University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

2011

Executable Attribute Grammars for Modular and Efficient Natural Language Processing

Rahmatullah Hafiz
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Hafiz, Rahmatullah, "Executable Attribute Grammars for Modular and Efficient Natural Language Processing" (2011). *Electronic Theses and Dissertations*. 412.
https://scholar.uwindsor.ca/etd/412

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Executable Attribute Grammars for Modular and Efficient Natural Language Processing

by

Rahmatullah Hafiz

A Dissertation
Submitted to the Faculty of Graduate Studies
through School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy at the
University of Windsor

Windsor, Ontario, Canada
2011

© 2011 Rahmatullah Hafiz

Executable Attribute Grammars for
Modular Natural Language Processing
by Rahmatullah Hafiz

APPROVED BY:

Dr. Guy Lapalme , External Examiner
Dpartement d'informatique et de recherche oprationnelle, University of Montreal

---

Dr. Richard J. Caron, External Reader
Department of Mathematics and Statistics

Dr. Jianguo Lu, Internal Reader
School of Computer Science

Dr. Luis Rueda, Internal Reader
School of Computer Science

Dr. Richard A. Frost, Advisor
School of Computer Science

Dr. Jichang Wang, Chair of Defense
Department of Chemistry and Biochemistry

September 15, 2011

Declaration of Co-Authorship / Previous Publication

I. Co-Authorship Declaration

I hereby declare that this thesis incorporates the outcome of joint research undertaken under the supervision of Dr. Richard A. Frost. The collaboration is covered in Chapters 2, 3, 4 and 5 of the dissertation. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the Rahmatullah Hafiz (the candidate) and Dr. Richard A. Frost (the supervisor) as primary authors and contributors, and for papers included in chapters 2 and 3 Dr. Paul Callaghan, from the University of Durham, was a contributory author.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This thesis includes three original papers that have been previously published, and one original paper that has been recently accepted for publication in peer-reviewed conferences, as follows:

Thesis Chapter	Publication title/full citation	Publication Status
2	Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars. ACL-IWPT, 2007, 109 - 120	Published
3	Parser Combinators for Ambiguous Left-Recursive Grammars. ACM-PADL, 2008, 167-181	Published
4	Lazy Combinators for Executable Specifications of General Attribute Grammars. ACM-PADL, 2010, 167-182	Published
5	A System for Modularly Constructing Efficient Natural Language Processors. Computational Linguistics Applications Conference 2011	Accepted for publication

I certify that I have obtained written permission from the copyright owner(s) to include the above published material(s) in my thesis (see Appendix D). I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone‘s copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis are fully acknowledged. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other Institution.

Abstract

Language-processors that are constructed using top-down recursive-descent with backtracking parsing are highly modular, and are easy to implement and maintain. However, a widely-held inaccurate view is that top-down processors are inherently exponential for ambiguous grammars and cannot accommodate left-recursive syntax rules. It has been known that exponential time and space complexities can be avoided by memoization and compact graph-structured representation, and that left- recursive productions can be accommodated through a variety of techniques. However, until now, memoization, compact representation, and techniques for handling left-recursion have either been presented independently, or else attempts at their integration have compromised modularity and correctness of the resulting parses.

Specifying syntax and semantics to describe formal languages using denotational notation of attribute grammars (AGs) has been widely practiced. However, very little work has shown the usefulness of declarative AGs for constructing computational models of natural language. Previous top-down approaches fall short in accommodating ambiguous and general CFGs with arbitrary semantics in one pass as executable specifications. Existing approaches lack in providing a declarative syntax-semantics interface that can take full advantages of dependencies between attributes of syntactic constituents to model linguistically-motivated cases.

This thesis solves these shortcomings by proposing a new modular top-down syn-

tactic and semantic analysis system, which is efficient and accommodates all forms of CFGs. Moreover, this system provides notation to declaratively specify semantics by establishing arbitrary dependencies between attributes of syntactic categories to perform linguistically-motivated tasks such as: building directly-executable natural-language query processors, computing meanings of sentences using compositional semantics, performing contextual disambiguation tasks, modelling restrictive classes of languages etc.

Dedication

To my father Md. Hafiz Uddin (1945 - 2009),

for teaching me the value of honesty and hard-work by leading an exemplary life.

Acknowledgements

I express my deepest gratefulness to my supervisor Dr. Richard A. Frost for believing in me. His passionate curiosity in solving problems, sincere commitment in creating knowledge, and natural enthusiasm in mentoring have inspired me in every step of my graduate study. His insightful observations and creative suggestions had helped me to see the bigger picture and to overcome obstacles on a daily basis. It has been a privilege to learn from him and to work with him so closely.

I want to convey my sincere thankfulness to my external examiner Dr. Guy Lapalme, my external reader Dr. Richard J. Caron, and my internal readers Dr. Luis Rueda and Dr. Jianguo Lu for their constructive feedback on my thesis report and the defense presentation. I also would like to thank Dr. Paul Callaghan of the University of Durham for his valuable contribution during the early stage of this work.

My parents' unconditional love and trust still guide me through many unproductive days. I am grateful to my parents for providing me with shelves full of books of diverse content. My heartfelt admiration goes to my wife and best friend Sanjukta Dutta for handling my long hours with such a grace and understanding. Her integrity, patience, and dedication in helping others guide me in becoming a better person.

Contents

Declaration of Co-Authorship / Previous Publication **iii**

Abstract **v**

Dedication **vii**

Acknowledgements **viii**

1 Introduction **1**

1.1 Introduction 1

1.2 Thesis Statement 2

1.3 Top-down Parsing Algorithm for General Syntax 3

1.4 Efficient Combinators for General Syntax 5

1.5 Fully-General Attribute Grammars 8

1.6 NLP Tasks with Modular AGs 11

1.7 Distinguishing Features for Effective NLP Applications 13

Bibliography 16

2 Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive

Grammars **19**

2.1 Introduction 20

2.2 Top-Down Backtracking Recognition 26

2.3 Left Recursion and Top-Down Parsing 28

2.4 The New Method 31

2.4.1 Memoization 31

2.4.2 Accommodating direct left recursion 32

2.4.3 Accommodating indirect left recursion 34

2.4.4 Extending recognizers to parsers 36

2.5 Termination 38

2.6 Complexity 39

2.7 Implementation 40

2.8 Experimental Results 40

2.8.1 Tomita's Grammars 41

2.8.2 Highly ambiguous abstract grammars 42

2.9 Concluding Comments 43

Bibliography 45

3 Parser Combinators for Ambiguous Left-Recursive Grammars **47**

3.1 Introduction 48

3.2 Background 52

3.2.1 Top down parsing and memoization 52

3.2.2 The need for left recursion 53

3.2.3 An introduction to parser combinators 53

3.3 The Combinators 55

3.3.1 Preliminaries 55

3.3.2 Memoizing recognizers 55

3.3.3 Accommodating direct left recursion 57

3.3.4 Accommodating indirect left recursion 59

3.3.5 Building parse trees 62

3.4 Experimental Results 64

3.4.1 Tomita' Grammars 64

3.4.2 Highly ambiguous abstract grammars 65

3.4.3 ATIS – A medium size NL grammar 65

3.5 Related Work 66

3.6 Concluding Comments 67

Bibliography 68

4 Lazy Combinators for Executable Specifications of General Attribute Grammars 70

4.1 Introduction 71

4.2 General AGs and Parser Combinators 75

4.3 Executable Specifications of General AGs 76

4.3.1 Preliminaries 76

4.3.2 Combinators for Syntax 78

4.3.3 Accommodating Arbitrary Dependencies in Semantics 80

4.3.4 Declarative Executable Specifications of Semantic Rules 82

4.4 Use of Memoization 84

4.5 Complexity Analysis 85

4.6 Implementation and an Example Application 86

4.7 Related Work 88

4.8 Concluding Comments 89

Bibliography 90

5 A System for Modularly Constructing Efficient Natural Language Processors 93

5.1 Introduction 94

5.2 General Notation 95

5.2.1 Operators for Syntactic Analysis 95

5.2.2 Operators for Semantic Analysis 101

5.2.2.1 Computing Meaning 101

5.2.2.2 Using Semantics for Disambiguation 103

5.3 Background 105

5.3.1 Grammar Formalisms 106

5.3.2 Top-Down Parsing with Parser Combinators 106

5.4 Declarative and Executable Attribute Grammars 107

5.4.1 Forming Arbitrary Dependency 107

5.4.2 Construction of Combinators for Executable AGs 109

5.5 Modeling Unification-Based Formalisms 110

5.6 Accommodating Phenomena Beyond Context-Free 113

5.7 Concluding Comments 117

Bibliography 118

6 Conclusion **120**

6.1 Concluding Comments 120

A Formal Properties of Our Approach **124**

A.1 Termination 124

A.2 Correctness Analysis 128

A.3 Complexity Analysis 132

A.3.1 Time Complexity w.r.t. the length of input n 132

A.3.2 Parsing complexity w.r.t. the size of the grammar p 134

A.3.3 Space Complexity w.r.t. the length of the input n 135

Bibliography 136

B Example NL Query-Processor Using the New Combinators **137**

B.1 The Dictionary 137

B.2 Executable Attribute Grammars 142

B.3 Functions for Attribute Evaluation 147

B.4 The Database 151

B.5 Sample NL Input Query 152

C List of Refereed Papers Related to the Thesis **154**

D Copyright Releases **156**

Vita Auctoris **159**

Chapter 1

Introduction

1.1 Introduction

We still lack systems that can help application developers to seamlessly build domain-specific natural language (NL) processing applications that can accommodate a wide range of linguistic phenomenon. One way to address this problem is by reducing the gap between the notation that the developers are comfortable working with and the theoretical foundations that are required for deep linguistic processing. The goal of this thesis is to provide an efficient and modular framework for syntactic and semantic analysis of natural languages. Here I describe the development of a programming environment, which enables the construction of declarative descriptions of natural languages as directly-executable specifications. (Project home-page: `http://cs.uwindsor.ca/~hafiz/xsaiga/proHome.html`)

An effective natural language processor should accommodate linguistic features such as correct construction of syntactic analysis, ability to compute meanings of sentences following linguistic theories, ability to accommodate syntactic and semantic ambiguities, mechanisms for disambiguation, capability to impose constraints to express characteristics beyond context-freeness etc. This thesis attempts to provide these features within a single system. The foundation of this work is a new top-down parsing algorithm that solves a forty-year-old problem by accommodating any form of context-free grammar (CFG) in-

cluding ambiguous left-recursive grammars in polynomial time and space [1, 2]. This algorithm employs a depth-checking and context-determining technique for left-recursive rules, uses a variation of memoization for time efficiency, and represents the potentially exponential number of ambiguous trees in a compact form for space efficiency. Our approach has been adopted by Python programmers in developing the Python library parser LEPL (`acooke.org/lepl/implementation.html`).

Specifying syntax and semantics to describe formal languages using denotational notation of attribute grammars (AGs) has been practiced extensively. However, very little work investigates the usefulness of declarative AGs for computational models of natural language. I have extended our general syntax analysis approach to allow declaratively coupling semantic rules with syntax. I have defined a set of combinators (higher-order functions) that enable the syntactic and semantic description of natural languages, using the general notation of AGs, as directly-executable specifications ([3, 4]. The underlying method depends on referential transparency of purely-functional definitions and on the non-strict evaluation strategy to establish arbitrary dependencies in semantic rules. The combinators are implemented in the purely-functional programming language Haskell (`haskell.org`). I have proposed an innovative technique [5] that allows parses to be pruned by arbitrary semantic constraints. This new technique is useful in modelling NL phenomena by imposing unification-like restrictions, accommodating long-distance and cross-serial dependencies, and performing limited forms of disambiguation tasks which cannot be handled by CFGs alone.

1.2 Thesis Statement

This report contains four papers addressing different aspects of the theme outlined above. Collectively these papers and the appendixes prove the following thesis statement:

"It is possible to build an efficient environment for the modular construction of executable specifications of arbitrary natural languages where syntactic and semantic rules can be expressed in a declarative notation."

The rest of this chapter introduces the four papers in general terms with some related

background information. Please see **Appendix C** for a list of all seven papers related to this thesis that I authored/co-authored.

