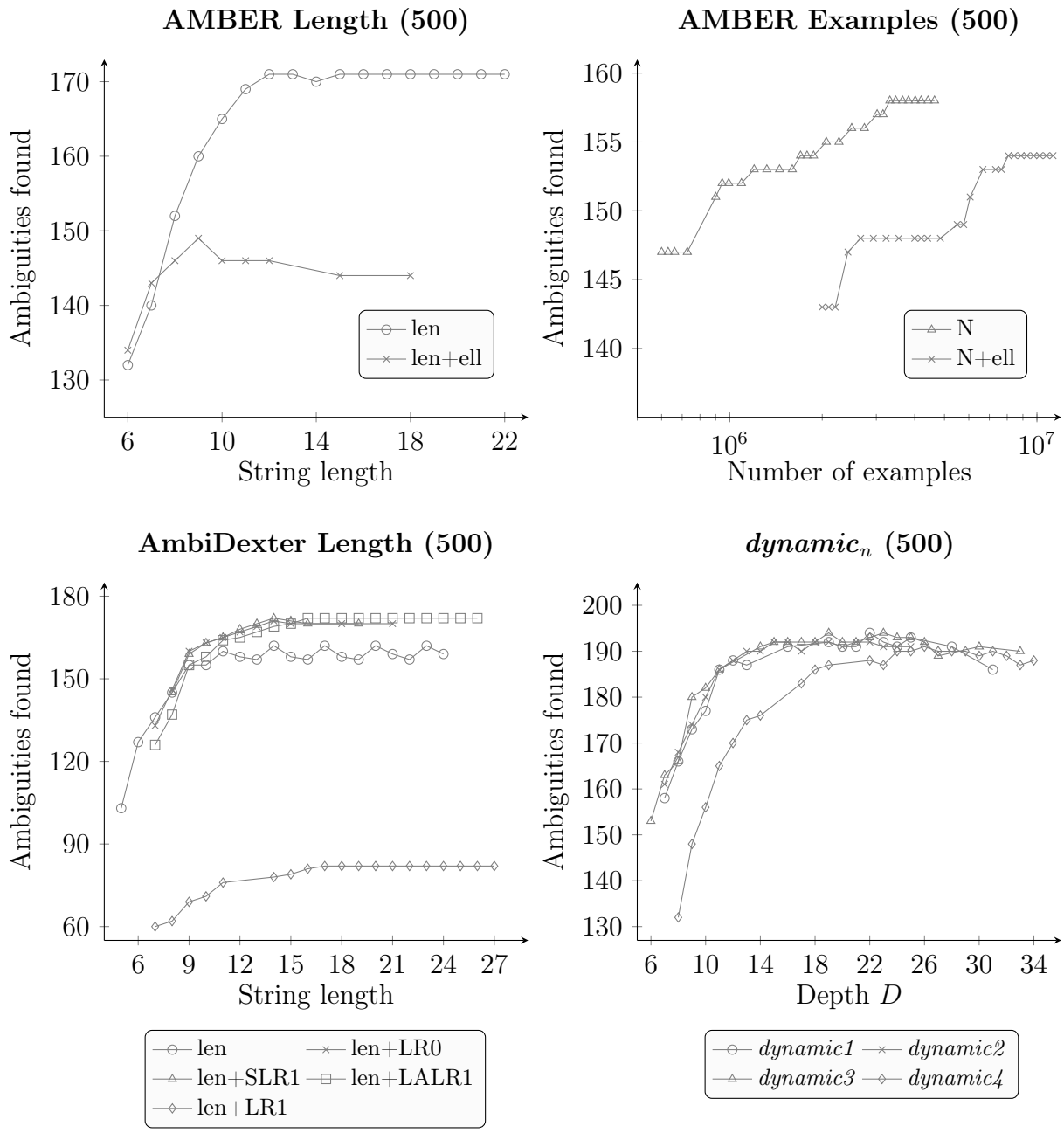

Figure 5.4: Fine dimensioning for the mutated grammar set: number of ambiguities found by tool options, AMBER length, AMBER examples, AmbiDexter length, and the $dynamic_n$ backends.

AmbiDexter *ilen*				
-	LR0	SLR1	LALR1	LR1
168	178	177	178	86

Table 5.17: Fine dimensioning for AmbiDexter *ilen*: number of ambiguities found for unfiltered and filtered options for the mutated grammar set.

W[♭]\D	9	10	11	12[‡]	13[‡]	14	15[‡]	16[‡]	17	18[‡]	19[‡]	20	21[‡]	22[‡]	23
0.03	193	192	197	197	196	195	197	198	196	196	196	196	195	195	195
0.0315	191	192	194	193	197	197	196	195	196	197	197	197	195	197	193
0.033075	188	192	195	195	196	196	195	198	196	197	198	195	194	195	192
0.034563												198[‡]			
0.036382	192	194	195	196	197	197	195	196	196	197	196	198[†]	196	197	194
0.038019						195[‡]									
0.038201												198[‡]			
0.040020		192			197	198[†]	196			196	196	197	196	196	
0.041821						197[‡]									
0.042021						198[‡]									
0.044022		194				198[†]					196		196		
0.046003						196[‡]									
0.046223						198[‡]									
0.048425		192				198[†]							195		
0.050604						197[‡]									
0.050846						198[‡]									
0.053267						198[†]									
0.055930						197[‡]									

[♭] Weights are approximated to the nearest hundred thousandth.
[†] Maximum number of ambiguities found.
[‡] Additional hill climb and tool runs.

Table 5.18: Fine dimensioning for the *dynamic2rws* backend: the number of ambiguities found for various D and W for the mutated grammar set. Several combinations of D and W found all the ambiguities.

Table 5.19 shows the additional tool runs for the AMBER length and the AMBER examples option for the mutated grammar set. For AMBER length (with ellipsis set or otherwise), the additional tool runs did not uncover a better solution. For AMBER examples, additional runs were performed on $\pm 5\%$ of the best solution from the hill climbing run. For the AMBER examples (with ellipsis or otherwise) option, the additional runs did not uncover a better solution.

Table 5.20 shows the additional tool runs performed for the AmbiDexter length option

AMBER			
len	len+ell	N	N+ell
(12, 171)	(9, 149)	(3320353, 158)	(8070853, 154)
(15, 171)		(3652389, 158)	(8877939, 154)
(18, 171)		(4017628, 158)	(9765733, 154)
(21, 171)		(4419391, 158)	(10742307, 154)
(13, 171)	(10, 146)	(3154335, 157)	(7667310 , 153)
(14, 170)	(11, 146)	(3469769, 158)	(8434042 , 154)
(15, 171)		(3816746, 158)	(9277446 , 154)
(16, 171)		(4198421, 158)	(10205191, 154)
(17, 171)		(4640360, 158)	(11279422, 154)
(18, 171)			
(19, 171)			
(20, 171)			
(22, 171)			

Table 5.19: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the AMBER length and the examples option for the mutated grammar set. Pair value (l, f) denote the length l sampled and its fitness value. Pair value (N, f) denote the number of examples N sampled and its fitness value.

for the mutated grammar set. For AmbiDexter length (filter set or otherwise) option, the additional runs did not uncover a better solution.

Table 5.21 shows the additional tool runs performed for the *SinBAD*'s backends – *dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*– depth option for the mutated grammar set. For the *dynamic1* and *dynamic2* the additional tool runs did not uncover a better solution. For the *dynamic3* backend, the additional runs uncovered a better solution for D=23 (ambiguities found=194). For the *dynamic4* backend, the additional runs uncovered a better solution for D=26 (ambiguities found=191).

For the additional run for $dynamic2_{rws}$ for the mutated grammar set, see Table 5.18. The additional hill climb and tool runs did not uncover a better solution.

For the main experiment, I pick the best performing run-time option for each tool for each grammar set from the fine dimensioning run. The next section narrows down the option for each tool and for each grammar set.

	AmbiDexter Length[≜]			
unf[≜]	LR0	SLR1	LALR1	LR1
(14, 162)	(14, 171)	(14, 172)	(16, 172)	(17, 82)
(17, 162)			(19, 172)	(20, 82)
(20, 162)			(22, 172)	(23, 82)
(23, 162)			(25, 172)	(26, 82)
(12, 158)	(12, 167)	(12, 168)	(14, 169)	(15, 79)
(13, 157)	(13, 169)	(13, 170)	(15, 170)	(16, 81)
(15, 158)			(17, 172)	(18, 82)
(16, 157)			(18, 172)	(19, 82)
(18, 158)			(20, 172)	(21, 82)
(19, 157)			(21, 172)	(22, 82)
(21, 159)			(23, 172)	(24, 82)
(22, 157)			(24, 172)	(25, 82)
(20, 162)			(26, 172)	(27, 82)
(24, 159)				

[≜] AmbiDexter invoked with filter not set.

Table 5.20: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the AmbiDexter length option for the mutated grammar set. Pair value (l, f) denote the length l sampled and its fitness value.

5.12 Best Performing Tool Options

My fine dimensioning of the tools and their options for each grammar set, in certain cases, uncovered multiple options and values that all performed equally well. For the main experiment, for each tool and for each grammar set, I had to pick an option and a run-time value for it. I now discuss my choice of the tool option and its run-time value for each tool in the following sections.

5.12.1 AMBER

For the Boltzmann grammar set, the best performing option was number of examples with ellipsis set, for N+ell=27593602. I use this option and its value as is for the main experiment. For the altered PL grammar set, options – length and number of examples (with ellipsis set) – performed equally well. For length, len=11, 12, and 13, and for number of examples, several values from N+ell=19404000 to N+ell=50328987 at 5% intervals, all uncovered the same number of ambiguities. For the altered PL grammar set, I chose the number of examples option with N+ell=19404000 for the main experiment.

$dynamic_n$ Depth			
dynamic1	*dynamic2*	*dynamic3*	*dynamic4*
(22, 194)	(16, 192)	(24, 193)	(25, 190)
	(19, 192)		(28, 190)
	(22, 192)		(31, 190)
(20, 191)	(14, 190)	(22, 193)	(23, 187)
(21, 191)	(15, 192)	(23, 194)♯	(24, 190)
(23, 192)♯	(17, 190)	(25, 193)	(26, 191)♯
(24, 191)	(18, 192)	(26, 192)	(27, 190)
	(20, 191)		(29, 190)
	(21, 192)		(30, 189)
	(23, 191)		(32, 189)
	(24, 191)		(33, 187)

♯ Better solution found from the additional tool runs.

Table 5.21: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the *SinBAD*'s backends – *dynamic1*, *dynamic2*, *dynamic3*, and *dynamic4*– depth option for the mutated grammar set. Pair value (D, f) denote the depth D sampled and its fitness value.

For the mutated grammar set, the length (with ellipsis unset) option uncovered the most number of ambiguities. All values sampled between 12 and 22 with the exception of 14 (which found one less ambiguity) performed well. For the main experiment, I had to pick a value for the length option. So, I looked at the results for the length option for the mutated grammar set from both the crude and the fine dimensioning run: in both cases, the distribution seems to plateau at ≈ len=12. For the main experiment, since I run my tool for much extended time limits, I felt that the distribution might marginally shift to the right, and so I went with len=18.

Table 5.22 shows the best performing run-time option and its value for AMBER for each grammar set.