1.3 Top-down Parsing Algorithm for General Syntax

Chapter 2 contains a paper that describes a new top-down parsing algorithm that can accommodate any form of ambiguous context-free grammar (CFG) (including direct and indirect left-recursive grammars) efficiently. A context-free grammar (CFG) G is 4-tuple $G = (N, T, P, S)$, where N is a finite set of non-terminals, T is a finite-set of terminals, P is a finite-set of syntax rules, S is the start non-terminal , $N \cap T = \phi$ and $(\forall p_i \in P)$ p_i is of the form $a ::= b$ where $a \in N$ and $b \in (N \cup T)^*$. The foundation of the algorithm is a traditional recursive-descent backtracking search, which normally fails to terminate while processing left-recursive grammar rules that would call itself indefinitely unless no method of prevention is taken. For example, consider the CFG, $S ::= S\ S\ `a` \mid \epsilon$ that describes a language containing members of the set $\{a, aa, aaa, aaaa, aaaa, ...\}$. The naive recursive descent approach fails for this grammar as the top-most symbol S will keep calling itself endlessly. This grammar is highly-ambiguous (its ambiguity can be represented with the Catalan number series [6, 7]), and even if left-recursion could be accommodated, the traditional backtracking search would require exponential time to process this grammar and exponential amount of space to represent the resulting parses.

The functional description of the top-down recursive-descent parsing algorithm treats each rule of a grammar as a function-definition, which can contain left and right recursion. Any rule or non-terminal of a CFG, say A, is in the form of $A \rightarrow \alpha$ where $A \in \{nonterminals\}$, and $\alpha \in (\{nonterminals\} \cup \{terminals\})^*$. The non-terminal A can have more than one such definition, which are called alternative definitions. A recursive rule can be direct (e.g., $A ::= \alpha_1\ A\ \alpha_2$) or indirect (e.g., $A ::= \alpha_1\ B\ \alpha_2,\ \ B ::= \alpha_3\ A\ \alpha_4$). The primary input arguments of these functions are the input tokens. By the end of one successful function execution, the rule consumes some or all of the tokens. In case of a right recursive rule, before rewriting to its own definition again, one non-terminal consumes one

or more input token, which ensures a finite number of recursive calls. On the other hand, a left recursive rule does not consume any token before the next recursive call, and thus falls into an infinite loop without giving all alternatives any chance to try to recognize or parse any input. One way to accommodate left-recursion with top-down parsing is to convert the left-recursive grammar into an equivalent right-recursive grammar [8]. But this technique sacrifices the original syntactic structure that the left-recursive grammar would provide, and as a result correct natural language semantic computations that utilizes syntactic constructs' meanings may not be viable. Also, the transformation technique can introduce many new rules in the grammar [9] that will require extra parsing time.

We can infer from the concept of accommodating all possible parses as the result, which assumes that if all alternative definitions of a rule are applied to each input token then the rules attempt to compute potentially ambiguous parses is correct (see Appendix A for more discussion). This is possible when a left-recursive rule is stopped at a depth that provides just enough of the remaining input for all alternatives to be applied to all of the remaining input tokens systematically. In order to ensure this correct depth-restriction, our new top-down algorithm keeps track of recursive calls for the current token and the length of the input which is currently being processed. Each time a particular rule is being called during recursive-descent, a 'counter' is incremented by one indicating the depth of the parse. This parse branch is *curtailed* when the counter exceeds the length of the remaining input and the process backtracks to apply another alternative parse, if any exists.

This curtailment of left-recursive calls may cause the computation of incomplete results for *indirect left-recursive* rules. This is because some non-terminals, that reside between the curtailed path for an indirect left-recursive non-terminal, can compute different results for the same input token in different *context*s. As we use *memoization* to reuse a non-terminal's previously computed result for an input token, an intermediate non-terminal may return a result containing an incomplete result for the current input, even though it can potentially generates more elaborate result. To avoid this, we explicitly make sure that the intermediate non-terminals are able to compare the previously-saved context (where the result was computed) with the current context before reusing.

In this paper, termination of the parsing algorithm, and achieving polynomial time and space complexity are discussed informally. Some experimental comparisons of the al-

gorithm's initial implementation with Tomita's [10] and Earley's [11] parsing algorithms, using four practical natural-language grammars, are also performed. This comparison implies that the time and space requirements of the algorithm match well against Tomita'a and Earley's, which suggests similar polynomial upper bound of complexities in these three algorithms. The algorithm is also tested with four variations (i.e., non-memoized, memoized left-recursive, memoized right-recursive, and highly-memoized versions) of a massively-ambiguous grammar from Aho and Ullman [7], and it is evident that the algorithm can generate billions of ambiguous parses compactly within a second when the grammars are memoized, whereas the non-memoized version quickly runs out of space.

1.4 Efficient Combinators for General Syntax

In functional programming, use of a higher-order function as an infix operator in a function-definition is known as a 'combinator'. A parsing method, which is constructed using combinators, is called 'combinatory-parsing' (as higher-order functions combine different parsers together). A language-processor can be constructed by combining small processors with combinators. Though the concept of combinatory-parsing was introduced by Burge in 1975 [12], it was Wadler [13] who first popularized this form of parsing. Wadler showed that results (*success* or *failure*) of a recognizer can be returned as a list. Multiple entries of this list represent ambiguous results, whereas an empty list represents a failure.

For programming languages, parsers are generated automatically using tools like *Lex* and *Yacc* (for imperative languages) or *Happy* (for functional language Haskell). One drawback of this approach is the user needs to learn a new language (Lex, Yacc or Happy) to generate a parser. Parser combinators are written and used within the same programming language as the rest of the program. As function application in functional languages is *juxtaposition*, a language-processor written using combinators can be very close in structure to the BackusNaur Form (BNF) representation of a contextfree grammar. By nature, a combinatory-parsing system is a top-down, recursive-descent (with full backtracking), which is able to accommodate ambiguity. These parser-combinators are straightforward to construct, readable, modular, well-structured, and easily maintainable and alterable. However, a naive implementation also shares the same drawbacks of naive top-down parsing e.g., they

exhibit exponential time and space complexities at worst case when processing ambiguous inputs, and they do not terminate when used to represent a left-recursive production rule. Moreover, the traditional combinators are mostly used as recognizers [14, 15], and not as complete parsing system.

In **Chapter 3** I have included our paper that describes a general top-down parsing algorithm with the help of a set of combinators to take advantage of the modular, declarative and directly-executable characteristics of parser-combinators. Combinators for sequencing of symbols (i.e., `*>`) and for alternative grammar rules (i.e., `<|>`) are defined in the purely-functional language *Haskell* ([16]) to form executable context-free syntax rules. Instead of the more conventional input as a sequence of tokens, these definitions of combinators use *indices*. Parser execution is now a mapping from the starting index of an input ($Start$) to a set of ending indices (End) of the input. This optimized approached is further improved by the use of a Haskell library `IntMap` for sets of integers.

This paper shows the systematic development of our combinators from recognizers to fully fledged parsers. The underlying definitions of these combinators now include techniques (from Chapter 2) for accommodating direct and indirect left-recursion, a memoization to ensure efficiency while processing highly ambiguous grammars, and a recursive data-structure to represent potentially exponential number of parses in a compact space. The data-structure that represents the graph of parse trees is of the following form:

$$
\begin{aligned}
\mathit{type}\ \ \mathit{Start/End} &= \mathit{Int} \\
\mathit{data}\ \ \mathit{PTree}\ \ \mathit{Label} &= \mathit{Leaf}\ \ \mathit{Label} \\
&\mid\ \mathit{Branch}[\mathit{PTree}\ \ \mathit{Label}] \\
&\mid\ \mathit{SubNode}(\mathit{Label}, (\mathit{Start}, \mathit{End}))
\end{aligned}
$$

The overall result of a non-terminal (that is commonly referred to as a *parser*) is a list of the $PTree$ instances - $Result = [((Start, End), [PTree\ Label])]$. We form a *memo-table* with a list of parse names along with respective $Result$s, and before every parser execution, the parser checks this table to find a previously computed result for the current input. This implies that every parser should have access to the most-current memo-table. The members of a result have $(Start, End)$ pairs that work as points to *where to go next* while

constructing parse trees. Ambiguous entries are merged as one entry in the memo-table so that they are shared between many other entries that have pointers to ambiguous entries.

As non-strict and pure functional languages (e.g., Haskell) do not permit side-effects (such as: assignments, exceptions, continuation etc), it is relatively complex to perform operations like IO, maintaining states, raising exceptions, error handling etc. The concept of a monad, which abstractly describes mathematical structures (categories) and relations between them, appears as an easy solution to these kinds of problems. Moggi [17] and other researches demonstrated that maintaining states, raising exceptions, error handling etc. can be structurally performed using monads. Wadler established monads as a convenient tool for structuring functional programs [18]. As a continuously changing state (or the memo-table) has to be maintained for all parser executions, we opt to use a version of the state-monad so that function-definitions can abstract the updated memo-table as a function parameter implicitly. A changing state-variable can be passed around within monadic function-definitions without forcing the functions to explicitly operate on it. One simple way to define a state-monad is as follows:

```
type State a = S -> (a, S)

unit :: a -> State a
unit x = \s -> (x, s)

bind :: State a -> (a -> State b)-> State b
m ‘bind‘ k = \x -> let (p, y) = m x in
                                let (q, z) = k p y in (q, z)
```

The *unit* and *bind* form the state monad. The output type of the *unit* is of type `State a`; that means a value of type `a` is contained in a state. The *bind* takes two parameters - the first one, the container `m`, is of type `State a` and the second one, `k`, which is a computation that takes a value of type `a` (from `m`) and returns a new state with the result of the computation on the value of the type `a`. Our definitions of combinators satisfy the definitions of *unit* and *bind*, the contained value in state is now of the type memo-table, and the required

computations in the *bind* are now operations for `*>` and `<|>`.

The optimized implementation of the general top-down parsing algorithm as combinatory parsers and by using faster Haskell libraries helped us to perform experiments to utilize our approach in parsing large-scale practical natural language grammars. For example, while parsing the grammar for the Air Travel Information System maintained by Carroll [19], that consists of 5,226 rules with 258 nonterminals and 991 terminals, our implementation needs, on average, 1.88 seconds to parse ambiguous sentences from the test-set of 98 NL sentences.

1.5 Fully-General Attribute Grammars

Chapter 4 includes a paper where I describe the extension of the efficient and modular parsing algorithm (chapters 2 and 3) not only for analyzing general context-free syntax but also for accommodating declarative semantic rules. These semantic rules can be formed by establishing arbitrary dependencies between syntactic constructs in a syntax rule of a context-free grammar. Knuth [20] proposed a formalism, called Attribute Grammar (AG), where the meaning of a formal language can be described in terms of the semantics associated with the context-free syntax. In an AG, syntax rules of CFGs are augmented with semantic rules to describe the meaning of the sentences of a context-free language. Although different definitions are given, we prefer to define a general AG by imposing minimal restrictions on attribute dependencies. An AG is a 3-tuple $AG = (G, A, R)$, where G is the founding CFG, A is a finite set of attributes and R is a finite set of semantic rules. Each $X \in (N \cup T)$ is associated with a set of attributes $A(X) \subset A$, and each $a \in A(X)$ can be described by a function/expression $r \in R$. The set $A(X)$ can be partitioned into two sets $A_i(X)$ and $A_s(X)$, which represent *inherited* and *synthesized* attributes respectively. A synthesized attribute is an attribute for the LHS non-terminal of a production rule, and it can be computed using any attributes from the RHS terminals/non-terminals. Whereas an inherited attribute is associated with a terminal/non-terminal that resides at the RHS of the production rule, which can be computed attributes of terminals/non-terminals that are on the right or on the left of the current non-terminal. In conventional term, inherited attributes propagate downwards and synthesized attributes propagate upwards.

AG-based systems have been mostly used for the specification and construction of compilable parser-generators for programming languages [21, 22, 23, 24]. These systems do not need to accommodate ambiguity, general CFGs, and the preservation of original syntactic structures. We differ from them by providing a declarative notation to build AG-like executable specifications that strictly preserve the syntactic structure of general and ambiguous CFGs. Our intended outcome is to take advantage of the modular top-down analyzing approach to build natural-language specifications where syntax and semantic descriptions can be constructed and executed piecewise without affecting the rest of the specification in a one-pass evaluation. Note that, most of the existing AG evaluation techniques require in two-passes that may require the evaluator to do extra work.