Boltzmann	Altered PL	Mutated
Examples+Ellipsis (N=27593602)	Examples (N=19404000)	Length (len=18)

Table 5.22: AMBER: the best performing run-time option and its value (in brackets) for each grammar set.

5.12.2 AmbiDexter

For AmbiDexter for the Boltzmann grammar set, the option 'search by incremental length' with no filter set performed the best. I go with this option and its value for the main experiment.

For the altered PL grammar set, both the options (search by incremental length and length) performed equally well. For search by incremental length, the best performing filters were LR0, SLR1, and LALR1. For the length option, the best performing values were len=14–15 (with filter=LR0), len=14–24 (with filter=SLR1), and len=14–19 (with filter=LALR1). For the main experiment, I chose the tool option 'search by incremental length' for filter=LALR1. Since I run my tools for much extended limits for the main experiment, I felt that the most powerful filter LR1 has a good a chance of performing well, so I additionally invoked AmbiDexter for option 'search by incremental length' for filter=LR1.

For the mutated grammar set, the search by incremental length option for filters LR0 and LALR1 performed the best. The more precise a filter is, the longer it takes to prune out the unambiguous subsets of the grammar. Since I run my tools for much extended limits for the main experiment, I felt that the more powerful filter of the two – LALR1 – has as good a chance of performing well, if not better, than LR0; I chose LALR1 for the main experiment.

Table 5.23 shows the best performing run-time option and its value for AmbiDexter for each grammar set.

Boltzmann	Altered PL	Mutated
Incremental length (*ilen*+filter=None)	Incremental length (*ilen*+filter=LALR1,LR1)	Incremental Length (*ilen*+filter=LALR1)

Table 5.23: AmbiDexter: the best performing option and its value (in brackets) for each grammar set.

5.12.3 *SinBAD*'s backends

For the Boltzmann grammar set, *dynamic3* was the best performing backend for D=16. For the altered PL grammar set, all the backends except *dynamic2* found all the ambiguities. For the altered PL grammar set, I chose *dynamic3* for D=11 for the main experiment. For the mutated grammar set, $dynamic2_{rws}$ was the best performing backend for D=20 and W=0.036382.

Table 5.24 shows the best performing run-time option and its value for each grammar set.

Boltzmann	Altered PL	Mutated
dynamic3 (D=13)	*dynamic3* (D=11)	$dynamic2_{rws}$ (D=20, W=0.036382)

Table 5.24: *SinBAD*'s backends: the best performing backend and their run-time values of D and weight W (in brackets) for each grammar set.

5.13 Summary

In this chapter, I uncovered the best performing run-time option for each tool and for each grammar set. First, I presented my experimental suite describing each of my tool and that various run-time options that it supports. I then presented a novel search-based approach to explore the solution space of a tool option to uncover its best performing run-time value. The best run-time option and its value for each tool is executed on a much larger grammar corpus in the main experiment. In the following chapter, I present my main experiment.

Chapter 6

Main Experiment

This chapter presents the largest ambiguity detection tool experiment to date. The main experiment uses the best performing run-time option for each tool (as uncovered by the fine dimensioning experiment), on a much larger grammar corpus (roughly 5 times bigger than the grammar corpus used for the fine dimensioning experiment).

The first part of this chapter covers the main experiment: its methodology; results; and an examination of the strengths and the weaknesses of each ambiguity tool. The second part introduces the validation experiment and discusses the threats to validity of the main experiment.

6.1 Experiment Methodology

The main experiment uses the best performing run-time option for each tool on the Boltzmann, altered PL, and the mutated grammars. The options for AMBER, AmbiDexter and *SinBAD* are shown in Tables 5.22, 5.23, and 5.24 respectively (ACLA doesn't support additional options, so was invoked as-is). Table 5.1 shows the size of each set of grammars.

Since grammars can specify infinite languages, grammar ambiguity tools can run forever. I am therefore also interested in how long it takes each tool to give quality results. For the main experiment, I therefore run each tool for 10, 30, 60, and 120 seconds, enforcing the limit with the `timeout` tool. In most cases, tools have clearly reached a point of diminishing returns by the upper time limit; in a few cases where manual inspection showed that this point had not been reached, I made additional runs with time limits of 300 and 600 seconds. All the data from my main experiment are available for download from: `https://figshare.com/s/d6ef0a1a3de26f6ca218`.

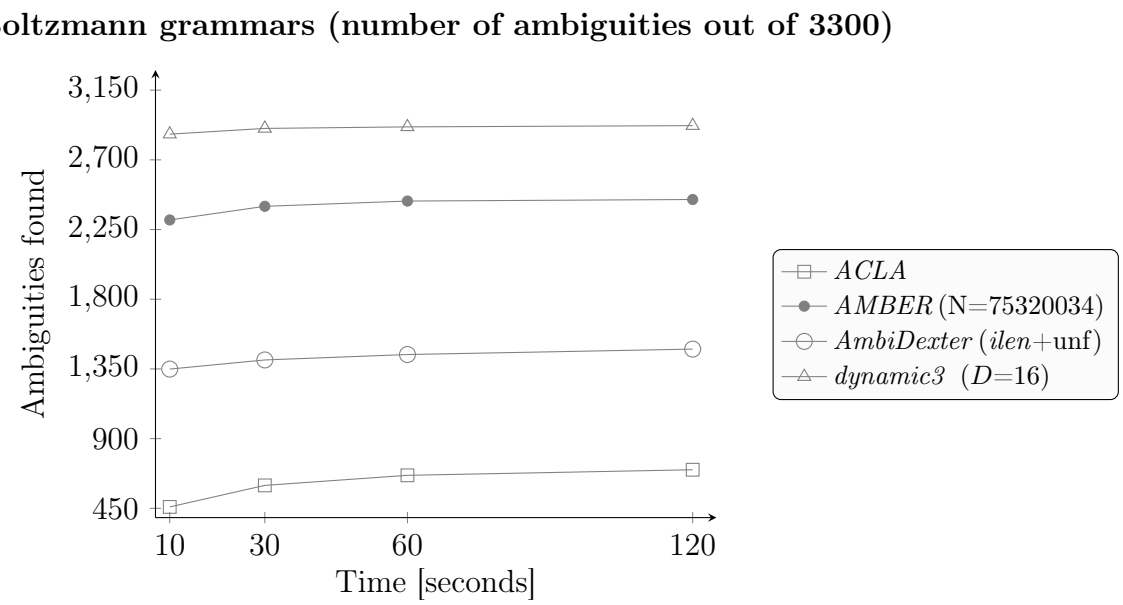

Figure 6.1: Number of ambiguities found for Boltzmann sampled grammars (3300 grammars).

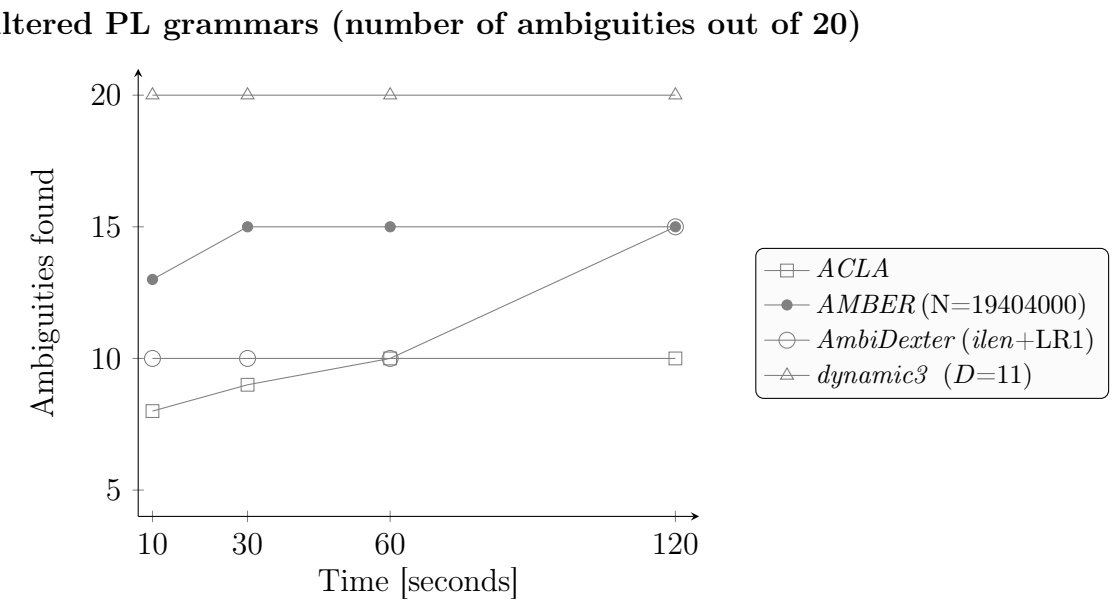

Figure 6.2: Number of ambiguities found for altered PL grammars (20 grammars).

6.2 Results

6.2.1 Tool Independent Analysis

Figures 6.1, 6.2, and 6.3 show the results of the main experiment for each grammar set. The results from the main experiment show that four of my grammar sets are highly ambiguous: Boltzmann grammars (88%), 'add empty alternative' mutated grammars (74%),

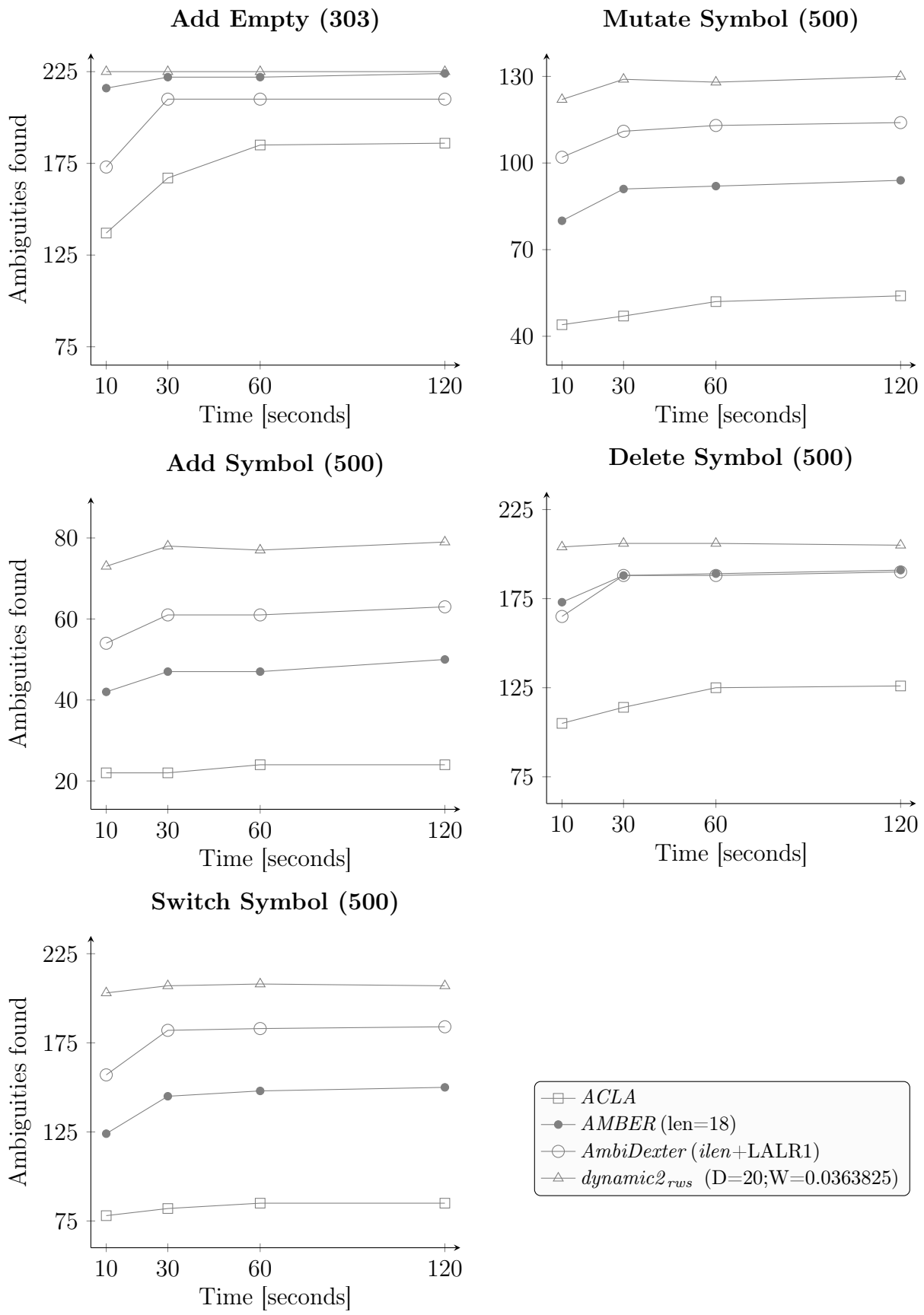

Figure 6.3: Number of ambiguities found for mutated grammars. The number in brackets in the title of each graph indicate the total number of grammars evaluated for that mutation type.

	ACLA		AMBER		AmbiDexter		*SinBAD*	
	Sen	Amb	Sen	Amb	Sen	Amb	Sen	Amb
Boltzmann	20	20	49	37	62	41	12518977	754658
Altered PL	11	11	37	19	15	15	768	402
Mutated	14	14	18	18	8214	26	1597252	7183

Table 6.1: Maximum sentence ('Sen') and ambiguous fragment ('Amb') length (defined as the number of tokens) detected by each tool using their best performing options when run for 120s.

'delete symbol' mutated grammars (41%), and 'switch symbol' mutated grammars (41%). Manual observation of these highly ambiguous grammar sets shows that many grammars contained in these sets have multiple ambiguities. A grammar has multiple ambiguity if it has more than one ambiguous subset. 36% of Boltzmann grammars contained 2 or more ambiguities per grammar. For mutated grammars the figures are: 61% for 'add empty alternative'; 57% for 'delete symbol'; and 43% for 'switch symbol'. In analysing the results, only in one case I had to rely on the results from the fine dimensioning experiment. In case of AmbiDexter, the best performing option (as uncovered by the fine dimensioning) for Boltzmann grammars was the unfiltered option. Since for the Boltzmann grammars for the main experiment, unfiltered option was used, to collect data on the effectiveness of AmbiDexter's filtering technique on Boltzmann grammars, the results from the fine dimensioning experiment were used.

6.2.2 Tool Overview

Table 6.1 shows the sentence and the ambiguous fragment length detected by each tool using its best performing run-time option for each grammar set. As is evident from the table, the *SinBAD*'s backends generate much longer sentences and ambiguous fragments compared to the other tools. The tools are ordered by the number of ambiguities found across all grammar sets. Out of 5623 grammars explored, the number of ambiguities found by each tool (from lowest to highest) are: ACLA (1213), AmbiDexter (2327), AMBER (3168), *SinBAD*'s backends (3787). In the rest of this section, I explore what the results mean for each tool starting with the tool ACLA.

6.2.3 ACLA

ACLA's approach to ambiguity detection is based on two linguistic properties: vertical and horizontal ambiguity. Vertical ambiguity means that during the parsing of a string, there is a choice between the alternatives of a non-terminal. Horizontal ambiguity means

that during the parsing a string according to an alternative, there is a choice in how it can be parsed. ACLA detects ambiguity in a grammar by iterating through its non-terminals, and checking their language for vertically or horizontally ambiguous strings. Given a grammar, ACLA will report it to be ambiguous, unambiguous, or possibly ambiguous (that is, it is unsure if the grammar is ambiguous). If a grammar is found to be ambiguous, ACLA reports the shortest possible example of vertical or horizontal ambiguity.

My results show that ACLA was unsure whether the grammar was ambiguous or otherwise on 75% and 62% of the Boltzmann and the mutated grammars respectively. For the altered PL grammars, on 7 of the 20 grammars, ACLA was unsure whether the grammar was ambiguous or otherwise. For unambiguity, ACLA reported one Boltzmann and 3 mutated grammars to be unambiguous. For Boltzmann grammars, ACLA generated much longer sentences, and so it uncovered marginally longer ambiguous fragment too (see Table 6.1). In most cases, where the ambiguous subset is deeply nested, ACLA is unsure if the grammar is ambiguous.

For Boltzmann grammars, the data (see Figure 6.1) suggested that, given additional time, ACLA might uncover further ambiguities. When run for 300s, ACLA found additional 4% ambiguities in this set of grammars; running for 600s found only an additional 4. No additional runs were performed on the altered PL or mutated grammars as these had already reached a clear point of diminishing returns in the main experiment.

6.2.4 AmbiDexter

AmbiDexter detects ambiguity by first pruning out the unambiguous subsets of a grammar and then exhaustively searching the remaining search space. AmbiDexter is effective for altered and mutated PL grammars, but is less effective for Boltzmann grammars. For Boltzmann and altered PL grammars, the results (see Figures 6.1 and 6.2) for the experiment suggested that given additional time, AmbiDexter might uncover further ambiguities. So, for both Boltzmann and altered PL grammars, I ran AmbiDexter for an extended time limit of 300s. For Boltzmann grammars, AmbiDexter uncovered an additional 3% ambiguities. Since the additional number of ambiguities found was significant, I further ran AmbiDexter for Boltzmann grammars for a much extended time limit of 600s. AmbiDexter uncovered only 1.7% additional ambiguities. Since the number of additional ambiguities found was much lower, I didn't explore further. For altered PL grammars, AmbiDexter found the one remaining ambiguity.

AmbiDexter does well on PL grammars for two reasons: first, filtering of unambiguous fragments was very effective on PL grammars; and second, PL grammars contain short ambiguous subsets. I now explain these two aspects in detail.

6.2.4.1 Filtering Performance

AmbiDexter's filtering technique was very effective on the PL grammars. For mutated grammars, where LALR1 was the best performing filter, on average, 74% of the grammar rules were identified to be unambiguous. For altered PL grammars, where LR1 was the best performing filter, 54% of the grammars rules were identified to be unambiguous. Therefore, in both cases, AmbiDexter operated on a much smaller state space, and so performed well. For Boltzmann grammars, since the best performing option was 'search by incremental length' with no filter set, AmbiDexter's exhaustive search had to operate on the whole of the grammar space, and so struggled. The corollary to effective filtering in the case of mutated PL grammars is that there were significant number of cases where all of a grammars' rules were filtered out (i.e. the grammar is unambiguous). For mutated grammars, 58% of the grammars were found to be unambiguous.

To give the reader a rough idea on why AmbiDexter's filtering technique was less effective on Boltzmann grammars, I refer to the filtering performance of the AmbiDexters' run from the fine dimension experiment. Based on the best performing filter – SLR1 – for Boltzmann grammars, only 17% of the grammars' rules were identified to be unambiguous. As a result, the percentage of the grammars that were found to be unambiguous was only 2%. This ties in with the observation that a Boltzmann grammar contains multiple ambiguities (see Section 6.2.1) spread across its grammar rules which then limits the portion of the grammar that can be proven to be unambiguous.

There was noticeable a difference in the filter execution time between the grammar sets. To analyse filter execution time for Boltzmann grammar, the data from the fine dimensioning experiment was used. The average time (in seconds) taken to filter out unambiguous subsets of a grammar for Boltzmann, mutated and altered PL grammars were 0.9, 2.1 and 4 respectively. The time taken to filter out the unambiguous subsets of a grammar depends on the precision of the filter applied. The higher the precision of the filter, the longer it takes to prune out the unambiguous subsets. Therefore, in the case of altered and mutated PL grammars, where the respective filters applied were LALR1 and LR1, relatively more powerful filters than SLR1, the filter execution time was higher.

6.2.4.2 Length of Ambiguous Fragments

Mutated PL grammars contain marginally shorter ambiguous fragments (mean 6.4) than Boltzmann grammars (mean 7.5), and so AmbiDexter was quick to find them. For altered PL grammars, the mean ambiguous fragment length was 9.

There was marginal difference in the length of the sentences generated between grammar sets. The mean sentence length for Boltzmann, altered PL and mutated grammars were

11, 10 and 7.5[1] respectively.

6.2.5 AMBER

AMBER performs extremely well on the Boltzmann grammars, but less well on manually altered or mutated grammars. AMBER uses an exhaustive approach to ambiguity detection, systematically enumerating strings for a given grammar. There are two possible reasons why AMBER does well on Boltzmann grammars. First, these grammars contain multiple ambiguities (see Section 6.2.1), and so AMBER has a greater chance of finding one of them. Second, the ambiguous subsets found are easily reachable, in the sense that they are referenced from very near the start of the grammar. For instance, in the case of the altered PL grammar "Java.2", the ambiguous subset originates from the rule `compilation_unit` that is close to the start rule, and so AMBER is quick to find ambiguity. In the case of Pascal.2, the ambiguous subset originates from within an expression rule set (`term`) that is, frequently referenced, and so AMBER is quick to find it. However, in the case of "Java.1", where the ambiguous subset is nested deep within the `cast_expression` rule and is hard to reach, AMBER does not find this ambiguity within our time limits.

6.2.6 *SinBAD*'s Backends

As Table 6.1 shows *SinBAD*'s dynamic backends generates much longer sentences than other tools. This allows it to find longer and/or deeply nested ambiguous subsets than other tools. The average sentence and ambiguous fragment length for each grammar set are: 134257 and 614 for Boltzmann grammars; 151 and 60 for altered PL grammars; and 825 and 134 for the mutated grammars. Since *SinBAD*'s backends use a non-deterministic approach for generating sentences, there can be significant variation in the sentence and ambiguous fragment length from run to run. I now explain why *SinBAD*'s non-deterministic breadth-based approach to ambiguity detection performs better than some of the deterministic depth-based approaches using an example.