We provide a set of combinators that can be used to build directly-executable AGs following conventional notation. For example, a text-book description of an AG for computing values of simple arithmetic expressions is:

$$
\begin{array}{ll}
expr(E_0) & ::= expe(E_1)\ \ op\ \ (O_0)\ \ expr(E_2) \\
 & \{E_0.val \uparrow = O_0.val \uparrow (E_1.val \uparrow, E_2.val \uparrow)\} \\
 & |\quad num(N_0) \\
 & \{E_0.val \uparrow = N_0.val \uparrow\} \\
op(O_0) & ::= + \\
 & \{O_0.val \uparrow = \lambda x \lambda y (x + y)\} \\
 & |\quad * \\
 & \{O_0.val \uparrow = \lambda x \lambda y (x * y)\} \\
 & |\quad \texttt{etc.} \\
num(N_0) & ::= 1 \\
 & \{N_0.val \uparrow = 1\} \\
 & |\quad 2 \\
 & \{N_0.val \uparrow = 2\} \\
 & |\quad \texttt{etc.}
\end{array}
$$

In the above, the underlying CFG is left and right recursive, ambiguous, and each syntax rule (e.g., $expr(E_0) ::= expe(E_1)\ op\ (O_0)\ expr(E_2)$) is coupled with a set of semantic rules (e.g., $\{E_0.val \uparrow = O_0.val \uparrow (E_1.val \uparrow, E_2.val \uparrow)\}$). The semantics can be evaluated by

syntactic constructs' upward-propagating values (i.e., *synthesized attributes*) or downward propagating values (i.e., *inherited attributes*). Using the combinators `<|>`, `*>`, `parser`, `nt` etc. we can create notationally-similar executable-specifications of the above AG as follows (see Chapter 4 for the descriptions of these combinators):

```
expr = memoize Expr
      (parser (nt expr E1 *> nt op O1 *> nt expr E2)
      [rule_s VAL OF LHS ISEQUALTO applyBiOp [synthesized VAL  OF E1
                                            , synthesized B_OP OF O1
                                            , synthesized VAL  OF E2]]
      <|>
      parser (nt num N1)
      [rule_s VAL OF LHS  ISEQUALTO copy [synthesized Val OF N1]] )

op   = memoize Op
      (terminal (term "+") [B_OP (+)] <|>
       terminal (term "*") [B_OP (*)] <|>  etc.
       )

num  = memoize Num
      ( terminal (term "1") [Val 1] <|> terminal (term "2") [Val 2] <|> etc.
```

When *expr* is applied to inputs such as `1*2-3+5`, the executable specification can produce either just the result of the arithmetic expression or complete parse trees with computed values attached in respective nodes. Along with utilizing the efficient top-down parsing algorithm and a new way to organize the data-structure to process both syntax and semantics, our approach also depends on non-strict evaluation (or lazy-evaluation or call-by-need evaluation) strategy that is an integral part of modern purely-functional languages such as Haskell [25]. This strategy ensures that a function is not evaluated until it is actually needed, and implicitly shares results of computations (i.e., outermost graph reduction [26]).

We treat the attributes as unevaluated pure-functions, group synthesized and inherited functions for respective non-terminals, and map the synthesized functions to the ending positions and inherited functions to the starting positions of respective non-terminals' result-sets. Now the result-set contains unevaluated semantic rules instead of attribute values, and they are evaluated only when they are ultimately required in other functions' executions. This method allows to establish arbitrary dependencies (including dependencies from the right, and except purely circular dependencies) between syntactic constituents in a syntax rule, which is useful in creating linguistically motivated semantic rules. Utilizing this technique, I have constructed domain-specific natural language query processor that is able to answer hundreds of thousands of natural-language questions. See Appendix B for the Haskell code for this query processor.

1.6 NLP Tasks with Modular AGs

Chapter 5 includes a paper that describes applications of our modular and executable attribute grammars (AGs) (chapter 4) for various natural-language processing (NLP) tasks. These tasks are performed with arbitrary semantics by establishing various dependencies on syntactic constructs. This approach demonstrates declarative prototyping of compositional semantics of natural language. During the nineteen seventies in a series of papers Richard Montague advocated that syntax and semantics of natural languages should be treated similar to formal languages. His approach (better known as Montague semantic) computes meaning of a NL phrase by primarily applying higher-order predicate logic and lambda calculus on syntactic phrases' meanings. Using our declarative notation for AG, we show how an efficient set-theoretic version of Montague's compositional semantics [27] and the associated CFGs can be integrated to build descriptions of English for computing meaning.

Constraint-based approaches [28, 29] have been used extensively in grammar formalisms to fulfill many linguistic requirements. The main objective of these formalisms is to unify grammatical information in atomic or composite grammatical units (e.g., words, phrases, sentences) by checking or imposing equality expressions on their associated sets of attribute-value pairs. We propose an innovative technique to impose unification-like restrictions on the results of the top-down syntactic and semantic analysis. By taking advantages of

declarative semantics, we introduce some special-purpose attributes for non-terminals that are capable of discarding parses that are not satisfied by the condition-checking semantics. This approach allows us to model unification-like formalisms as executable specification to impose linguistic restrictions such as grammatical, and number/gender agreements etc.

This form of semantic-based pruning of syntactic structures can also be used to model cross-serial and long-distance dependencies between syntactic categories that are necessary for accommodating certain languages, which can not be modelled by CFGs [30]. For example, cross-serial dependencies in Dutch, Swiss-German [31] and long-distance dependencies in relative clauses need a slightly more powerful formalism than CFGs. We can write declarative restrictive semantic to prototype processors for such languages by imposing conditions on associated syntax rules. We also show that this type of agreement can be used to disambiguate some types of syntactic ambiguity by using linguistically-sound semantics.

In addition to the informal discussion on various properties of the overall top-down analysis in individual chapters, Appendix A contains more elaborate discussions on these properties. These properties include the termination analysis, time and space complexity analysis, and correctness analysis of the parsing algorithm. As mentioned earlier, Appendix B contains a sample natural-language database query application, which is constructed the declarative notation described in this thesis. This application demonstrates the seamless construction of language processors without mastering the underlying analysis technique.

The format of citations and references in the subsequent chapters are different to that in the original papers. One uniform format for citations and references is used in all chapters to conform to the university requirements for doctoral theses. Also, `<|>`, `<+>`, and `‘orelse‘` are used interchangeably to represent the combinator for alternative rules, and `*>` and `‘thenS‘` are used interchangeably to represent the combinator for sequencing of symbols throughout the report.

1.7 Distinguishing Features for Effective NLP Applications

In this section we discuss some of the features of our system that provide advantages to software developers over other techniques in building effective natural language processing applications.

Modularity of Top-down Parsing

A top-down process satisfies a fundamental requirement for effective software development - modularity or *separation of concerns*. Components of a top-down process can be easily added or edited without affecting the rest of the system. For example, in a top-down parser, new alternatives (to a production rule) can be created and tested separately and then added to a larger parser. In this way, the top-down parser can be constructed piecewise (see introductory examples in Chapter 5). In our top-down system, a new component consisting of syntactic and the related semantic rules can be developed and tested separately and then added as a new alternative. In comparison, a general bottom-up parsing technique requires maintaining an action table, which needs global change whenever a new rule is added. In order to accommodate the semantic rules associated with a new syntax rule, the developer needs to make extensive changes to the whole parser. This dependency between one component (the implementation of the processing of the syntax and semantics associated with a new rule) and the existing parser code violates modularity.

Ambiguity w.r.t. Natural Language Processing

While parsing programming languages, the compiler constructs only one syntax tree. Ambiguity is undesirable as it may generate more than one machine-code output for program. However, natural language (NL) is inherently ambiguous. Many NL sentences have more than one meaning. This is why we want to generate all possible parses while processing a NL sentence. Each ambiguous parse can potentially generate a different meaning, and semantic processing is needed to determine which meaning is most likely to be the one intended. Converting a left-recursive grammar into a non-left recursive form can, in some cases, cause the loss of some parses (e.g., missing left-branching parses; see the example in

Chapter 2 at page 23), and as a result, we may miss the intended meaning of the sentence. This is why we want to accommodate left-recursive rules directly rather than transform to non-left-recursive form.

Declarative Specifications

Our declarative combinatory programming approach allows us to build language processors as executable specifications of syntax and semantics, which are very similar to the standard declarative Attribute Grammar notation that is frequently used in textbooks (for example) to describe languages. Owing to the fact that declarative notation does not specify the order in which the processing associated with the description is to be carried out, reasoning about the specification is relatively straight forward as equational reasoning and inductive proofs can be used, rather than more complicated reasoning systems such as temporal logic.

Lazy Evaluation

Owing to the fact that lazy (or non-strict) evaluation delays computations until their results are needed, our functional combinators allow us to specify arbitrary dependencies between semantic attributes (including inheritance from the right, where an attribute of a syntactic component on the right hand side of a production rule can be defined in terms of an attribute of some syntactic component on its right). The declarative nature of the programming language, together with lazy evaluation, frees us from considering the order in which the computations will take place.

In addition, lazy evaluation means that no attribute computation takes place until required. Therefore even though the semantic rules and the syntactic rules are tightly integrated (each executable production rule can be tested separately) no semantic action takes place unless a syntactic parse is successful, and only then when the user requires the result. For example, suppose if the user only wants the results of the first successful parse, then no other attribute computations are carried out. This can be achieved in a non-lazy language only through having a two-pass process (create the syntax tree and then compute values for chosen trees - which results in a separation of the syntax and semantics for each rule) or implement a complex process that mimics laziness.

The Kill Attribute

Our approach allows us to impose restrictions with the Kill attribute. As our evaluation strategy is lazy, this Kill attribute prevents our one-pass parsing method from generating unneeded trees, which would otherwise be generated if we had used a two-pass strategy. Although some capabilities of the Kill attribute (phenomenon that can be accommodated by CFGs e.g., number-gender agreements) can be achieved in a two-pass strategy by adding extra grammar rues, that would require extra computational time for processing the added rules.

Memoization

Memoization causes the reuse of results of functions that have been previously computed. The use of the functional state-monad allows the procedural aspects of memoization to be hidden in the executable specification. In fact, the developer of the executable specification does not even require knowing that memoization is taking place. In particular, there is no effect on the modularity of the specification, which mirrors the modularity of the grammar.

The use of memoization is quite different to dynamic programming where the developer needs to think about which results are going to be reused and then develops the program accordingly. Thus, the structure of the program is affected by the reuse of results. This is not the case with memoization, which does not influence the structure of the program at all (other than the insertion of the combinatory "memoize" in front of functions which are to be memoized.

Bibliography

[1] Frost, R., Hafiz, R.: A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. SIGPLAN Notices **41(5)** (2006) 46–54

[2] Frost, R., Hafiz, R., Callaghan, P.: Modular and efficient top-down parsing for ambiguous left-recursive grammars. ACL-IWPT (2007) 109 – 120

[3] Hafiz, R.: Executable specifications of fully general attribute grammars with ambiguity and left-recursion. In: Proceedings of the 22nd Canadian Conference on AI 2009, Extended Abstract . (2009) 274–278

[4] Hafiz, R., Frost, R.A.: Lazy combinators for executable specifications of general attribute grammars. In: ACM-PADL. (2010) 167–182

[5] Hafiz, R., Frost, R.A.: A system for modularly constructing efficient natural language processors. In: Computational linguistics-Applications Conference. (2011)

[6] Church, K., Patil, R.: Coping with syntactic ambiguity or how to put the block in the box on the table. Computational Linguistics **8, (3-4)** (1982) 139–149

[7] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall (1972)

[8] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley Longman Publishing Co. (1986)

[9] Moore, R.C.: Removing left recursion from context-free grammars. In: Proceedings, 1st Meeting of the North American Chapter of the ACL. (2000)

[10] Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1986)

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

[12] Burge, W.H.: Recursive Programming Techniques. Addison-Wesley. (1975)

[13] Wadler, P.: How to replace failure by a list of successes. In: FP languages and computer architecture. Volume 201. (1985) 113 – 128

[14] Frost, R., Launchbury, J.: Constructing natural language interpreters in a lazy functional language. The Computer Journal **32(2)** (1989) 108–121

[15] Hutton, G., Meijer, E.: Monadic parser combinators. J. Funct. Program. **8(4)** (1998) 437 – 444

[16] Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of haskell: being lazy with class. Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III) (2007)

[17] Moggi, E.: Computational lambda-calculus and monads. In: In Symposium on Logic in Computer Science, Asilomar, California; IEEE. (1989) 14–23

[18] Wadler, P.: Monads for functional programming. 1st Spring School on Advanced FP **925** (1995) 24 – 52

[19] Carroll, J.: Efficiency in large-scale parsing systems -parser comparison. (2003)

[20] Knuth, D.: Semantics of context-free languages. Theory of Computing Systems, Springer New York **2**(2) (1968) 127–145

[21] Paakki, J.: Attribute grammar paradigms a high-level methodology in language implementation. ACM Comput. Survey **27**(2) (1995) 196–255

[22] Ekman, T.: Extensible Compiler Construction. PhD thesis, Comp Science, Lund University (2006)

[23] Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. In: LDTA. (2007) 103 – 116

[24] Kats, L., Sloane, A., Visser, E.: Decorated attribute grammars. attribute evaluation meets strategic programming. In: 18th International Conference on Compiler Construction. (2009) 142 – 157

[25] Hughes, J.: Why functional programming matters. The Computer Journal - Special issue on lazy functional programming. **32, (2)** (1989) 1742

[26] Bird, R.: Intro. to Functional Programming using Haskell. Prentice Hall (1998)

[27] Frost, R., Fortier, R.: An efficient denotational semantics for natural language database queries. In: Applications of NLDB. (2007) 12–24

[28] Shieber, S.: An introduction to unification-based theories of grammar,. CSLI Lecture Notes Series, University of Chicago Press. (1986)

[29] Shieber, S.: Constraint-based grammar formalisms,. The MIT Press, Cambridge, Massachusetts. (1992)

[30] Joshi, A.K.: Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? NL Parsing, Cambridge University Press (1985)

[31] Shieber, S.: Evidence against the context-freeness of natural language. Linguistics and Philosophy **8, (3)** (1982) 333–343

Chapter 2

Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars

This paper was published as:

Frost, R., Hafiz, R. and Callaghan, P. (2007) *Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars*. Proceedings of the 10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE. Pages: 109 - 120, June 2007, Prague.