6.2.6.1 Depth. vs. Breadth – a Comparison

As an example, I use the altered PL grammar 'C.2' from the experimental suite. For the example grammar, none of the depth-based approaches detected ambiguity for a time limit of t=120s, whereas all of the *SinBAD*'s backends did. The ambiguous fragment length for my example grammar was manually found to be 12. For my example grammar, AMBER

[1] In calculating the average, I had to exclude two of the grammars with exceedingly high sentence lengths (2143 and 8214), as including them would not be a true reflection of the sentence length of the vast proportion of the grammars.

with length=12 for a time limit of 2 hours explored just over 1.2×10^9 examples but didn't find any ambiguity. Running AmbiDexter with length=12 (for unfiltered version) didn't find ambiguity even after a day. The relevant parts of the grammar that contribute to ambiguity is shown below:

```
1    file: external_definition | ... ;
2    external_definition: function_definition | ... ;
3    function_definition: declarator function_body | ... ;
4    declarator: declarator2 | ... ;
5    declarator2: declarator2 '(' parameter_identifier_list ')'
6      | declarator2 '(' parameter_type_list ')'
7      | declarator2 '(' ')'
8      | declarator2 '[' constant_expr ']'
9      | declarator2 '[' ']'
10     | '(' declarator ')'
11     | identifier ;
12   function_body: compound_statement | declaration_list compound_statement ;
13   compound_statement: '{' declaration_list statement_list '}'
14     | '{' declaration_list '}'
15     | '{' statement_list '}'
16     | '{' '}' ;
17   statement_list: statement_list statement | statement ;
18   statement: jump_statement | selection_statement | compound_statement | ... ;
19   selection_statement: IF '(' expr ')' statement
20     | IF '(' expr ')' statement ELSE statement
21     ...
```

The grammar contains the classic nested *if else* ambiguity. The source of the ambiguity is located deep within the grammar, in the first and the second alternative of the '`selection_statement`' rule (lines 19 and 20). The path to the non-terminal '`selection_statement`' from the start symbol '`file`' is as follows: `file` → `external_definition` → `function_definition` → `function_body` → `compound_statement` → `statement_list` → `statement` → `selection_statement`. When looking from the top level of the grammar, the ambiguity is located in the section of the grammar that derives from the non-terminal '`function_body`' referenced in the first alternative of rule '`function_definition`' (line 3).

It is evident from the grammar that the ambiguous subset is located at a fair 'distance' from the start rule of the grammar. In a depth-based approach, when the sentence generator starts deriving from the start rule of the grammar, it explores each of the non-terminals referenced systematically. Each non-terminal is explored by exhaustively searching each of its alternatives. For the example grammar, when the rule '`function_definition`' is entered and its first alternative is being derived, the leftmost non-terminal `declarator` is explored first by exhaustively searching each of its alternatives systematically. Since '`declarator`' is recursive (`declarator` ⇒ `declarator2` ⇒ `declarator`) and several of the non-terminals that it references describe an infinite language, a depth-based search gets stuck in exploring the grammar subsets related to the non-terminal '`declarator`'. Within reasonable time limits, the search doesn't get the opportunity to explore the grammar subsets reachable from the non-terminal '`function_body`', and so misses out on the ambiguity.

SinBAD's backends use a non-deterministic breadth-based approach to ambiguity detection. Whereas the extant depth-based approaches focus on a specific subset of the grammar, my breadth-based approach aims for grammar coverage. Given a grammar, *SinBAD*'s sentence generator builds a sentence by picking alternatives randomly, which allows it to explore as much of a grammars' rules as possible without focussing on any specific subset in exhaustive detail. When the threshold depth D is reached, favouritism is applied to encourage sentence termination.

For my example grammar, the sentence generator starts generating a sentence from the start rule of the grammar by picking alternatives randomly. When the rule '`function_definition`' is entered and its first alternative is (randomly) picked, the sentence generator explores the rule '`declarator`' and the grammar subsets reachable from it, for a while. In exploring the rule '`declarator`', the sentence generator recurses into the grammar deep enough but not too deep and when the threshold depth D is reached, favouritism ensures that '`declarator`' is quickly derived. The sentence generator then explores the rule '`function_body`' and the grammar subsets reachable from it by picking alternatives randomly. Since the rule '`selection_statement`' containing the ambiguity is easily accessible (i. e. several paths from '`function_body`' lead to '`selection_statement`'), ambiguity is found fairly quickly. Thus by simply exploring a grammar in breadth through a combination of random selection and 'quickly deriving' alternatives, *SinBAD*'s backends are able to detect ambiguity much quicker than other depth-based approaches.

6.2.6.2 Ambiguities that *SinBAD* Backends Didn't Find

Although *SinBAD* generally finds larger numbers of ambiguous grammars, it is not always a pure superset. For example, for the Boltzmann grammars, AMBER uncovered 2, and AmbiDexter 1, grammar that *dynamic3* did not. For 3 mutated grammars that AmbiDexter found as ambiguous, the best performing backend, $dynamic2_{rws}$, failed to find any. In case of ACLA, *SinBAD*'s backends found all of the grammars to be ambiguous that ACLA detected to be ambiguous. I now provide a brief explanation on why *SinBAD*'s backends failed to detect ambiguity in some of the grammars that other tools found to be ambiguous.

Of the 2 Boltzmann grammars that AMBER found to be ambiguous, on one of them *dynamic3* generated 232 sentences but didn't uncover any ambiguity. For this grammar, AMBER generated over 40 million examples in order to uncover ambiguity. For the other, *dynamic3* got stuck in generating an extremely long sentence. *dynamic3* generated a sentence with 10,666,503 words, spending 6 seconds on sentence generation and 94 seconds on parsing the sentence. For this grammar, AMBER found the ambiguity in just over 700,000 examples. Of the one Boltzmann grammar that AmbiDexter found to be ambiguous, *dynamic3* generated 3609 sentences but didn't uncover any ambiguity. For

this grammar, the ambiguous subset was by no means long (len=17) and deeply nested but *dynamic3* was unable to detect it.

For the 3 mutated grammars that AmbiDexter found to be ambiguous and $dynamic2_{rws}$ failed to detect any, in all three cases, $dynamic2_{rws}$ generated over 2K sentences but failed to detect ambiguity. In all three cases, the ambiguous subsets were short (len < 9) but deeply nested.

6.3 Validation Experiment

In order to ensure that the results of Figures 6.1, 6.2, 6.3 scale to larger sets of grammars, I used the best performing backend for the Boltzmann and the mutated grammars to perform a validation experiment on a much larger set of grammars (see Table 5.1). The number of ambiguities found for 120 seconds were 88% (Boltzmann) and 75%, 24%, 14%, 37% and 39% (for mutated types: add empty alternative, mutate symbol, add symbol delete symbol and switch symbol respectively). The proportion of ambiguities found in our validation experiment is close to the number of ambiguities found in the main experiment (see Figures 6.1 and 6.3). All the data involved are available from: `https://figshare.com/s/55805aaa7e56f8e36d8d`.

6.4 Validating the Hypotheses

In Section 1.2, I stated two hypotheses which informed my work. In this section, I revisit the hypotheses in the light of my results.

Hypothesis H1 postulates that "covering a grammar in breadth is more likely to uncover ambiguity than covering it in depth." The non-deterministic approach used by *SinBAD*'s backends tend to naturally generate sentences which cover much larger portions of a grammar than previous approaches. It is therefore more successful at uncovering ambiguity against my grammar corpus than other tools. Although non-determinism clearly plays its part, I believe that *SinBAD*'s backends coverage is key and strongly validates hypothesis H1.

Hypothesis H2 postulates that "PL grammars are only a small step away from being ambiguous." The mutated grammars are my attempt to explore this hypothesis and as the validation experiment shows, just over a third of mutations to real PL grammars result in $dynamic2_{rws}$'s detecting ambiguity. This proportion is a lower-bound: it is possible that there is further ambiguity in the mutated grammars that $dynamic2_{rws}$ (and, indeed, any other tool) does not discover. I consider this validation of hypothesis H2.

6.5 Threats to Validity

The most obvious threat to the validity of my results are the grammars used.

In a previous experiment [39] I used a hand-written generator to create random grammars. In this thesis, I created a Boltzmann sampler to reduce the chances of bias in my hand-written generator. Interestingly, this made relatively little difference to the number of ambiguous grammars I found. However, it is impractical to generate completely arbitrary grammars, since they have no size limit. My Boltzmann specification is therefore geared towards generating grammars which are "somewhat PL like". It is possible that it still produces overly biased grammars, particularly as I am forced to apply filters to remove some grammars I consider irrelevant or unrepresentative. However, I believe that, overall, it is more trustworthy than any previous random grammar generator.

The mutated grammars are also a potential threat to validity as I might have chosen unrepresentative grammars as a base. Since they come from an external source, I have some level of confidence in them.

Finally, there are two other threats to validity. The first is my use of the dimensioning experiments to determine a reasonable set of run-time options for the various tools used. The hill climbing technique could have continually got stuck in a local maxima. However, the sheer quantity of results that I have got from my hill climbing run lessens the chances of this dominating my results. The second is that it is also possible that the grammars used in the fine dimensioning experiment were unrepresentative of those used in the main experiment, though simple measurements suggest this is unlikely. The percentage of ambiguous Boltzmann grammars for the fine dimensioning and main experiments are identical (88% for both). The percentage of ambiguous mutated grammars, though not identical, were all similar: add empty alternative (70% and 74% for fine dimensioning and main experiments respectively), mutate symbol (29% and 26%), add symbol (13% and 15%), delete symbol (47% and 41%), and switch symbol (39% and 41$).

6.6 Summary

In this chapter, I presented the results of my main experiment. I first outlined the setup of the main experiment. I then presented the results of the main experiment and discussed the strengths and the weaknesses of each tool in detail. To be reasonably sure that the grammars I picked for the main experiment are not biased in any way, I ran a validation experiment using the best performing backends on a much larger grammar corpus. The results from the validation experiment confirmed that the proportion of the ambiguities found on a larger set of grammars matches closely to the proportion of ambiguities found by the main experiment.

Although *SinBAD*'s backends are quick to detect ambiguity, the ambiguous fragments that they uncover can often be exceedingly long (see Table 6.1). Long ambiguous fragments invariably means deep and nested parse trees that are quite hard to understand. Clearly a short ambiguous string must exist, since other tools in my experimental suite are able to find them. There are a few possible solutions as to how to minimise the ambiguous fragment before it is presented to the end-user. One possibility is to inspect the ambiguous parses generated by a backend and pick the rules that contribute to ambiguity to create a minimised grammar. The minimised grammar can then be fed back to the backend that generated it or it can be fed to an external ambiguity detection tool for further minimisation. I explore the first possibility in the following chapter.

Chapter 7

Minimising Grammar Ambiguity

This chapter presents a novel minimisation technique for grammars guided by ambiguity detection. As noted in Chapter 6, *SinBAD*'s backends often generates a long and a deeply nested ambiguous fragment that is quite hard to understand. In order to present the user with a shorter version of the ambiguity, I created a grammar minimiser. Since grammars can contain infinitely long ambiguous subsets, it may not be always possible to minimise a grammar. Therefore, the purpose of the minimiser is to identify as small an ambiguous subset of the grammar as possible in the hope of discovering a short ambiguous string. My results show that the minimiser does reasonably well at uncovering small ambiguous subsets for grammars from my experimental corpus.

This chapter comes in two parts. The first part of this chapter introduces *minimiser1*, a novel approach for minimising a grammar ambiguity. I then demonstrate grammar minimisation using an example. The second part presents an experiment which evaluates the effectiveness of my minimisation technique on a large grammar corpus from my experimental suite.

7.1 Definitions

Before presenting my minimisation algorithm, I first introduce some brief definitions and notations. The definitions from Section 3.4 are re-used and some additional notations are defined.

A minimised grammar is denoted as $G_m = \langle N_m, T_m, P_m, S_m \rangle$ where N_m is the set of non-terminals, T_m is the set of terminals, P_m is the set of production rules over $N_m \times (N_m \cup T_m)^*$ and S_m is the start non-terminal of the grammar. For a given grammar G, backend b, threshold depth D, and weight W, the function $\mathsf{dynamic}(G, b, D, W)$ invokes the dynamic backend b on grammar G for a threshold depth D and a probabilistic weight W, and returns any ambiguity found. Although ambiguities can have infinite parses, for the purposes of this section, I define ambiguity to be a pair of parse trees. Given an ambiguity

Algorithm 20 Algorithm to minimise a grammar using the *minimiser1* minimiser.

```
1: function CFG-MINIMISER(G, b, D, W=NONE)
2:     G_c ← G
3:     ambs_c ← NONE
4:     while TIMEOUT NOT REACHED do
5:         parses ← dynamic(G_c, b, D, W)
6:         if parses ≠ NONE then
7:             G_c ← concfg(parses)
8:             ambs_c ← ambstr(parses)
9:             write(G_c, ambs_c)
10:        end if
11:    end while
12: end function
```

p, the function ambstr(p) returns the string by concatenating the terminals from the leaf nodes of its parse tree in a depth-first fashion, and the function concfg(p) returns a grammar G constructed from p's parse trees (i. e. the constructed grammar is a subset of the original grammar). Given a grammar G and a string s, the function write(G, s) writes the grammar G and the string s to disk.

7.2 The *minimiser1* Minimiser

minimiser1 minimises a grammar by ambiguity detection using *SinBAD*. *minimiser1* takes an iterative approach towards grammar minimisation, whereby in each iteration the given grammar is explored for ambiguity in the hope of detecting a smaller ambiguous subset. An iteration works as follows. The given grammar is explored for ambiguity using the given *SinBAD* backend. On finding ambiguity, from the resulting parse trees, I identify the (non-strict) subset of the grammar that gave rise to ambiguity. I then extract that subset of the grammar and re-run the given *SinBAD* backend hoping to find a smaller ambiguous subset. If an iteration doesn't result in a smaller grammar, I still continue to minimise in the hope that future iterations might uncover a smaller ambiguous subset. *minimiser1* is externally halted on reaching a certain time limit.

Algorithm 20 describes how a minimised grammar is constructed using the *minimiser1* minimiser. The function CFG-MINIMISER is initialised with grammar G that is to be minimised, the *SinBAD* backend b to be invoked, the threshold depth D, and the probabilistic weight W.

G_c tracks the current minimised grammar, and is initialised to G. $ambs_c$ tracks the current ambiguous fragment, and is initialised to NONE. The function dynamic is invoked to launch the backend b on G_c for threshold depth D and probabilistic weight W, and

the parse trees returned are recorded in *parses* (line 5). The function concfg is invoked on *parses* to construct the minimised grammar (line 7). The function ambstr is invoked on the ambiguous parses *parses* to construct the ambiguous fragment (line 8). At the end of each iteration, the current minimised grammar G_c and the ambiguous fragment $ambs_c$ are written to the disk (line 9). The heuristic continues to iterate until a certain time limit is reached and is externally halted.

To be sure that my grammar minimiser *minimiser1* has not altered the ambiguity that was contained in the original grammar, I perform a sanity check: the ambiguous fragment generated by the minimisation is verified against the original grammar. I now outline my approach for sanity checking *minimiser1*'s minimisation approach.

7.2.1 Sanity Checking *minimiser1*'s Grammar Minimisation

To sanity check *minimiser1*'s grammar minimisation, I re-use the ambiguous fragment that it uncovered by minimisation during sentence generation. To do so, I make a small change to the sentence generator that each of my backends use. The revised sentence generator works as follows. Given a grammar, the sentence generator continues to pick alternatives randomly. When the sentence generator recurses beyond a certain threshold depth D, alternatives are favoured. The favouring of alternatives is specific to each backend and this had not changed. When a rule is entered, if the alternative picked contains the sequence of symbols that contribute to ambiguity, then I don't recurse into them. Instead I derive them using the ambiguous fragment uncovered by the minimisation. If the generated sentence is found to be ambiguous, then I am certain that the minimisation has not altered the original ambiguity. The algorithm for the modified sentence generator for one of the *SinBAD*'s backend (*dynamic3*) is shown in Appendix D.

7.3 Minimisation using *minimiser1* – an Example

To show how minimisation operates, I use the Pascal grammar 'Pascal.3' from the experimental corpus. This grammar contains the classic nested *if-else* ambiguity [25]. *minimiser1* was invoked on the Pascal grammar using the *dynamic3* backend for D=10 for a total minimisation time limit of 5 seconds.

Table 7.1 shows an example run of the minimiser on the Pascal grammar. 'Sen' refers to the sentence generated from the minimised grammar, and 'Amb' refers to the ambiguous fragment contained within Sen. Sen is a necessary input to parse Amb relative to the grammar. Note that while iterations never increase the Grammar size (though they may reduce it), Sen and Amb may increase from one iteration to the next due to *minimiser1*'s non-deterministic nature (as this may suggest, there is no relation between the Sen and Amb from one iteration to the next). The minimised grammar at iteration N=2 is shown

Iteration	Grammar	Sen	Amb
1	79	101	20
2	10	14	11
3	8	223	16
4	8	77	15
5	8	142	13
6	7	53	9
7	7	101	9
	...		
22	7	21	9

Table 7.1: An example run of the minimiser showing, for each iteration, the grammar size ('Grammar'), sentence length ('Sen'), and ambiguous fragment length ('Amb' where Amb $\leq$ Sen) generated by *minimiser1*. The first iteration corresponds to the unaltered "Pascal.3" grammar which is gradually minimised over multiple iterations.

in Listing 7.1. For the minimised grammar, the uppercase symbols are terminals and the rest of the symbols are non-terminals.

```
1  root: statement;
2  statement: IF expression THEN statement ELSE statement
3           | IF expression THEN statement
4           | REPEAT statements UNTIL expression
5           | ;
6  statements: statement;
7  expression: simple_expr | simple_expr relational_op simple_expr;
8  simple_expr: simple_expr add_op term | '-' term | term;
9  relational_op: IN;
10 add_op: OR;
11 term: factor;
12 factor: unsigned_lit;
13 unsigned_lit: NIL;
```

Listing 7.1: The minimised Pascal grammar at N=2.

After further 20 iterations, at N=22, the minimised grammar looks much simpler (see Listing 7.2) containing just 7 rules and 9 alternatives. At the end of the time limit, *minimiser1* had reduced the grammar by 91% and the ambiguous fragment by 55%.

```
1 root: statement;
2 statement: IF expression THEN statement ELSE statement
3           | IF expression THEN statement
4           | ;
5 expression: simple_expr;
6 simple_expr: '-' term;
7 term: factor;
8 factor: unsigned_lit;
9 unsigned_lit: NIL;
```

Listing 7.2: The minimised Pascal grammar at N=22.

7.4 Evaluating *minimiser1* – Minimiser Experiment

The aim of my minimiser experiment is to evaluate the effectiveness of the *minimiser1* minimiser on grammars from my experimental suite. To evaluate *minimiser1*, I use the grammar corpus from the main experiment (see Table 5.1). For each grammar, the run is split into two parts: the first part minimises the grammar; and the second part verifies the ambiguous fragment generated from the minimisation.

7.4.1 Run-Time Values

Evaluating *minimiser1* requires various run-time values to be set. My choice of the run-time values for *minimiser1* is based on the results from my main experiment (see Section 6.2). To evaluate *minimiser1*, the various run-time parameters that need setting include: backend b for generating a sentence; threshold depth D and the probabilistic weight W for occasionally picking an alternative other than the low scoring alternative; and a maximum time limit t to terminate *minimiser1*. For each grammar set, the best performing backend and the best performing value of D and W from the main experiment is chosen. The best performing backend for Boltzmann and altered PL grammars were *dynamic3* (for D=16) and *dynamic3* (for D=11) respectively. For mutated PL grammars, the best performing backend was $dynamic2_{rws}$ (D=20, W=0.036382).

The results from my main experiment revealed that my backends uncover reasonably good number of ambiguities within a time limit of 10 seconds. For the minimisation, I give it an additional 20 seconds, and so for the minimiser experiment I set t=30. For verification, I invoke the modified sentence generator for *dynamic3* for the Boltzmann and the altered PL grammars, and $dynamic2_{rws}$ for the mutated PL grammars for a time limit of 30 seconds. The results of the minimiser experiment can be downloaded from `https://figshare.com/s/f8d177a041dfa0e902ff`.

I now share the results from the minimiser experiment.

	Boltzmann	Altered PL	Mutated
Grammar	63%	89%	87%
Sen	99%	93%	98%
Amb	0%	0%	54%

Table 7.2: Median percentage decrease in grammar size ('Grammar'), sentence length ('Sen'), and ambiguous fragment length ('Amb' where Amb $\leq$ Sen) detected by each tool using their best performing options when run for 30s. Grammar size is defined as the number of rules. A sentences' length is defined as the number of tokens it contains. An ambiguous fragment length is defined as the number of tokens that make up the ambiguity.

	Boltzmann	Altered PL	Mutated
Decrease	44%	45%	69%
No change	36%	55%	23%
Increase	20%	-	8%

Table 7.3: Effectiveness of *minimiser1* in reducing ambiguous fragment length. Proportion of grammars that resulted in reduced ambiguous fragment length ('Decrease'), no change ('No change'), and increased ambiguous fragment length ('Increase') for each grammar sets.

7.4.2 Minimiser Experiment – Results

Table 7.2 shows the median percentage decrease in grammar size, sentence and ambiguous fragment length for each grammar set. Table 7.3 shows the effectiveness of *minimiser1* in reducing the ambiguous fragment length for each grammar sets. For the grammar size, the median percentage decrease refers to the difference in the number of rules between the original and the final minimised grammar. For the sentence length, the median percentage decrease refers to the difference in the length of the sentence that resulted in ambiguity between the original and the final minimised grammar. For a given sentence s, the ambiguous fragment refers to the subset of tokens from s that contributes to ambiguity. For the ambiguous fragment length, the median percentage decrease refers to the difference in the length of the ambiguous fragment uncovered between the original and the final minimised grammar. I now discuss my results in detail.

7.4.2.1 Grammar Size

My results showed that *minimiser1* was quite effective in minimising grammars across all my grammar sets (see Table 7.2). For the PL grammars, the minimisation was marginally

better. The minimised grammars across the grammar sets were quite slim containing just an alternative per rule. The median number of alternatives to the number of rules ratio were 1.13 for Boltzmann grammars and 1.2 for (altered and mutated) PL grammars. My results show that most of the grammar minimisation occurs during the first few iterations of a run. That is, we reach a point of diminishing returns fairly soon. Across the grammar sets, we reach a point of diminishing returns at N=5.

7.4.2.2 Sentence Length

minimiser1 was also quite effective in reducing the sentence length across all grammar sets (see Table 7.2). In a small number of cases, *minimiser1* generated relatively longer sentences on the minimised grammars. The percentage of grammars for which *minimiser1* generated longer sentences were 3.4% and 3.01% for Boltzmann and mutated PL grammars respectively. For altered PL grammars, *minimiser1* generated a longer sentence for just one of the grammars. For one of the Boltzmann grammars (worse case), the sentence was 43 times longer. These figures were 1.15 and 37 for altered and mutated PL grammars respectively. I shall cover the reason for the increased length shortly.