2.1 Introduction

Top-down parsers can be built as a set of mutually-recursive processes. Such implementations are modular in the sense that parsers for terminals and simple non-terminals can be built and tested first. Subsequently, parsers for more complex non-terminals can be constructed and tested. Koskimies (1990)[1], and Nederhof and Koster (1993) [2] discuss this and other advantages of top-down parsing.

In functional and logic programming, top-down parsers can be built using parser combinators (e.g. see [3] for a discussion of the origins of parser combinators, and [4] for a discussion of their use in natural-language processing) and definite clause grammars (DCGs) respectively. For example, consider the following grammar, in which `s` stands for sentence, `np` for nounphrase, `vp` for verbphrase, and `det` for determiner:

```
s    ::= np vp
np   ::= noun | det noun
vp   ::= verb np
det  ::= 'a' | 't'
noun ::= 'i' | 'm' | 'p' | 'b'
verb ::= 's'
```

A set of parsers for this grammar can be constructed in the Haskell functional programming language as follows, where `term, ‘orelse‘, and ‘thenS‘` are appropriately-defined higher-order functions called parser combinators. (Note that backquotes surround infix functions in Haskell).

```
s    = np ‘thenS‘ vp
np   = noun ‘orelse‘ (det ‘thenS‘ noun)
vp   = verb ‘thenS‘ np
det  = term 'a' ‘orelse‘ term 't'
noun = term 'i' ‘orelse‘ term 'm'
                ‘orelse‘ term 'p'
                ‘orelse‘ term 'b'
```

```
verb = term 's'
```

Note that the parsers are written directly in the programming language, in code which is similar in structure to the rules of the grammar. As such, the implementation can be thought of as an executable specification with all of the associated advantages. In particular, this approach facilitates modular piecewise construction and testing of component parsers. It also allows parsers to be defined to return semantic values directly instead of intermediate parse results, and parsers to be parameterized in order to accommodate context-sensitive languages (e.g. [5]). Also, in functional programming, the type checker can be used to catch errors in parsers attributed with semantic actions.

Parser combinators and DCGs have been used extensively in applications such as prototyping of compilers, and the creation of natural language interfaces to databases, search engines, and web pages, where complex and varied semantic actions are closely integrated with syntactic processing. However, both techniques are based on top-down recursive descent search with backtracking. Commonly used implementations have exponential complexity for ambiguous languages, cannot handle left-recursion, and do not produce compact representations of parse trees. (Note, a left-recursive grammar is one in which a non-terminal `p` derives an expansion `p ..` headed with a `p` either directly or indirectly. Application of a parser for such a grammar results in infinite descent.) These shortcomings limit the use of parser combinators and DCGs especially in natural-language processing.

The problem of exponential time complexity in top-down parsers constructed as sets of mutually-recursive functions has been solved by Norvig (1991) [6] who uses memotables to achieve polynomial complexity. Norvig's technique is similar to the use of dynamic programming and state sets in Earley's algorithm [7], and tables in the CYK algorithm of Cocke, Younger and Kasami. The basic idea in Norvig's approach is that when a parser is applied to the input, the result is stored in a memotable for subsequent reuse if the same parser is ever reapplied to the same input. In the context of parser combinators, Norvig's approach can be implemented using a function `memoize` to selectively "memoize" parsers.

In some applications, the problem of left-recursion can be overcome by transforming the grammar to a weakly equivalent non-left-recursive form. (i.e. to a grammar which derives the same set of sentences). Early methods of doing this resulted in grammars

that are significantly larger than the original grammars. This problem of grammar size has been solved by Moore [8] who developed a method, based on a left-corner grammar transformation, which produces non-left recursive grammars that are not much larger than the originals. However, although converting a grammar to a weakly-equivalent form is appropriate in some applications (such as speech recognition) it is not appropriate in other applications. According to Aho, Sethi, and Ullman [9] converting a grammar to non-left recursive form makes it harder to translate expressions containing left-associative operators. Also, in NLP it is easier to integrate semantic actions with parsing when both leftmost and rightmost parses of ambiguous input are being generated. For example, consider the first of the following grammar rules:

```
np   ::= noun  | np conj np
conj ::= "and" | "or"
noun ::= "jim" | "su" | "ali"
```

and its non-left-recursive weakly equivalent form:

```
np  ::= noun np'
np' ::= conj np np' | empty
```

The non-left-recursive form loses the leftmost parses generated by the left-recursive form. Integrating semantic actions with the non-left-recursive rule in order to achieve the two correct interpretations of input such as `["john", "and", "su", "or", "ali"]` is significantly harder than with the left-recursive form.

Several researchers have recognized the importance of accommodating left-recursive grammars in top-down parsing, in general and in the context of parser combinators and DCGs in particular, and have proposed various solutions. That work is described in detail in section 2.3.

In this paper, we integrate Norvig's technique with aspects of existing techniques for dealing with left recursion. In particular: a) we make use of the length of the remaining input as does Kuno [10], b) we keep a record of how many times each parser is applied to each input position in a way that is similar to the use of cancellation sets by Nederhof and Koster

[2], c) we integrate memoization with a technique for dealing with left recursion as does Johnson [11], and d) we store "left-recursion counts" in the memotable, and encapsulate the memoization process in a programming construct called a monad, as suggested by Frost and Hafiz [12].

Our method includes a new technique for accommodating indirect left recursion which ensures correct reuse of stored results created through curtailment of left-recursive parsers. We also modify the memoization process so that the memotable represents the potentially exponential number of parse trees in a compact polynomial sized form using a technique derived from the chart parsing methods of Kay [13] and Tomita [14].

As an example use of our method, consider the following ambiguous left-recursive grammar from Tomita (1985) in which `pp` stands for prepositional phrase, and `prep` for preposition. This grammar is left recursive in the rules for `s` and `np`. Experimental results using larger grammars are given later.

```
s    ::= np vp | s pp
np   ::= noun | det noun | np pp
pp   ::= prep np
vp   ::= verb np
det  ::= 'a' | 't'
noun ::= 'i' | 'm' | 'p' | 'b'
verb ::= 's'
prep ::= 'n' | 'w'
```

The Haskell code below defines a parser for the above grammar, using our combinators:

```
s    = memoize "s"  ((np  `thenS` vp) `orelse` (s `thenS` pp))
np   = memoize "np" (noun `orelse` (det `thenS` noun)
                         `orelse` (np  `thenS` pp))

pp   = memoize "pp"   (prep `thenS` np)
```

```
vp   = memoize "vp"   (verb `thenS` np)
det  = memoize "det"  (term 'a'   `orelse` term 't')
noun = memoize "noun" (term 'i'   `orelse` term 'm'
                                  `orelse` term 'p'
                                  `orelse` term 'b')
verb = memoize "verb" (term 's')
prep = memoize "prep" (term 'n'   `orelse` term 'w')
```

The following shows the output when the parser function `s` is applied to the input string `"isamntpwab"`, representing the sentence "I saw a man in the park with a bat". It is a compact representation of the parse trees corresponding to the several ways in which the whole input can be parsed as a sentence, and the many ways in which subsequences of it can be parsed as nounphrases etc. We discuss this representation in more detail in subsection 2.4.4.

```
apply s "isamntpwab" =>

"noun"
  1  ((1,2),   [Leaf "i"])
  4  ((4,5),   [Leaf "m"])
  7  ((7,8),   [Leaf "p"])
 10  ((10,11), [Leaf "b"])
"det"
  3  ((3,4),   [Leaf "a"])
  6  ((6,7),   [Leaf "t"])
  9  ((9,10),  [Leaf "a"])
"np"
  1 ((1,2), [SubNode ("noun", (1,2))])

  3 ((3,5), [Branch [SubNode ("det", (3,4)), SubNode ("noun",(4,5))]])
    ((3,8), [Branch [SubNode ("np",  (3,5)), SubNode ("pp",  (5,8))]])
    ((3,11),[Branch [SubNode ("np",  (3,5)), SubNode ("pp",  (5,11))],
             Branch [SubNode ("np",  (3,8)), SubNode ("pp",  (8,11))]])

  6 ((6,8), [Branch [SubNode ("det", (6,7)), SubNode ("noun",(7,8))]])
    ((6,11),[Branch [SubNode ("np",  (6,8)), SubNode ("pp",  (8,11))]])

  9 ((9,11),[Branch [SubNode ("det", (9,10)),SubNode ("noun",(10,11))]])
"prep"
```

```
  5 ((5,6), [Leaf "n"])
  8 ((8,9), [Leaf "w"])
"pp"
  8 ((8,11),[Branch [SubNode ("prep",(8,9)), SubNode ("np",  (9,11))]])

  5 ((5,8), [Branch [SubNode ("prep",(5,6)), SubNode ("np",  (6,8))]])
    ((5,11),[Branch [SubNode ("prep",(5,6)), SubNode ("np",  (6,11))]])
"verb"
  2 ((2,3), [Leaf "s"])
"vp"
  2 ((2,5), [Branch [SubNode ("verb",(2,3)), SubNode ("np",  (3,5))]])
    ((2,8), [Branch [SubNode ("verb",(2,3)), SubNode ("np",  (3,8))]])
    ((2,11),[Branch [SubNode ("verb",(2,3)), SubNode ("np",  (3,11))]])
"s"
  1 ((1,5), [Branch [SubNode ("np",  (1,2)), SubNode ("vp",  (2,5))]])
    ((1,8), [Branch [SubNode ("np",  (1,2)), SubNode ("vp",  (2,8))],
             Branch [SubNode ("s",   (1,5)), SubNode ("pp",  (5,8))]])
    ((1,11),[Branch [SubNode ("np",  (1,2)), SubNode ("vp",  (2,11))],
             Branch [SubNode ("s",   (1,5)), SubNode ("pp",  (5,11))],
             Branch [SubNode ("s",   (1,8)), SubNode ("pp",  (8,11))]]
```

Our method has two disadvantages: a) it has $O(n^4)$ time complexity, for ambiguous grammars, compared with $O(n^3)$ for Earley-style parsers [7], and b) it requires the length of the input to be known before parsing can commence.

Our method maintains all of the advantages of top-down parsing and parser combinators discussed earlier. In addition, our method accommodates arbitrary context-free grammars, terminates correctly and correctly reuses results generated by direct and indirect left recursive rules. It parses ambiguous languages in polynomial time and creates polynomial-sized representations of parse trees.

In many applications the advantages of our approach will outweigh the disadvantages. In particular, the additional time required for parsing will not be a major factor in the overall time required when semantic processing, especially of ambiguous input, is taken into account.

We begin with some background material, showing how our approach relates to previous work by others. We follow that with a detailed description of our method. Sections 2.5, 2.6, and 2.7 contain informal proofs of termination and complexity, and a brief description of a Haskell implementation of our algorithm. Complete proofs are available in the Appendix

A, and the Haskell code are available from the author's site `cs.uwindsor.ca/~hafiz`.