7.4.2.3 Ambiguous Fragment Length

minimiser1 was much more effective in uncovering shorter ambiguous fragments on mutated PL grammars than on Boltzmann and altered PL grammars (see Table 7.2). For the mutated grammar set, minimisation resulted in a shorter ambiguous fragment for a large proportion of the grammars (see Table 7.3). Only in a small proportion of the cases did the minimisation result in a longer ambiguous fragment. In case of both Boltzmann and altered PL grammar sets, minimisation resulted in a shorter ambiguous fragment for a smaller proportion of the grammars. For the Boltzmann grammar set, for a sizeable proportion of the grammars, minimisation resulted in longer ambiguous fragments. For the altered PL grammar set, for just over half the number of grammars, minimisation didn't result in a change to the size of the ambiguous fragments. A worse case example where the minimisation resulted in a longer ambiguous fragment is: 440 times longer (for Boltzmann grammar set); and 17.5 times longer (for mutated PL grammar set).

7.4.2.4 Longer Sentences and Ambiguous Fragments

My experiment revealed that the minimisation resulted in longer sentences and ambiguous fragments for certain grammars. During minimisation, I invoke the best performing *SinBAD* backend on the minimised grammar with the same value of D as the one used for the original grammar. In some cases, where a large proportion of the rules in the minimised grammar form a recursive loop, this results in *SinBAD*'s backend to generate

exceedingly long sentences. Long sentences invariably lead to long ambiguous fragments. I now explain for an example grammar[1] from my experimental corpus for which *minimiser1* generates, more often than not, long sentences from its minimised version. The minimised version for my example grammar is shown below:

```
root : FL;
FL : DWAB RRV;
DWAB : IKT;
IKT : | TK_GNEV;
RRV : LXMHF LQEY | TK_PTTLP;
LXMHF : DWAB TK_UY TK_EJQA RRV;
LQEY : TK_UY TK_UY JX RRV TK_PTTLP TK_NUJSX;
JX : TK_PTTLP;
```

In the above grammar, for rule `RRV`, each of the non-terminals from the first alternative '`LXMHF LQEY`' is recursive: non-terminal `LXMHF` forms a recursion loop (`RRV` → `LXMHF` → `RRV`); and non-terminal `LQEY` forms a recursion loop (`RRV` → `LQEY` → `RRV`).

During sentence generation, when the depth of recursion $d<D$, for rule '`RRV`', the random selection picks the first (recursive) alternative roughly half the number of times. When the depth of recursion has reached beyond the threshold depth D, favouritism picks the second alternative, allowing the sentence generation to unwind. When $d<D$ again, the random selection of the recursive alternative continues until $d>D$, and favouritism is triggered. Sentence generation terminates when the non-recursive is picked in succession by chance. The picking of the recursive alternative in half the number of cases during random selection leads to long sentences, and invariably long ambiguous fragments too. One possible solution to reduce the length of the sentences generated by the minimised grammar is to invoke them with a lower value of D than the one used for the original version. This will allow the sentence generation to apply favouritism much earlier thereby encouraging sentence termination.

7.5 Summary

In this chapter, I presented *minimiser1*, a novel search-based minimiser for grammars guided by ambiguity detection. I then presented my experimental evaluation of *minimiser1* on a large grammar corpus to measure its effectiveness in grammar minimisation. My results show that *minimiser1* was quite effective at minimising both the ambiguous input and the subset of the grammar identified as containing the ambiguity.

[1] `https://github.com/nvasudevan/experiment/blob/master/grammars/boltzcfg/12/34.acc`

Chapter 8

Conclusions

8.1 Conclusions

In this thesis, I introduced the concept of a search-based approach to CFG ambiguity detection with the *SinBAD* tool and its heuristic driven backends. Using the largest grammar corpus to date, I showed that *SinBAD*'s backends can detect a larger number of ambiguities than previous approaches.

The key to *SinBAD*'s success is the use of non-determinism in the backends, which has several surprising consequences. It freed me from having to design many complex heuristics. The only additional requirement is the need to terminate the sentence generator. By applying a series of gradually better heuristics that favour easily derived grammar subsets, I showed that the sentence generator is able to terminate. This in turn allows the backends to explore a much larger portion of a grammar than previous approaches. Whereas the extant deterministic approaches detect ambiguity by generating lots of short sentences typically containing tens of tokens, *SinBAD*'s backends detect ambiguity by generating long sentences that can easily contain hundreds of tokens. This then allows *SinBAD*'s backends to uncover ambiguous fragments nested deep within a grammar. My results show that *SinBAD*'s non-deterministic breadth-based backends uncovered 16% more ambiguities than the deterministic approaches. In essence, my results suggest that covering a grammar's state space in breadth is more important than covering it in depth.

I also introduced two new ways of generating large grammar corpuses: Boltzmann sampling and grammar mutation. The grammars created using Boltzmann sampling were highly ambiguous and thus not entirely representative of PL grammars. The mutated grammars, on the other hand, are representative of PL grammars although different mutations lead to different degrees of ambiguity. My experience suggests that for uses that require exploring a wide class of grammars, one should use Boltzmann sampling, whereas for uses that require exploring PL grammars one should use grammar mutation.

My breadth-based ambiguity detection approach often uncovers ambiguous fragments

that are long and deeply nested. The ambiguous fragment uncovered can easily exceed hundreds of tokens. By using a search-based approach to grammar ambiguity minimisation, I showed that both the ambiguous input and the grammar portion that contributes to ambiguity can be effectively minimised. My results show that the grammar size is significantly reduced (by at least 60%) across grammar sets.

8.2 Future Work

Since the problem of ambiguity detection is inherently undecidable, there is always potential to improve an ambiguity detection approach. An obvious approach would be to experiment with a hybrid deterministic/non-deterministic approach, perhaps using some of AmbiDexter or AMBER's methods alongside some of *SinBAD*'s methods.

Although my evaluation of the ambiguity detection tools is the largest to date, it is by no means perfect. My cross ambiguity detection tool experiment evaluated four tools. I believe that adding more ambiguity detection tools to my experimental suite would be a significant improvement.

Another opportunity for future research is to extend the PL grammar corpuses in my experimental suite. The current mutated PL grammar corpus was generated from a tiny set of unambiguous grammars, which can easily lead to over-training. It would be useful, though neither easy or pleasant, work to increase the size of this set by using more real programming language grammars.

Appendices

Appendix A

Non-termination in *dynamic2* – Boltzmann Grammar

The relevant subset of the Boltzmann grammar[1] from my experimental corpus that runs into non-termination is shown below:

```
BDU : ... EURPA 'Z' TK_KI FUQR TK_GF | DSH P TK_SA BDU EGCB
```

For the example grammar, *dynamic2* was invoked with D=10. For the rule `BDU`, for the first alternative, `FUQR` was the hardest to derive symbol; for the second alternative, since it is recursive, `BDU` was the hardest to derive symbol. Therefore, the score of the first and second alternative is set by the score of the symbol `FUQR` and `BDU` respectively. For a sentence generation run that didn't terminate and ran out of stack, the number of required recursive invocations to `BDU` until its score reached parity with `FUQR`'s score at various points during sentence generation run, is shown in the Table A.1.

The first recursive cycle begins at d=45. Since d>D, favouritism will apply. When rule `BDU` is entered, the second (recursive) alternative with the lower score (0.5) is picked. A recursive cycle ensues: the symbol `BDU` is recursively called 5 times until `BDU`'s score $\geq$ `FUQR`'s score. Subsequently, when the rule `BDU` is entered, the first (non-recursive) alternative gets picked, and the sentence generation unwinds from recursion.

The second recursive cycle begins at d=41. Since d>D, when rule `BDU` is entered, the second (recursive) alternative with the lower score (0.125) is picked. This time, the symbol `BDU` is recursively called 17 times until `BDU`'s score $\geq$ `FUQR`'s score. Subsequently, when the rule `BDU` is entered, the first (non-recursive) alternative gets picked, and the sentence generation unwinds from recursion.

As the sentence generation progresses, each time `FUQR` is picked and the sentence generation unwinds, the number of recursive calls to `BDU` that is needed for its score to reach

[1] `https://github.com/nvasudevan/experiment/blob/master/grammars/boltzcfg/75/4.acc`

	FUQR		BDU	
depth (d)	exited/entered	score	exited/entered	score
45	1/6	0.8333	1/2	0.5
50	1/6	0.8333	1/7	0.8571
41	2/7	0.7142	7/8	0.125
58	2/7	0.7142	7/25	0.72
58	3/9	0.3	25/26	0.0384
107	3/9	0.3	25/75	0.3
57	4/10	0.6	75/76	0.0132
169	4/10	0.6	75/188	0.6010
56	5/11	0.5454	188/189	0.0052
281	5/11	0.5454	188/414	0.5458
62	6/12	0.5	414/415	0.0024
476	6/12	0.5	414/829	0.5006
60	7/13	0.4615	829/830	0.0012
770	7/13	0.4615	829/1540	0.4616
57	8/14	0.4285	1540/1541	0.0006
977[†]	8/14	0.4285	1540/2461	0.3742

[†] The recursion limit of the Python stack was reached.

Table A.1: Table showing the required number of recursive invocations to BDU until its score reaches parity with FUQR's score at various points during a sentence generation run that didn't terminate. For a given symbol s, its score is calculated as: (1-($s.exited/s.entered$)).

parity with FUQR's, also increases. The number of recursive invocations to BDU until its score reaches parity with FUQR's score at various point during the sentence generation run are 49, 112, 225, 416, 710, and 920. In the final case, the number of recursive invocations (920) needed was high enough for the sentence generator to run out of stack.

Increasing the recursion limit of the underlying stack only prolongs the non-termination problem. For a sentence generation run for the example grammar with D=10 and recursion limit set to 10000, the number of times BDU entered into recursion was 341. The

number of recursive invocations to `BDU` just before the sentence generator ran out of stack was 9360. The output from sentence generation for both examples sentences (for recursion limit=1000 and 10000) can be downloaded from: `https://figshare.com/s/fa80114ab5233ed8ceba`.

Appendix B

Crude Dimensioning – Altered PL Grammars

Figure B.1 shows the results from the crude dimensioning run for AMBER length and examples options, AmbiDexter length and the $dynamic_n$ backends for the altered PL grammar set. Table B.1 shows the results from the crude dimensioning run for the $dynamic2_{rws}$ backend for the altered PL grammar set.

W[♭]\D	1	2	3	6	9	12	15	18	30	50
0.005	16	17	17	17	20	20	20	20	18	10
0.01	18	17	18	18	20	20	19	19		
0.0105	18	17	18	18	20	20	20	20		
0.011025	15	17	17	19	19	20	19	20		
0.012127	15	18	18	17	20	20	20	20		
0.013340		18		19			20	20		
0.014674		16		19						
0.016141		18								
0.05	19	19	20	20	20	20	19	20	19	11
0.075	18	18	20	20	20	20	20	20	17	10
0.1	20	20	20	20	20	20	20	20	17	11
0.2	20	20	20	20	20	20	20	19	16	12
0.5	11	12	11	12	11	11	11	12	12	9

[♭] Weights are approximated to the nearest hundred thousandth.

Table B.1: Crude dimensioning for the $dynamic2_{rws}$ backend: the number of ambiguities found for various D and W for altered PL grammars. Several combinations of D and W found all the ambiguities.

For the AMBER and the AmbiDexter length option, additional tool runs were performed for length len=30, 50, 75, 100, 150, 200, and 500. For the $dynamic_n$ backends, additional tool runs were performed for depth D=30, 50, 75, 100, 150, 200, and 500. For $dynamic4$,

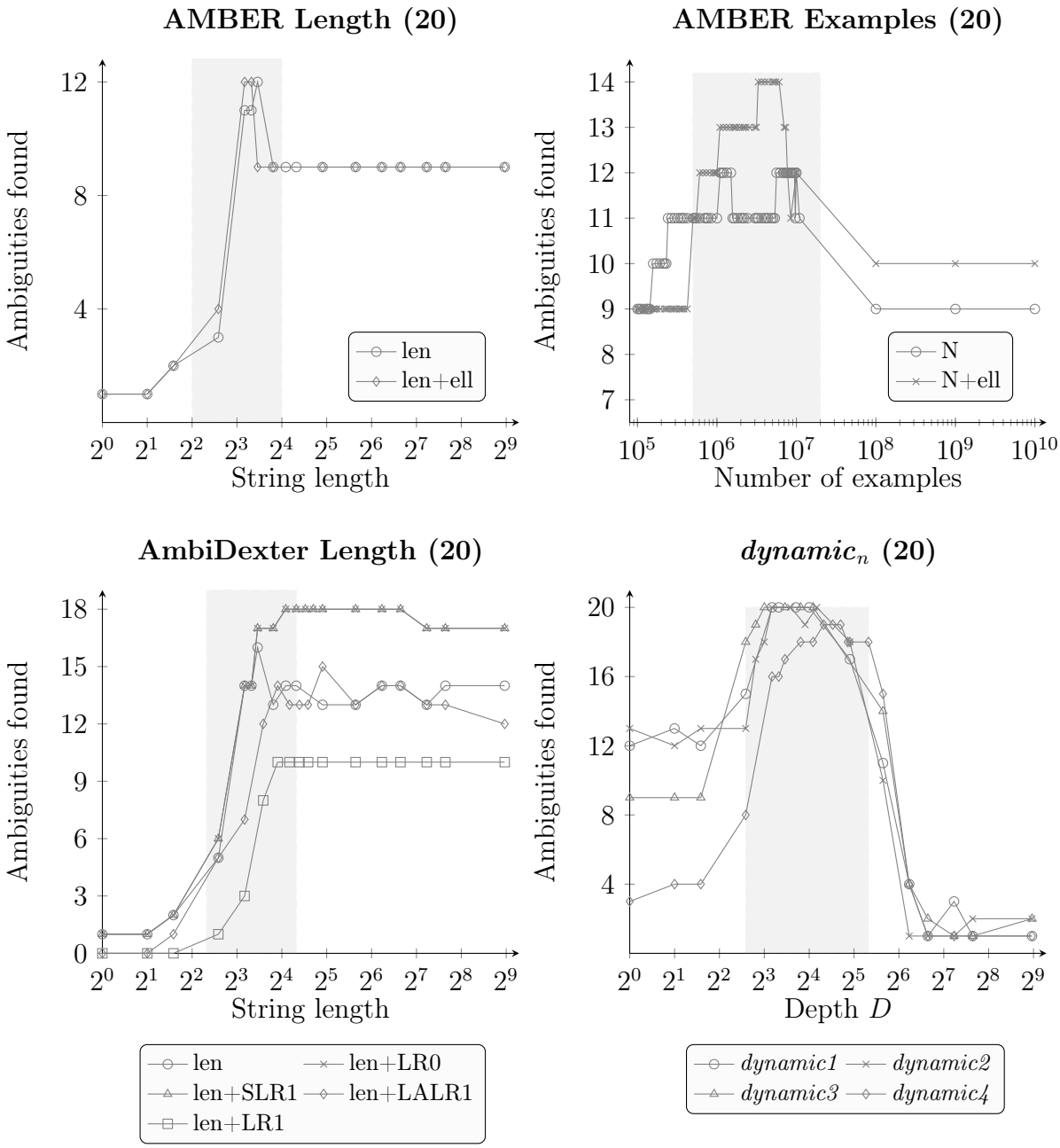

Figure B.1: Crude dimensioning for altered PL grammars: the number of ambiguities found by tool options, AMBER and AmbiDexter length, AMBER examples, and the $dynamic_n$ backends. The number in brackets in the title of each graph indicate the number of grammars.

since the hill climbing run finished fairly early at a low value of D=6, the backend was additionally invoked for D=15, and 25. For the $dynamic2_{rws}$ backend, additional tool runs were performed for combination of depths D=30 and 50 and weights W=0.005, 0.05, 0.075, 0.1, 0.2, and 0.5. For AMBER examples, the hill climbing run required several restarts. In both cases, with ellipsis set or otherwise, hill climbing was restarted 10 times.

From Figure B.1, we can see that the fitness distribution for tool option AMBER length (with ellipsis set or otherwise) and AmbiDexter length (for unfiltered and for each filter) is asymptotic. In case of AMBER length, just before the fitness distribution starts to plateau, there seem to exist a small band in the solution space where the fitness peaks; I explore this solution space in detail in fine dimensioning. For the AMBER examples (with ellipsis set or otherwise), the fitness distribution appears to like a bell curve.

For the $dynamic_n$ backends, from Figure B.1, we can see that the fitness distribution is a bell curve for *dynamic1*, *dynamic2* and *dynamic4*. *dynamic3* showed maximum fitness even for low values of D. For all the $dynamic_n$ backends, as D increases, the fitness drops. For the $dynamic2_{rws}$ backend, the fitness distribution appears to look like a hill. As the depth increases, the fitness ramps up for a while and then starts to dip. For each depth, as the value of weight increases, the fitness ramps up for a while and then start to dip. For each tool option, the solution space containing potential good solutions (indicated by gray shaded region in Figure B.1) is explored in detail in fine dimensioning.

Appendix C

Fine Dimensioning – Altered PL Grammars

Figure C.1 shows the results of the fine dimensioning run for the tool options AMBER length, AMBER examples, AmbiDexter length, and the $dynamic_n$ backends for the altered PL grammar set. Table C.1 shows the results of the fine dimensioning run for the AmbiDexter incremental length option for the unfiltered and the filtered options for the altered PL grammar set. Table C.2 shows the results of the fine dimensioning run for the $dynamic2_{rws}$ backend for the altered grammar set.

AmbiDexter *ilen*				
-	LR0	SLR1	LALR1	LR1
16	18	18	18	10

Table C.1: Fine dimensioning for AmbiDexter *ilen*: number of ambiguities found for the unfiltered and the filtered options for the altered PL grammar set.

Table C.3 shows for the altered PL grammar set: the best length and the additional tool runs performed for the AMBER length option; and the best N for the AMBER examples option. For AMBER length (with ellipsis set or otherwise), the additional tool runs did not uncover a better solution. For AMBER examples, I terminated the hill climbing run: at N=412×10^6 (with ellipsis not set) and at N=118×10^6 (with ellipsis set).

Table C.4 shows the additional tool runs performed for the AmbiDexter length option for the altered PL grammar set. For AmbiDexter length (unfiltered and for each filter) option, several values of length uncovered maximum number of ambiguities. The list is too exhaustive to mention it in the table, so I summarise it here: at the end of both the hill climbing and the additional runs, for unfiltered, I ended up exploring all the values

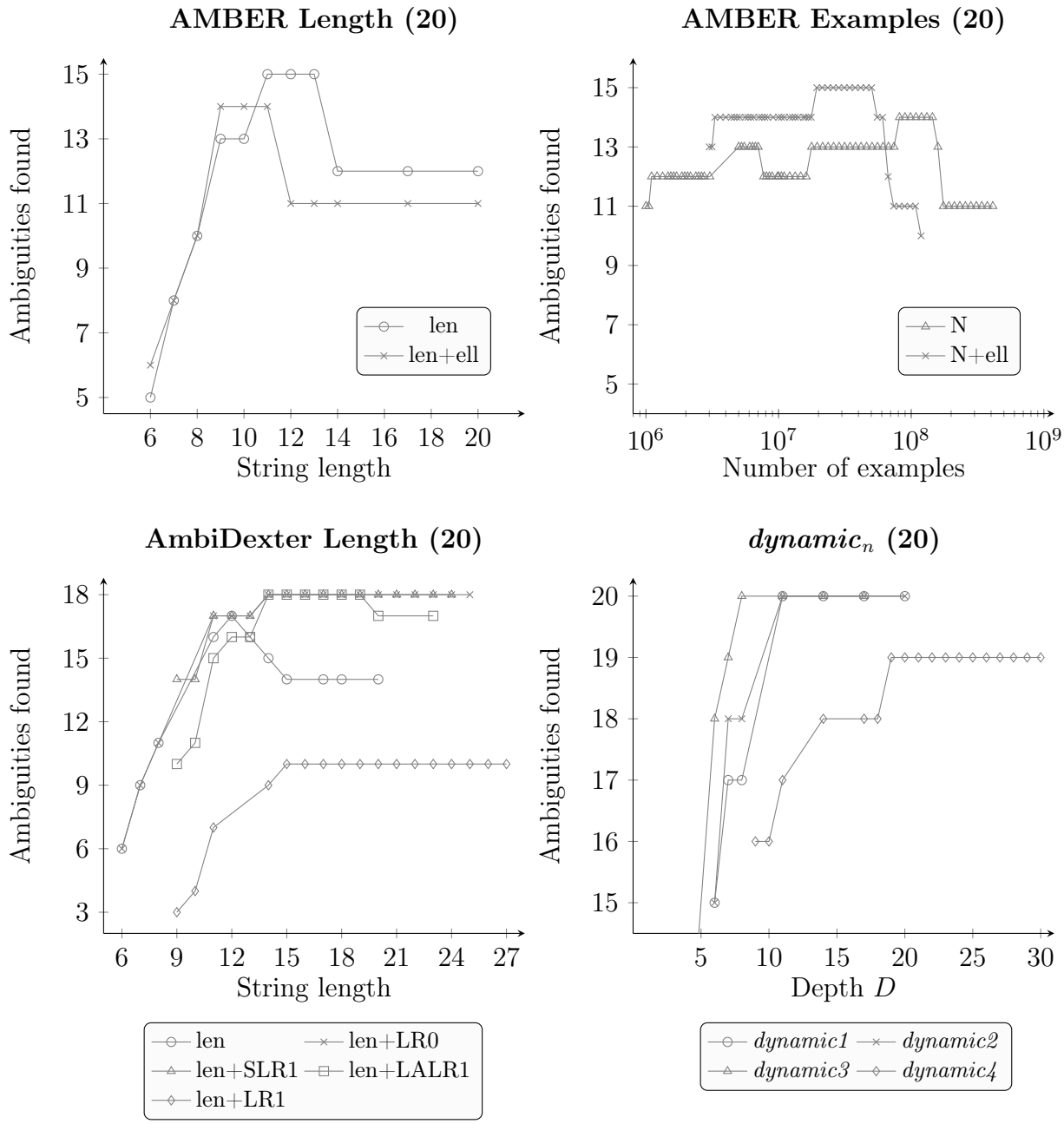

Figure C.1: Fine dimensioning for the altered PL grammar set: number of ambiguities found by tool options, AMBER length, AMBER examples, AmbiDexter length, and the $dynamic_n$ backends. The number in brackets in the title of each graph indicate the number of grammars.

W[♭]\D	6	7	8	11
0.01	20	20	20	20
0.0105	20	20	20	20
0.011025	20	20	20	20
0.012127	20	20	19	20

[♭] Weights are approximated to the nearest hundred thousandth.