We tested our implementation on four natural-language grammars from Tomita (1986), and on four abstract highly-ambiguous grammars. The results, which are presented in section 2.8, indicate that our method is viable for many applications, especially those for which parser combinators and definite clause grammars are particularly well-suited.

We present our approach with respect to parser combinators. However, our method can also be implemented in other languages which support recursion and dynamic data structures.

2.2 Top-Down Backtracking Recognition

Top-down recognizers can be implemented as a set of mutually recursive processes which search for parses using a top-down expansion of the grammar rules defining non-terminals while looking for matches of terminals with tokens on the input. Tokens are consumed from left to right. Backtracking is used to expand all alternative right-hand-sides of grammar rules in order to identify all possible parses. In the following we assume that the input is a sequence of tokens `input`, of length `l_input` the members of which are accessed through an index `j`. Unlike commonly-used implementations of parser combinators, which produce recognizers that manipulate subsequences of the input, we assume, as in Frost and Hafiz (2006), that recognizers are functions which take an index `j` as argument and which return a set of indices as result. Each index in the result set corresponds to the position at which the recognizer successfully finished recognizing a sequence of tokens that began at position `j` . An empty result set indicates that the recognizer failed to recognize any sequence beginning at j. Multiple results are returned for ambiguous input.

According to this approach, a recognizer `term_t` for a terminal `t` is a function which takes an index `j` as input, and if `j` is greater than `l_input`, the recognizer returns an empty set. Otherwise, it checks to see if the token at position `j` in the input corresponds to the terminal `t`. If so, it returns a singleton set containing `j + 1`, otherwise it returns the empty set. For example, a basic recognizer for the terminal `'s'` can be defined as follows (note that we use a functional pseudo code throughout, in order to make the paper accessible to

a wide audience. We also use a list lookup offset of 1):

```
term_s     = term 's'
where term t  j
           = {}     , if j > l_input
           = {j + 1}, if jth element of input = t
           = {}     , otherwise
```

The `empty` recognizer is a function which always succeeds returning its input index in a set:

```
empty j = {j}
```

A recognizer corresponding to a construct `p | q` in the grammar is built by combining recognizers for `p` and `q`, using the parser combinator `` `orelse` ``. When the composite recognizer is applied to index `j`, it applies `p` to `j`, then it applies `q` to `j`, and subsequently unites the resulting sets.:

```
(p `orelse` q) j = unite (p j) (q j)
```

e.g, assuming that the input is `"ssss"`, then

```
(empty `orelse` term_s) 2 => {2, 3}
```

A composite recognizer corresponding to a sequence of recognizers `p q` on the right hand side of a grammar rule, is built by combining those recognizers using the parser combinator `` `thenS` ``. When the composite recognizer is applied to an index `j`, it first applies `p` to `j`, then it applies `q` to each index in the set of results returned by `p`. It returns the union of these applications of `q`.

```
(p `thenS` q) j = union (map q (p j))
```

e.g., assuming that the input is `"ssss"`, then

```
(term_s ‘thenS‘ term_s) 1 => {3}
```

The combinators above can be used to define composite mutually-recursive recognizers. For example, the grammar `sS ::= 's' sS sS | empty` can be encoded as follows:

```
sS = (term_s ‘thenS‘ sS ‘thenS‘ sS) ‘orelse‘ empty
```

Assuming that the input is `"ssss"`, the recognizer `sS` returns a set of five results, the first four corresponding to proper prefixes of the input being recognized as an `sS`. The result `5` corresponds to the case where the whole input is recognized as an `sS`.

```
sS   1  => {1, 2, 3, 4, 5}
```

The method above does not terminate for left-recursive grammars, and has exponential time complexity with respect to `l_input` for non-left-recursive grammars. The complexity is due to the fact that recognizers may be repeatedly applied to the same index during backtracking induced by the operator `‘orelse‘`. We show later how complexity can be improved, using Norvig's memoization technique. We also show, in section 2.4.4, how the combinators `term, ‘orelse‘, and ‘thenS‘` can be redefined so that the processors create compact representations of parse trees in the memotable, with no effect on the form of the executable specification.

2.3 Left Recursion and Top-Down Parsing

Several researchers have proposed ways in which left-recursion and top-down parsing can coexist:

1) Kuno [10] was the first to use the length of the input to force termination of left-recursive descent in top-down parsing. The minimal lengths of the strings generated by the grammar on the continuation stack are added and when their sum exceeds the length

of the remaining input, expansion of the current non-terminal is terminated. Dynamic programming in parsing was not known at that time, and Kuno's method has exponential complexity.

2) Shiel [15] recognized the relationship between top-down parsing and the use of state sets and tables in Earley and SYK parsers and developed an approach in which procedures corresponding to non-terminals are called with an extra parameter indicating how many terminals they should read from the input. When a procedure corresponding to a non-terminal `n` is applied, the value of this extra parameter is partitioned into smaller values which are passed to the component procedures on the right of the rule defining `n`. The processor backtracks when a procedure defining a non-terminal is applied with the same parameter to the same input position. The method terminates for left-recursion but has exponential complexity.

3) Leermakers [16] introduced an approach which accommodates left-recursion through "recursive ascent" rather than top-down search. Although achieving polynomial complexity through memoization, the approach no longer has the modularity and clarity associated with pure top-down parsing. Leermakers did not extend his method to produce compact representations of trees.

4) Nederhof and Koster [2] introduced "cancellation" parsing in which grammar rules are translated into DCG rules such that each DCG non-terminal is given a "cancellation set" as an extra argument. Each time a new non-terminal is derived in the expansion of a rule, this non-terminal is added to the cancellation set and the resulting set is passed on to the next symbol in the expansion. If a non-terminal is derived which is already in the set then the parser backtracks. This technique prevents non-termination, but loses some parses. To solve this, for each non-terminal `n`, which has a left-recursive alternative 1) a function is added to the parser which places a special token $\underline{\mathtt{n}}$ at the front of the input to be recognized, 2) a DCG corresponding to the rule `n ::=` $\underline{\mathtt{n}}$ is added to the parser, and 3) the new DCG is invoked after the left-recursive DCG has been called. The approach accommodates left-recursion and maintains modularity. An extension to it also accommodates hidden left recursion which can occur when the grammar contains rules with empty right-hand sides. The shortcoming of Nederhof and Koster's approach is that it is exponential in the worst case and that the resulting code is less clear as it contains additional production rules and

code to insert the special tokens.

5) Lickman [17] defined a set of parser combinators which accommodate left recursion. The method is based on an idea by Philip Wadler in an unpublished paper in which he claimed that fixed points could be used to accommodate left recursion. Lickman implemented Wadler's idea and provided a proof of termination. The method accommodates left recursion and maintains modularity and clarity of the code. However, it has exponential complexity, even for recognition.

6) Johnson [11] appears to have been the first to integrate memoization with a method for dealing with left recursion in pure top-down parsing. The basic idea is to use the continuation-passing style of programming (CPS) so that the parser computes multiple results, for ambiguous cases, incrementally. There appears to have been no attempt to extend Johnson's approach to create compact representations of parse trees. One explanation for this could be that the approach is somewhat convoluted and extending it appears to be very difficult. In fact, Johnson states, in his conclusion, that "an implementation attempt (to create a compact representation) would probably be very complicated."

7) Frost and Hafiz [12] defined a set of parser combinators which can be used to create polynomial time recognizers for grammars with direct left recursion. Their method stores left-recursive counts in the memotable and curtails parses when a count exceeds the length of the remaining input. Their method does not accommodate indirect left recursion, nor does it create parse trees.

Our new method combines many of the ideas developed by others: as with the approach of Kuno (1965) we use the length of the remaining input to curtail recursive descent. Following Shiel (1976), we pass additional information to parsers which is used to curtail recursion. The information that we pass to parsers is similar to the cancellation sets used by Nederhof and Koster (1993) and includes the number of times a parser is applied to each input position. However, in our approach this information is stored in a memotable which is also used to achieve polynomial complexity. Although Johnson (1995) also integrates a technique for dealing with left recursion with memoization, our method differs from Johnson's $O(n^3)$ approach in the technique that we use to accommodate left recursion. Also, our approach facilitates the construction of compact representations of parse trees whereas Johnson's

appears not to. In the Haskell implementation of our algorithm, we use a functional programming structure called a monad to encapsulate the details of the parser combinators. Lickman's (1995) approach also uses a monad, but for a different purpose. Our algorithm stores "left-recursion counts" in the memotable as does the approach of Frost and Hafiz (2006). However, our method accommodates indirect left recursion and can be used to create parsers, whereas the method of Frost and Hafiz can only accommodate direct left recursion and creates recognizers not parsers.

2.4 The New Method

We begin by describing how we improve complexity of the recognizers defined in section 2.2. We then show how to accommodate direct and indirect left recursion. We end this section by showing how recognizers can be extended to parsers.

2.4.1 Memoization

As in Norvig [6] a memotable is constructed during recognition. At first the table is empty. During the process it is updated with an entry for each recognizer r_*i* that is applied. The entry consists of a set of pairs, each consisting of an index `j` at which the recognizer r_*i* has been applied, and a set of results of the application of r_*i* to `j`.

The memotable is used as follows: whenever a recognizer r_*i* is about to be applied to an index `j`, the memotable is checked to see if that recognizer has ever been applied to that index before. If so, the results from the memotable are returned. If not, the recognizer is applied to the input at index `j`, the memotable is updated, and the results are returned. For non-left-recursive recognizers, this process ensures that no recognizer is ever applied to the same index more than once.

The process of memoization is achieved through the function `memoize` which is defined as follows, where the `update` function stores the result of recognizer application in the table:

```
memoize label r_i j
 = if lookup label j succeeds,
```

```
        return memotable result

    else apply r_i to j,
        update table, and return results
```

Memoized recognizers, such as the following, have cubic complexity (see later):

```
msS = memoize "msS"((ms ‘thenS‘ msS ‘thenS‘ msS) ‘orelse‘ empty)
 ms = memoize "ms" term_s
```

2.4.2 Accommodating direct left recursion

In order to accommodate direct left recursion, we introduce a set of values c_*ij* denoting the number of times each recognizer r_*i* has been applied to the index `j`. For non-left-recursive recognizers this "left-rec count" will be at most one, as the memotable lookup will prevent such recognizers from ever being applied to the same input twice. However, for left-recursive recognizers, the left-rec count is increased on recursive descent (owing to the fact that the memotable is only updated on recursive ascent after the recognizer has been applied). Application of a recognizer `r` to an index `j` is failed whenever the left-rec count exceeds the number of unconsumed tokens of the input plus 1. At this point no parse is possible (other than spurious parses, which could occur with circular grammars — that we want to reject). As illustration, consider the following branch being created during the parse of two remaining tokens on the input (where `N`, `P` and `Q` are nodes in the parse search space corresponding to non-terminals, and `A`, `B` and `C` to terminals or non-terminals):

```
            N
           / \
          N   A
         / \
        N   B
       / \
      P   C
     /
    Q
   /
  N
```

The last call of the parser for `N` should be failed owing to the fact that, irrespective of what `A`, `B`, and `C` are, either they must require at least one input token, otherwise they must rewrite to `empty`. If they all require a token, then the parse cannot succeed. If any of them rewrite to `empty`, then the grammar is circular (`N` is being rewritten to `N`) and the last call should be failed in either case.

Note that failing a parse when a branch is longer than the length of the remaining input is incorrect as this can occur in a correct parse if recognizers are rewritten into other recognizers which do not have "token requirements to the right". For example, we cannot fail the parse at `P` or `Q` as these could rewrite to `empty` without indicating circularity. Also note that we curtail the recognizer when the left-rec count exceeds the number of unconsumed tokens *plus 1*. The plus 1 is necessary to accommodate the case where the recognizer rewrites to empty on application to the end of the input.

To make use of the left-rec counts, we simply modify the `memoize` function to refer to an additional table called `ctable` which contains the left-rec counts c_*ij*, and to check and increment these counters at appropriate points in the computation: if the memotable lookup for the recognizer r_*i* and the index `j` produces a result, that result is returned. However, if the memotable does not contain a result for that recognizer and that index, c_*ij* is checked to see if the recognizer should be failed because it has descended too far through left-recursion. If so, `memoize` returns an empty set as result with the memotable unchanged.

Otherwise, the counter c_*ij* is incremented and the recognizer r_*i* is applied to `j`, and the memotable is updated with the result before it is returned. The function `memoize` defined below, can now be applied to rules with direct left recursion.

```
memoize label r_i j  =
 if lookup label j succeeds
    return memotable results

 else if c_ij > (l_input)-j+1, return {}
     else increment c_ij, apply r_i to j,
          update memotable,
              and return results
```

2.4.3 Accommodating indirect left recursion

We begin by illustrating how the method described above may return incomplete results for grammars containing indirect left recursion.