Table C.2: Fine dimensioning for the *dynamic2*$_{rws}$ backend: the number of ambiguities found by for various D and W for altered PL grammars. Several combinations of D and W found all the ambiguities.

AMBER			
len	len+ell	N	N+ell
(11, 15)	(11, 14)	(81587796, 14)[⊕]	(19404000, 15)[⊕]
(9, 13)	(9, 14)	-	-
(10, 13)	(10, 14)		
(12, 15)	(12, 11)		
(13, 15)	(13, 11)		

[⊕] Several values of N uncovered maximum number of ambiguities.

Table C.3: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the AMBER length and the examples option for the altered PL grammar set. Pair value (l, f) denote the length l sampled and its fitness value. Pair value (N, f) denote the number of examples N sampled and its fitness value.

between 8 and 20; and for each of the filters, I ended up exploring most of the values between 9 and 27. Only for the unfiltered option, did the additional runs uncovered a better solution for len=12 (ambiguities found=17).

Since the backends *dynamic1*, *dynamic2*, *dynamic3*, and *dynamic2*$_{rws}$ uncovered all the ambiguities in the hill climbing run, additional tool runs were not performed. For *dynamic4* backend, the hill climbing run uncovered maximum number of ambiguities (19) for D=20, 23, 26, and 29. Additional tool runs were performed for D=18, 19, 21, 22, 24, 25, 27, 28, and 30. The additional runs did not uncover a better solution.

AmbiDexter Length[≜]				
unf[≜]	LR0	SLR1	LALR1	LR1
(11, 16)	(15, 18)	(14, 18)	(14, 18)	(17, 10)
(12, 17)[♯]				

[♯] Better solution found from the additional tool runs.
[≜] AmbiDexter invoked with filter not set.

Table C.4: Best solution from the hill climbing run (shown in the top half of the table) and additional tool runs (shown in the bottom half of the table) for the AmbiDexter length option for the altered PL grammar set. Pair value (l, f) denote the length l sampled and its fitness value.

Appendix D

Verifying *minimiser1*

Algorithm 21 describes how *minimiser1*'s grammar minimisation approach is verified for the *dynamic3* backend. The function START is initialised with grammar G, and its minimised version G_m, the ambiguous fragment amb_s generated by *minimiser1*, the threshold depth D beyond which alternatives are favoured.

In addition to the notations outlined in Section 7.1, I define one other notation. For a given grammar G, and a given sequence of symbols *syms*, the function getalt(*syms*) iterates through G's rules and returns the position of the first occurrence of *syms* as a tuple $\langle r,\text{alt},i\rangle$, where r is the rule, *alt* is the alternative of rule r, and i is the index at which the *syms* is positioned in *alt*.

The sequence of symbols *syms* that contribute to ambiguity is calculated from the start rule of the minimised grammar (line 2). The occurrence of *syms* in G is obtained by invoking the function getalt (line 3). The function GENERATE is invoked to start sentence generation from the start rule $\mathcal{R}(S)$ of the grammar, with d=0 (line 5). The original GENERATE function (see Algorithm 9) has been modified to include two changes. First, the modified GENERATE function accepts two additional parameters: the tuple $\langle\widehat{r},\widehat{\text{alt}},\widehat{i}\rangle$ containing the reference to the sequence of symbols that contribute to ambiguity; and the pair (*syms*, amb_s). Second, the modified GENERATE function invokes the function INSERT-AMBS when $\widehat{r}$ is visited to insert the ambiguous fragment amb_s into sentence *sen* (lines 9 and 10) .

The function INSERT-AMBS is initialised with grammar G, sentence *sen*, current depth of recursion d, threshold depth D, the tuple $\langle\widehat{r},\widehat{\text{alt}},\widehat{i}\rangle$, and the pair (*syms*, amb_s).

j tracks the position of the current symbol in the alternative that is being derived (line 38). Each symbol from the alternative $\widehat{alt}$ is sequentially explored. For each symbol *sym* that precedes or succeeds *syms*, if *sym* is a non-terminal, the function GENERATE is invoked (lines 40–43); if *sym* is a terminal then it is appended to the sentence *sen* (line 45). To derive the sequence of symbols *syms*, the string amb_s is appended to sentence

sen (line 49). If the generated sentence is ambiguous, then *minimiser1*'s minimisation approach has been verified.

Algorithm 21 Algorithm to verify *minimiser1* using the *dynamic3* backend

```
1: function START(G, G_m, amb_s, D)
2:     syms ← R(S_m).seqs[0]                          ▷ G_m = ⟨N_m,T_m,P_m,S_m⟩
3:     ⟨r̂,alt̂,î⟩ ← getalt(syms)
4:     sen ← [ ]
5:     GENERATE(G, R(S), sen, d=0, D, ⟨r̂,alt̂,î⟩, (syms, amb_s))
6: end function

7: function GENERATE(G, rule, sen, d, D, ⟨r̂,alt̂,î⟩, (syms, amb_s))
8:     d ← d + 1
9:     if rule = r̂ then
10:        INSERT-AMBS(G, sen, d, D, ⟨r̂,alt̂,î⟩, (syms, amb_s))
11:    else
12:        if d > D then
13:            if rule.finite_depth ≠ NONE then
14:                alt ← rule.finite_depth
15:            else
16:                alt_fd ←CALC-ALT-FINITE-DEPTH(G, rule)
17:                if alt_fd ≠ NONE then
18:                    rule.finite_depth ← alt_fd
19:                    alt ← alt_fd
20:                else
21:                    alt ← rand(rule.alts)
22:                end if
23:            end if
24:        else
25:            alt ← rand(rule.alts)
26:        end if
27:        for sym in alt do
28:            if sym ∈ N then                        ▷ If sym is a non-terminal
29:                GENERATE(G, R(sym), sen, d, D, ⟨r̂,alt̂,î⟩, (syms, amb_s))
30:            else
31:                append(sen, sym)
32:            end if
33:        end for
34:    end if
35:    d ← d − 1
36: end function
```

37: **function** INSERT-AMBS(G, sen, d, D, $\langle \widehat{r}, \widehat{\text{alt}}, \widehat{i} \rangle$, ($syms$, amb_s))
38: $j \leftarrow 0$
39: **while** $j <$ len($\widehat{alt}$) **do**
40: **if** ($j < i$) **or** ($j \geq (i +$ len($syms$))) **then**
41: $sym \leftarrow \widehat{alt}[j]$
42: **if** $sym \in N$ **then**
43: GENERATE(G, $\mathcal{R}(sym)$, sen, d, D, $\langle \widehat{r}, \widehat{\text{alt}}, \widehat{i} \rangle$, ($syms$, amb_s))
44: **else**
45: append(sen, sym)
46: **end if**
47: $j \leftarrow j + 1$
48: **else**
49: append(sen, amb_s)
50: $j \leftarrow j +$ len($syms$)
51: **end if**
52: **end while**
53: **end function**

Bibliography

[1] GNU Bison: The YACC-compatible parser generator. `https://www.gnu.org/software/bison`, 1988–2015.

[2] ANTLR grammars. `https://github.com/antlr/grammars-v4`, 2012–2015.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. .H. Wegstein, A. van Wijngaarden, , and M. Woodger. Report on the algorithmic language ALGOL 60. *Journal of the ACM*, 3(5):299–314, 1960.

[4] H. J. S. Basten. Ambiguity detection methods for context-free grammars. Master's thesis, Universiteit van Amsterdam, Aug 2007.

[5] H. J. S. Basten. Grammar collection containing altered programming languages. `http://sites.google.com/site/basbasten/files/grammars.zip`, 2010.

[6] H. J. S. Basten and T. van der Storm. AMBIDEXTER: Practical ambiguity detection. In *Proc. SCAM 2010*, pages 101–102, 2010.

[7] H. J. S. Basten and J. J. Vinju. Faster ambiguity detection by grammar filtering. In *Proc. LDTA*, pages 5:1–5:9, 2010.

[8] H. J. S. Basten and J. J. Vinju. Parse forest diagnostics with dr. ambiguity. In *Proceedings of the 4th International Conference on Software Language Engineering*, pages 283–302. Springer-Verlag, 2012.

[9] E. Bendersky. Weighted random generation in python, 01 2010. `http://eli.thegreenplace.net/2010/01/22/weighted-random-generation-in-python`.

[10] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, Mar 2010.

[11] B. Canou and A. Darrasse. Fast and sound random generation for automated testing and benchmarking in objective caml. In *Proc. Workshop on ML*, pages 61–70, 2009.

[12] D. G. Cantor. On the ambiguity problem of backus systems. *Journal of the ACM*, 9(4):477–479, 1962.

[13] X. Chen. Hyacc parser generator. `http://hyacc.sourceforge.net`, 2008–2015.

[14] B. S. N. Cheung and R. C. Uzgalis. Ambiguity in context-free grammars. In *Proc. SAC*, pages 272–276. ACM, 1995.

[15] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, June 1959.

[16] L. Diekmann and L. Tratt. Eco: A language composition editor. In *Software Language Engineering (SLE)*, pages 82–101. Springer, Sep 2014.

[17] J. Earley. An efficient context-free parsing algorithm. *Journal of the ACM*, 13(2):94–102, 1970.

[18] S. Gorn. Detection of generative ambiguities in context-free mechanical languages. *Journal of the ACM*, 10(2):196–208, 1963.

[19] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood series in computers and their applications. Ellis Horwood, New York, 1990.

[20] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Techniques, taxonomy, tutorial. *ACM Comput. Surv.*, 45(1):11:1–11:61, Dec 2012.

[21] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, Dec 2012.

[22] M. D. Hutton. Noncanonical extensions of LR parsing methods. 1990.

[23] K. Ulik II and R. Cohen. LR-regular grammars-an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7(1):66–96, 1973.

[24] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T Bell Laboratories, 1979.

[25] A. F. Kaupe. A note on the dangling else algol 60. *Commun. ACM*, 6(8):460–, Aug 1963.

[26] D. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[27] R. Lämmel. Grammar testing. 2029:201–216, 04 2001.

[28] B. Jones M. Harman. Search based software engineering. *Journal of Information and Software Technology*, 43(14):833–839, 2001.

[29] V. Makarov. MSTA (*syntax description translator*). `http://cocom.sourceforge.net/msta.html`, 1999.

[30] M. Mehryar and M. Nederhof. *Regular Approximation of Context-Free Grammars through Transformation*. Springer Netherlands, 2001.

[31] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria. Uniform random generation of huge metamodel instances. In *ECMDA-FA*, pages 130–145, 2009.

[32] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[33] P. Purdom. A sentence generator for testing parsers. 12:366–375, 09 1972.

[34] S. Schmitz. Conservative ambiguity detection in context-free grammars. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 692–703. Springer, July 2007.

[35] F. W. Schröer. ACCENT, a compiler compiler for the entire class of context-free grammars. Technical report, 2000. `http://accent.compilertools.net/Accent.html`.

[36] F. W. Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, 2001. `http://accent.compilertools.net/Amber.html`.

[37] E. Scott and A. Johnstone. GLL parsing. In *Proc. LDTA*, volume 253, pages 177–189, 2010.

[38] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[39] N. Vasudevan and L. Tratt. Search-based ambiguity detection in context-free grammars. In *Proc. ICCSW*, pages 142–148, Sep 2012.

[40] N. Vasudevan and L. Tratt. Detecting ambiguity in programming language grammars. In *Proc. SLE*, Oct 2013.