Consider the following grammar, and subset of the search space, where the left and right branches represent the expansions of the first two alternate right-hand-sides of the rule for the non terminal `S`, applied to the same position on the input:

```
S ::= S then ..| Q | P | x       S
P ::= S then .                 /    \
Q ::= T                   S then ..  Q
T ::= P                   |          |
                          S then ..  T
                          |          |
                          P          P
                          |          |
                        S then..  S then ..
                         |
                       fail S
```

Suppose that the left branch occurs before the right branch, and that the left branch was failed due to the left-rec count for `S` exceeding its limit. The results stored for `P` on recursive ascent of the left branch would be an empty set. The problem is that the later call of `P` on the right branch should not reuse the empty set of results from the first call of `P` as they are incomplete with respect to the position of `P` on the right branch (i.e. if `P` were to be re-applied to the input in the context of the right branch, the results would not necessarily be an empty set). This problem is a result of the fact that `S` caused curtailment of the results for `P` as well as for itself. This problem can be solved as follows:

1) Pass left-rec contexts down the parse space. We need additional information when storing and considering results for reuse. We begin by defining the "left-rec-context" of a node in the parse search space as a list of the following type, containing for each index, the left-rec count for each recognizer, including the current recognizer, which have been called in the search branch leading to that node:

```
[(index,[(recog_label,left_rec_count)])]
```

2) Generate the reasons for curtailment when computing results. For each result we need to know if the subtrees contributing to it have been curtailed through a left-rec limits, and if so, which recognizers, at which indices, caused the curtailment. A list of `(recog_label,`

`index)` pairs which caused curtailment in any of the subtrees is returned with the result. `‘orelse‘` and `‘thenS‘` are modified, accordingly, to merge these lists, in addition to merging the results from subtrees.

3) Store results in the memotable together with a subset of the current left-rec context corresponding to those recognizers which caused the curtailment. When a result is to be stored in the memotable for a recognizer `P`, the list of recognizers which caused curtailment (if any) in the subtrees contributing to this result is examined. For each recognizer `S` which caused curtailment at some index, the current left-rec counter for `S` at that index (in the left-rec context for `P`) is stored with the result for `P`. This means that the only part of the left-rec context of a node, that is stored with the result for that node, is a list of those recognizers and current left-rec counts which had an effect on curtailing the result. The limited left-rec context which is stored with the result is called the "left-rec context of the result".

4) Consider results for reuse. Whenever a memotable result is being considered for reuse, the left-rec-context of that result is compared with the left-rec-context of the current node in the parse search. The result is only reused if, for each recognizer and index in the left-rec context of the result, the left-rec-count is smaller than or equal to the left-rec-count of that recognizer and index in the current context. This ensures that a result stored for some application `P` of a recognizer at index `j` is only reused by a subsequent application `P'` of the same recognizer at the same position, if the left-rec context for `P'` would constrain the result more, or equally as much, as it had been constrained by the left-rec context for `P` at `j`. If there were no curtailment, the left-rec context of a result would be empty and that result can be reused anywhere irrespective of the current left-rec context.

2.4.4 Extending recognizers to parsers

Instead of returning a list of indices representing successful end points for recognition, parsers also return the parse trees. However, in order that these trees be represented in a compact form, they are constructed with reference to other trees that are stored in the memotable, enabling the explicit sharing of common subtrees, as in Kay's [13] and Tomita's [14] methods. The example in section 2.1 illustrates the results returned by a parser.

Parsers for terminals return a leaf value together with an endpoint, stored in the memotable as illustrated below, indicating that the terminal `"s"` was identified at position 2 on the input:

```
"verb" 2 ((2,3),[Leaf "s"])
```

The combinator `‘thenS‘` is extended so that parsers constructed with it return parse trees which are represented using reference to their immediate subtrees. For example:

```
"np" ......
3 ((3,5),[Branch[SubNode("det", (3,4)),
                 SubNode("noun",(4,5))]])
```

This memotable entry shows that a parse tree for a nounphrase `"np"` has been identified, starting at position `3` and finishing at position `5`, and which consists of two subtrees, corresponding to a determiner and a noun.

The combinator `‘orelse‘` unites results from two parsers and also groups together trees which have the same begin and end points. For example:

```
"np" .........
3 ((3,5), [Branch[SubNode("det", (3,4)),   SubNode("noun",(4,5))]])

  ((3,8), [Branch[SubNode("np",  (3,5)),  SubNode("pp",  (5,8))]])

  ((3,11),[Branch[SubNode("np",  (3,5)),  SubNode("pp",  (5,11))],
          Branch[SubNode("np",  (3,8)),  SubNode("pp",  (8,11))]])
```

which shows that four parses of a nounphrase `"np"` have been found starting at position 3, two of which share the endpoint 11.

An important feature is that trees for the same syntactic category having the same start/end points are grouped together and it is the group that is referred to by other trees of which it is a constituent. For example, in the following the parse tree for a `"vp"` spanning positions 2 to 11 refers to a group of subtrees corresponding to the two parses of an `"np"` both of which span positions 3 to 11:

```
"vp" 2 (["np"],[])
 ((2,5), [Branch[SubNode("verb",(2,3)), SubNode("np",  (3,5))]])

 ((2,8), [Branch[SubNode("verb",(2,3)), SubNode("np",  (3,8))]])

 ((2,11),[Branch[SubNode("verb",(2,3)), SubNode("np",  (3,11))]])
```

2.5 Termination

The only source of iteration is in recursive function calls. Therefore, proof of termination is based on the identification of a measure function which maps the arguments of recursive calls to a well-founded ascending sequence of integers.

Basic recognizers such as `term 'i'` and the recognizer `empty` have no recursion and clearly terminate for finite input. Other recognizers that are defined in terms of these basic recognizers, through mutual and nested recursion, are applied by the `memoize` function which takes a recognizer and an index `j` as input and which accesses the `memotable`. An appropriate measure function maps the index and the set of left–rec values to an integer, which increases by at least one for each recursive call. The fact that the integer is bounded by conditions imposed on the maximum value of the index, the maximum values of the left-rec counters, and the maximum number of left-rec contexts, establishes termination. Extending recognizers to parsers does not involve any additional recursive calls and consequently, the proof also applies to parsers. A formal proof is available in the Appendix A.1.2.

2.6 Complexity

The following is an informal discussion. A formal proof is available in the Appendix A.1.3.

We begin by showing that memoized non-left-recursive and left-recursive recognizers have a worst-case time complexities of $O(n^3)$ and $O(n^4)$ respectively, where n is the number of tokens in the input. The proof proceeds as follows: `‘orelse‘` requires O(n) operations to merge the results from two alternate recognizers provided that the indices are kept in ascending order. `‘then‘` involves $O(n^2)$ operations when applying the second recognizer in a sequence to the results returned by the first recognizer. (The fact that recognizers can have multiple alternatives involving multiple recognizers in sequence increases cost by a factor that depends on the grammar, but not on the length of the input). For non-left-recursive recognizers, `memoize` guarantees that each recognizer is applied at most once to each input position. It follows that non-left recursive recognizers have $O(n^3)$ complexity. Recognizers with direct left recursion can be applied to the same input position at most `n` times. It follows that such recognizers have $O(n^4)$ complexity. In the worst case a recognizer with indirect left recursion could be applied to the same input position `n * nt` times where `nt` is the number of nonterminals in the grammar. This worst case would occur when every nonterminal was involved in the path of indirect recursion for some nonterminal. Complexity remains $O(n^4)$.

The only difference between parsers and recognizers is that parsers construct and store parts of parse trees rather than end points. We extend the complexity analysis of recognizers to that of parsers and show that for grammars in Chomsky Normal Form (CNF) (i.e. grammars whose right-hand-sides have at most two symbols, each of which can be either a terminal or a non-terminal), the complexity of non-left recursive parsers is $O(n^3)$ and of left-recursive parsers it is $O(n^4)$. The analysis begins by defining a “parse tuple” consisting of a parser name `p`, a start/end point pair `(s, e)`, and a list of parser names and end/point pairs corresponding to the first level of the parse tree returned by `p` for the sequence of tokens from `s` to `e`. (Note that this corresponds to an entry in the compact representation). The analysis then considers the effect of manipulating sets of parse tuples, rather than endpoints which are the values manipulated by recognizers. Parsers corresponding to grammars in CNF will return, in the worst case, for each start/end point pair (s, e) , (((e - s) + 1) *

t^2) parse tuples, where `t` is the number of terminals and non-terminals in the grammar. It follows that there are O(n) parse tuples for each parser and begin/endpoint pair. Each parse tuple corresponds to a bi-partition of the sequence starting at `s` and finishing at `e` by two parsers (possibly the same) from the set of parsers corresponding to terminals and non-terminals in the grammar. It is these parse tuples that are manipulated by `‘orelse‘` and `‘thenS‘`. The only effect on complexity of these operations is to increase the complexity of `‘orelse‘` from O(n) to O(n^2), which is the same as the complexity of `‘thenS‘`. Owing to the fact that the complexity of `‘thenS‘` had the highest degree in the application of a compound recognizer to an index, increasing the complexity of `‘orelse‘` to the same degree in parsing has no effect on the overall complexity of the process.

The representation of trees in the memotable has one entry for each parser. In the worst case, when the parser is applied to every index, the entry has `n` sub-entries, corresponding to `n` begin points. For each of these sub-entries there are up to `n` sub-sub-entries, each corresponding to an end point of the parse. Each of these sub-entries contains O(n) parse tuples as discussed above. It follows that the size of the compact representation is O(n^3).

2.7 Implementation

We have implemented our method in the pure functional programming language Haskell. We use a monad [18] to implement memoization. Use of a monad allows the memotable to be systematically threaded through the parsers while hiding the details of table update and reuse, allowing a clean and simple interface to be presented to the user. The complete Haskell code is available from any of the authors.

2.8 Experimental Results

In order to provide evidence of the low-order polynomial costs and scalability of our method, we conducted a limited evaluation with respect to four practical natural-language grammars used by Tomita ([14]) when comparing his algorithm with Earley's, and four variants of an abstract highly ambiguous grammar from Aho and Ullman [19]. Our Haskell program was

Input	No. of	Our algorithm (complete parsing)-PC				Tomitas (complete parsing)-DEC 20				Earleys (recognition only)-DEC 20			
length	Parses	G1	G2	G3	G4	G1	G2	G3	G4	G1	G2	G3	G4
Input from Tomitas sentence set 1. Timings are in seconds.													
19	346			0.02				4.79				7.66	
26	1,464			0.03				8.66				14.65	
Input from Tomitas sentence set 2. Timings are in seconds.													
22	429	0.00	0.00	0.03	0.05	2.80	6.40	4.74	19.93	2.04	7.87	7.25	42.75
31	16,796	0.00	0.02	0.05	0.09	6.14	14.40	10.40	45.28	4.01	14.09	12.06	70.74
40	742,900	0.03	0.08	0.11	0.14	11.70	28.15	18.97	90.85	6.75	22.42	19.12	104.91

Figure 2.1: Informal comparison with Tomita/Earley results

compiled using the Glasgow Haskell Compiler 6.6 (the code has not yet been tuned to obtain the best performance from this patform). We used a 3GHz/1GB PC in our experiments.

2.8.1 Tomita's Grammars

The Tomita grammars used were: G1 (8 rules), G2 (40 rules), G3 (220 rules), and G4 (400 rules). We used two sets of input: a) the three most-ambiguous inputs from Tomita's sentence set 1 of lengths `19, 26,` and `26` which we parsed with G3 (as did Tomita), and b) three inputs of lengths `4, 10,` and `40`, with systematically increasing ambiguity, chosen from Tomita's sentence set 2, which he generated automatically using the formula:

noun verb det noun (prep det noun)*

The results, which are tabulated in figure 2.1, show our timings and those recorded by Tomita for his original algorithm and for an improved Earley method, using a DEC-20 machine ([14]).

Considered by themselves our timings are low enough to suggest that our method is feasible for use in small to medium applications, such as NL interfaces to databases or rhythm analysis in poetry. Such applications typically have modest grammars (no more than a few hundred rules) and are not required to parse huge volumes of input.

Clearly there can be no direct comparison against years-old DEC-20 times, and improved versions of both of these algorithms do exist. However, we point to some relevant trends in the results. The increases in times for our method roughly mirror the increases shown for

Tomita's algorithm, as grammar complexity and/or input size increase. This suggests that our algorithm scales adequately well, and not dissimilarly to the earlier algorithms.

2.8.2 Highly ambiguous abstract grammars

We defined four parsers as executable specifications of four variants of a highly-ambiguous grammar introduced by Aho and Ullman [19] when discussing ambiguity: an unmemoized non-left–recursive parser `s`, a memoized version `ms`, a memoized left–recursive version `sml`, and a left–recursive version with all parts memoized. (This improves efficiency similarly to converting the grammar to Chomsky Normal Form.):

```
s    = (term 'a' `thenS` s `thenS` s)
       `orelse` empty

sm   = memoize "sm"
       ((term 'a' `thenS` sm `thenS` sm)
       `orelse` empty)

sml  = memoize "sml"
       ((sml `thenS` sml `thenS` term 'a')
       `orelse` empty)

smml = memoize "smml"
       ((smml `thenS`
         (memoize "smml_a"
           (smml `thenS` term 'a')))
       `orelse` empty)
```

We chose these four grammars as they are highly ambiguous. According to Aho and Ullman [19], `s` generates over `128` billion complete parses of an input consisting of 24 `'a'`'s. Although the left-recursive grammar does not generate exactly the same parses, it generates

the same number of parses, as it matches a terminal at the end of the rule rather than at the start.

Input length	No. of parses excluding partial parses	Seconds to generate the packed representation of full and partial parses			
		s	sm	sml	smml
6	132	1.22	0.00	0.00	0.00
12	208,012	out of space	0.00	0.00	0.02
24	1.29e+12		0.08	0.13	0.06
48	1.313e+26		0.83	0.97	0.80

Figure 2.2: Times to compute forest for n

These results show that our method can accommodate massively-ambiguous input involving the generation of large and complex parse forests. For example, the full forest for `n=48` contains 1,225 choice nodes and 19,600 branch nodes. Note also that the use of more memoization in `smml` reduces the cost of left-rec checking.

2.9 Concluding Comments

We have extended previous work of others on modular parsers constructed as executable specifications of grammars, in order to accommodate ambiguity and left recursion in polynomial time and space. We have implemented our method as a set of parser combinators in the functional programming language Haskell, and have conducted experiments which demonstrate the viability of the approach.

The results of the experiments suggest that our method is feasible for use in small to medium applications which need parsing of ambiguous grammars. Our method, like other

methods which use parser combinators or DCGs, allows parsers to be created as executable specifications which are "embedded" in the host programming language. It is often claimed that this embedded approach is more convenient than indirect methods which involve the use of separate compiler tools such as yacc, for reasons such as support from the host language (including type checking) and ease of use. The major advantage of our method is that it increases the type of grammars that can be accommodated in the embedded style, by supporting left recursion and ambiguity. This greatly increases what can be done in this approach to parser construction, and removes the need for non-expert users to painfully rewrite and debug their grammars to avoid left recursion. We believe such advantages balance well against any reduction in performance, especially when an application is being prototyped.

The Haskell implementation is in its initial stage. We are in the process of modifying it to improve efficiency, and to make better use of Haskell's lazy evaluation strategy (e.g. to return only the first `n` successful parses of the input).

Future work includes proof of correctness, analysis with respect to grammar size, testing with larger natural language grammars, and extending the approach so that language evaluators can be constructed as modular executable specifications of attribute grammars.

Bibliography

[1] Koskimies, K.: Lazy recursive descent parsing for modular language implementation. Software Practice and Experience **20 (8)** (1990) 453–462

[2] Nederhof, M.J., Koster, C.H.A.: Top-down parsing for left-recursive grammars. Technical Report, Research Institute for Declarative Systems, Department of Informatics, Katholieke Universiteit, Nijmegen. **93 - 10.** (1993)

[3] Hutton, G.: Higher-order functions for parsing. J. Functional Programming **2(3)** (1992) 323–343

[4] Frost, R.A.: Realization of natural-language interfaces using lazy functional programming. ACM Comput. Surv. **38(4)** (2006)

[5] Eijck, J.v.: Parser combinators for extraction. In: In Paul Dekker and Robert van Rooy, editors. (2003) 99 – 104

[6] Norvig, P.: Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics **17(1)** (1991) 91 – 98

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

[8] Moore, R.C.: Removing left recursion from context-free grammars. In: Proceedings, 1st Meeting of the North American Chapter of the ACL. (2000)

[9] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley Longman Publishing Co. (1986)

[10] Kuno, S.: The augmented predictive analyzer for context-free languages - its relative efficiency. Communications of the ACM (1966) 810 – 826

[11] Johnson, M.: Memoization in top-down parsing. Computational Linguistics **21(3)** (1995) 405–417

[12] Frost, R., Hafiz, R.: A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. SIGPLAN Notices **41(5)** (2006) 46–54

[13] Kay, M.: Algorithm schemata and data structures in syntactic processing. Technical Report CSL, XEROX Palo Alto Research (1980)

[14] Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1986)

[15] Shiel, B.A.: Observations on context-free parsing. Technical Report, Center for Research in Computing Technology, Aiken Computational Laboratory, Harvard University. (1976)

[16] Leermakers, R.: The Functional Treatment of Parsing. Kluwer Academic Publishers (1993)

[17] Lickman, P.: Parsing With Fixed Points. PhD thesis, University of Cambridge (1995)

[18] Wadler, P.: Monads for functional programming. 1st Spring School on Advanced FP **925** (1995) 24 – 52

[19] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall (1972)

Chapter 3

Parser Combinators for Ambiguous Left-Recursive Grammars

This paper was published as:

Frost, R., Hafiz, R. and Callaghan, P. (2008) *Parser Combinators for Ambiguous Left-Recursive Grammars.* Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN, Pages: 167-181. January 2008, San Francisco, USA.

3.1 Introduction

In functional programming, higher order functions called parser combinators can be used to build basic parsers and to construct complex parsers for nonterminals from other parsers. Parser combinators allow parsers to be defined in an embedded style, in code which is similar in structure to the rules of the grammar. As such, implementations can be thought of as executable specifications with all of the associated advantages. In addition, parser combinators use a top-down parsing strategy which facilitates modular piecewise construction and testing.

Parser combinators have been used extensively in the prototyping of compilers and processors for domain-specific languages such as natural language interfaces to databases, where complex and varied semantic actions are closely integrated with syntactic processing. However, simple implementations have exponential time complexity and inefficient representations of parse results for ambiguous inputs. Their inability to handle left-recursion is a long-standing problem. These shortcomings limit the use of parser combinators especially in applications with large and complex grammars.

Various techniques have been developed by others to address some of these shortcomings. However, none of that previous work has solved all of them.

The parser combinators that we present here are the first which can be used to create executable specifications of ambiguous grammars with unconstrained left-recursion, which execute in polynomial time, and which generate compact polynomial-sized representations of the potentially exponential number of results for highly ambiguous input.

The combinators are based on an algorithm developed by Frost, Hafiz and Callaghan [1]. That algorithm combines memoization with existing techniques for dealing with left recursion. The memotables are modified to represent the potentially exponential number of parse trees in a compact polynomial sized representation using a technique derived from [2] and [3]. A novel technique is used to accommodate indirect as well as direct left recursion.

This paper has three objectives: 1) To make the algorithm of Frost, Hafiz and Callaghan known to a wider audience beyond the Computational Linguistics community. In particular by the functional and logic programming communities both of which have a long history of

creating parsers as executable specifications (using parser combinators and Definite Clause Grammars respectively), 2) to introduce a library of parser combinators for immediate use by functional programmers, and 3) to illustrate how a declarative language facilitates the incremental implementation of a complex algorithm. Note that extension to include semantics will be straightforward, and that this work can be seen as an important step towards combinators that support general attribute grammars.

As example use of our combinators, consider the following ambiguous grammar from Tomita (1986) [3]. The nonterminal `s` stands for sentence, `np` for nounphrase, `vp` for verbphrase, `det` for determiner, `pp` for prepositional phrase, and `prep` for preposition. This grammar is left recursive in the rules for `s` and `np`.

```
s    ::= np vp | s pp      np   ::= noun  | det noun | np pp
pp   ::= prep np           vp   ::= verb np
det  ::= "a"   | "the"     noun ::= "i"  | "man" | "park" | "bat"
verb ::= "saw"             prep ::= "in"  | "with"
```

The Haskell code below defines a parser for the above grammar using our combinators `term`, `<+>`, and `*>`.

```
data Label = S | ... | PREP
s    = memoize S    $ np *> vp <+> s *> pp
np   = memoize NP   $ noun <+> det *> noun <+> np  *> pp
pp   = memoize PP   $ prep *> np
vp   = memoize VP   $ verb *> np
det  = memoize DET  $ term "a"  <+> term "the"
noun = memoize NOUN $ term "i"<+>term "man"<+>term "park" <+> term "bat"
verb = memoize VERB $ term "saw"
prep = memoize PREP $ term "in" <+> term "with"
```

The next page shows the "prettyprinted" output when the parser function `s` is applied to "`i saw a man in the park with a bat`". The compact representation corresponds to the several ways in which the whole input can be parsed as a sentence, and the many ways

in which subsequences of it can be parsed as nounphrases etc. For example, the entry for `NP` shows that nounphrases were identified starting at positions 1, 3, 6, and 9. Some of which were identified as spanning positions 3 to 5, 8, and 11. Two were found spanning positions 3 to 11. The first of which consists of a `NP` spanning 3 to 5 followed by a `PP` spanning 5 to 11. (We define a span from `x` to `y` as consuming terminals from `x` to `y - 1`.)

```
NOUN [1 ->[2 ->[Leaf "i"]]
     ,4 ->[5 ->[Leaf "man"]]
     ,7 ->[8 ->[Leaf "park"]]
     ,10->[11->[Leaf "bat"]]]
DET  [3 ->[4 ->[Leaf "a"]]
     ,6 ->[7 ->[Leaf "the"]]
     ,9 ->[10->[Leaf "a"]]]
NP   [1 ->[2 ->[SubNode NOUN (1,2)]]
     ,3 ->[5 ->[Branch [SubNode DET  (3,4) , SubNode NOUN (4,5)]]
          ,8 ->[Branch [SubNode NP   (3,5) , SubNode PP   (5,8)]]
          ,11->[Branch [SubNode NP   (3,5) , SubNode PP   (5,11)]
               ,Branch [SubNode NP   (3,8) , SubNode PP   (8,11)]]]
     ,6 ->[8 ->[Branch [SubNode DET  (6,7) , SubNode NOUN (7,8)]]
          ,11->[Branch [SubNode NP   (6,8) , SubNode PP   (8,11)]]]
     ,9 ->[11->[Branch [SubNode DET  (9,10), SubNode NOUN (10,11)]]]]
PREP [5 ->[6 ->[Leaf "in"]]
     ,8 ->[9 ->[Leaf "with"]]]
PP   [8 ->[11->[Branch [SubNode PREP (8,9), SubNode NP (9,11)]]]
     ,5 ->[8 ->[Branch [SubNode PREP (5,6), SubNode NP (6,8)]]
          ,11->[Branch [SubNode PREP (5,6), SubNode NP (6,11)]]]]
VERB [2 ->[3 ->[Leaf "saw"]]]
VP   [2 ->[5 ->[Branch [SubNode VERB (2,3), SubNode NP (3,5)]]
          ,8 ->[Branch [SubNode VERB (2,3), SubNode NP (3,8)]]
          ,11->[Branch [SubNode VERB (2,3), SubNode NP (3,11)]]]]
S    [1 ->[5 ->[Branch [SubNode NP   (1,2), SubNode VP (2,5)]]
          ,8 ->[Branch [SubNode NP   (1,2), SubNode VP (2,8)]
               ,Branch [SubNode S    (1,5), SubNode PP (5,8)]]
          ,11->[Branch [SubNode NP   (1,2), SubNode VP (2,11)]
               ,Branch [SubNode S    (1,5), SubNode PP (5,11)]
               ,Branch [SubNode S    (1,8), SubNode PP (8,11)]]]]
```

Parsers constructed with our combinators have $O(n^3)$ worst case time complexity for non-left-recursive ambiguous grammars (where n is the length of the input), and $O(n^4)$ for left recursive ambiguous grammars. This compares well with $O(n^3)$ limits on standard algorithms for CFGs such as Earley-style parsers (Earley 1970). The increase to n^4 is due

to expansion of the left recursive nonterminals in the grammar. Experimental evidence suggests that typical performance is closer to $O(n^3)$, possibly because few subparsers are left recursive and hence the $O(n^3)$ term predominates. Experimental evaluation involved four natural-language grammars from (Tomita 1986), four variants of an abstract highly-ambiguous grammar, and a medium-size natural-language grammar with 5,226 rules. The potentially-exponential number of parse trees for highly-ambiguous input are represented in polynomial space as in Tomita's algorithm.

We begin with background material followed by a detailed description of the Haskell implementation. Experimental results, related work, and conclusions are given in sections 3.3.4, 3.3.5 and 3.3.6. Formal proofs of termination and complexity are available in the Appendix 1.1 and 1.3., and the code of the initial Haskell implementation, are available at: `http://cs.uwindsor.ca/~hafiz/SyntaxParser/Demonstration.hs`

3.2 Background

3.2.1 Top down parsing and memoization

Top-down parsers search for parses using a top-down expansion of the grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. Simple implementations do not terminate for left-recursive grammars, and have exponential time complexity with respect to the length of the input for non-left-recursive ambiguous grammars.

The problem of exponential time complexity in top-down parsers constructed as sets of mutually-recursive functions has been solved by Norvig [4]. His technique is similar to the use of dynamic programming and state sets in Earley's algorithm [5], and tables in the CYK algorithm of Cocke, Younger and Kasami. The key idea is to store results of applying a parser `p` at position j in a memotable and to reuse results whenever the same situation arises. It can be implemented as a wrapper function `memoize` which can be applied selectively to component parsers.

3.2.2 The need for left recursion

Left-recursion can be avoided by transforming the grammar to a weakly equivalent non-left-recursive form (i.e. to a grammar which derives the same set of sentences, but does not generate the same set of parse trees). Such transformation has two disadvantages: 1) it is error prone, especially for non-trivial grammars, and 2) the loss of some parses (as illustrated in the example in [1]) complicates the integration of semantic actions, especially in NLP.

3.2.3 An introduction to parser combinators

The details in this description have been adapted to our approach, and are limited to recognition. We extend the technique to parsers later. Assume that the input is a sequence of tokens *input*, of length *#input* the members of which are accessed through an index j. Recognizers are functions which take an index j as argument and which return a set of indices. Each index in the result set corresponds to a position at which the parser successfully finished recognizing a sequence of tokens that began at position j. An empty result set indicates that the recognizer failed to recognize any sequence beginning at j. The result for an ambiguous input is a set with more than one element. This use of *indices* instead of the more conventional subsequence of input is a key detail of the approach: we need the positions to index into the memotables.

A recognizer `term 'x'` for a terminal `'x'` is a function which takes an index j as input, and if j is less than *#input* and if the token at position j in the input corresponds to the terminal `'x'`, it returns a singleton set containing $j + 1$, otherwise it returns the empty set. The `empty` recognizer is a function which always succeeds returning a singleton set containing the current position. A recognizer for alternation `p|q` is built by combining recognizers for `p` and `q`, using the combinator `<+>`. When the composite recognizer is applied to index j, it applies `p` to j, applies `q` to j, and subsequently unites the resulting sets.

A composite recognizer corresponding to a sequence of recognizers `p q` on the right hand side of a grammar rule, is built by combining those recognizers using the parser combinator

`*>`. When the composite recognizer is applied to an index j, it first applies `p` to j, then it applies `q` to each index in the set of results returned by `p`. It returns the union of these applications of `q`. The combinators `term`, `empty`, `<+>` and `*>` are defined (in functional pseudo code) as follows:

$$term \quad t \quad j = \begin{cases} \{\} & , j \geq \#input \\ \{j+1\} & , j^{th} \text{ element of } input = t \\ \{\} & , \text{otherwise} \end{cases}$$

$$empty \quad j = \{j\}$$

$$(p \quad \texttt{<+>} \quad q) \quad j = (p \quad j) \quad \cup \quad (q \quad j)$$

$$(p \quad \texttt{*>} \quad q) \quad j = \bigcup (map \quad q \quad (p \quad j))$$

The combinators can be used to define composite mutually-recursive recognizers. For example, the grammar `s ::= 'x' s s | empty` can be encoded as `s = (term 'x' *> s *> s) <+> empty`. Assuming the input is "`xxxx`", then:

```
(empty <+> term 'x')   2 => {2,3}
(term 'x' *> term 'x') 1 => {3}

s 0  => {4, 3, 2, 1, 0}
```

The last four values in the result for `s 0` correspond to proper prefixes of the input being recognized as an `s`. The result `4` corresponds to the case where the whole input is recognized as an `s`. Note that we have used sets in this explanation to simplify later development of the combinators.

3.3 The Combinators

3.3.1 Preliminaries

The actual implementation of the combinators `*>` and `<+>` for plain recognizers in Haskell is straightforward, and makes use of a library for Sets of `Ints`. An excerpt is given below. In this fragment and the ones that follow, we make some simplifications to ease presentation of the key details. Full working code is available from the URL given in Section 3.1. We omit details of how we access the input throughout this paper, treating it as a constant value.

```
type Pos    = Int
type PosSet = IntSet
type R      = Pos -> PosSet
(<+>)       :: R -> R -> R
p <+> q     = \r -> union (p r) (q r)
(*>)        :: R -> R -> R
p *> q      = \r -> unions $ map q $ elems $ p r
parse       :: R -> PosSet
parse p     = p 0
```

In the following we develop the combinators `*>` and `<+>` incrementally, by adding new features one at a time in each subsection. We begin with memoization, then direct left recursion, then indirect left recursion, then parsing (to trees). The revised definitions accompany new types which indicate a particular version of the combinator. The modifications to `<+>` are omitted when they are reasonably straightforward or trivial.

3.3.2 Memoizing recognizers

We modify the combinators so that a memotable is used during recognition. At first the table is empty. During the process it is updated with an entry for each recognizer that is applied to a position. Recognizers to be memoized are labelled with values of a type chosen by the programmer. These labels usually appear as node labels in resulting parse trees, but

more generality is possible, e.g. to hold limited semantic information. We require only that these labels be enumerable, i.e. have a unique mapping to and from `Ints`, a property that we use to make table lookups more efficient by converting label occurrences internally to `Ints` and using the optimized `IntMap` library.

The memotable is a map of memo label and start position to a result set. The combinators are lifted to the monad level and the memotable is the state that is threaded through the parser operations, and consulted and/or updated during the `memoize` operation. We use a standard state monad:

```
type ILabel  = Int
type RM memoLabel = Pos -> StateM (State memoLabel) PosSet
data StateM s t = State {unState:: s -> (t, s)}
type State nodeName = IntMap (IntMap PosSet)
(*>) :: RM l -> RM l -> RM l
p *> q = \r -> do end_p  <- p r
                  end_qs <- mapM q (elems end_p)
                  return $ unions end_qs
```

The `memoize` function makes the decision regarding reuse of results. It is implemented as a "wrapper" around other parsers, hence any sub-parser can be memoized. The function checks whether an entry exists in the memotable for the given parser label and position, returning the stored result if yes, otherwise it runs the parser and stores the results before returning them. `update_table` adds the new information to the table. Note that the update effect is to "overwrite" the previous information. The `insertWith`...`insert` combination merges into the outer table (`insertWith`) a new inner table that discards (`insert`) any previous entry for that label and start position. This is necessary to update the stored information as (left) recursion unwinds (see section 3.3.3):

```
memoize :: Enum l => l -> RM l -> RM l
memoize e_name parser pos
 = do mt <- get
      case lookupT i_name pos mt of
        Just res -> return res
        Nothing  -> do res <- parser pos
```

```
                    modify (update_table res)
                    return res
where
  i_name = fromEnum e_name
  update_table :: PosSet -> State l -> State l
  update_table res = insertWith (\_ prev -> insert pos res prev)
                                          i_name (singleton pos res)
```

3.3.3 Accommodating direct left recursion

To accommodate direct left recursion, we use "left-rec counts" c_{ij} denoting the number of times a recognizer r_i has been applied to an index j. For non-left-recursive recognizers c_{ij} will be at most one. For left-recursive recognizers, c_{ij} is increased on recursive descent. Application of a recognizer r_i to an index j is curtailed whenever c_{ij} exceeds the number of unconsumed tokens of the input plus 1. At this point no parse is possible (other than spurious parses from cyclic grammars — which we want to curtail anyway.) As an illustration, consider the following portion of the search space being created during the parse of two remaining tokens on the input (where `N`, `P` and `Q` are nodes in the parse search space corresponding to nonterminals. `A`, `B` and `C` are nodes corresponding to terminals or nonterminals):

```
          N
         /   \
        N     A
       /  \
      N    B
     /  \
    P    C
   /
  Q
 /
N
```

The last call of the parser for `N` should be curtailed owing to the fact that, irrespective

of what `A`, `B`, and `C` are, either they must require at least one input token, or else they must rewrite to `empty`. If they all require a token, then the parse cannot succeed. If any rewrite to `empty`, then the grammar is cyclic (`N` is being rewritten to `N`). The last call should be curtailed in either case.

Curtailing a parse when a branch is longer than the length of the remaining input is incorrect as this can occur in a correct parse if recognizers are rewritten into other recognizers which do not have "token requirements to the right". Also, we curtail the recognizer when the left-rec count exceeds the number of unconsumed tokens *plus 1*. The plus 1 is necessary for the case where the recognizer rewrites to empty on application to the end of the input.

This curtailment test is implemented by passing a "left-rec context" down the invocation tree. The context is a frequency table of calls to the memoized parsers encountered on the current chain.

```
type L_Context = [(ILabel, Int)]
type LRM memolabel = L_Context -> RM memolabel
```

Only `*>` and `memoize` need to be altered beyond propagating the information downwards. `memoize` checks before expanding a parser `p` to see if it has been called more than there are tokens left in the input, and if so, returns an empty result, otherwise continues as before though passing a context updated with an extra call to `p`. The alteration to `*>` controls what context should be passed to `q`: the current context should only be passed when `p` has consumed no tokens, i.e. has done nothing to break the left-recursive chain.

```
(*>) :: LRM l -> LRM l -> LRM l
p *> q = \ctxt r -> do end_p <- p ctxt r
                       let pass_ctxt e | e == r    = ctxt
                                       | otherwise = []
                       end_qs<-mapM (\e-> q (pass_ctxt e) e)(elems end_p)
                       return $ unions end_qs


memoize :: Enum l => l -> LRM l -> LRM l
```

```
memoize e_name p ctxt pos
 = do mt <- get
      case lookupT i_name pos mt of
           Just res -> return res
           Nothing  | depth_cutoff i_name ctxt >
                                (length_input - pos + 1) -> empty
                    | otherwise -> do
                                       .. p (increment i_name ctxt) pos ..
   where i_name = fromEnum e_name
         depth_cutoff i e = case lookup i e of Nothing -> 0
                                               Just fe -> fe
```

Notice what happens when unwinding the left-recursive calls. At each level, `memoize` runs the parser and adds the results to the table for the given label and start position. This table update, as mentioned earlier, overwrites previous information at the start position, and therefore the table always contains the "best results so far". Note that the algorithm accommodates cyclic grammars. It terminates for such grammars with information being stored in the memotable which can be subsequently used to identify cycles.

3.3.4 Accommodating indirect left recursion

We begin by illustrating how the method above may return incomplete results for grammars containing indirect left recursion. Consider the following grammar, and subset of the search space, where the left and right branches represent the expansions of the first two alternate right-hand-sides of the rule for the nonterminal `S`, applied to the same position on the input:

```
S ::= S ..| Q | P | x                    S
P ::= S ..                         /     +    \
Q ::= T                          S ..          Q
T ::= P                          |             |
                                 S ..          T
                                 |             |
                                 P             P
```

```
|          |
S ..       S ..
|
curtail S
```

Suppose that the branch for the left alternative is expanded before the right branch during the search, and that the left branch is curtailed due to the left-rec count for `S` exceeding its limit. The results stored for `P` on recursive ascent of the left branch is an empty set. The problem is that the later call of `P` on the right branch should not reuse the empty set of results from the first call of `P` as they are incomplete with respect to the position of `P` on the right branch (i.e. if `P` were to be reapplied to the input in the context of the right branch, the results would not necessarily be an empty set.) This problem is a result of the fact that, on the left branch, `S` caused curtailment of the results for `P` as well as for itself.

Our solution to this problem is as follows: 1) Pass left-rec contexts downwards as in subsection 3.3.3. 2) Generate the reasons for curtailment when computing results. For each result we need to know if the subtrees contributing to it have been curtailed through any left-rec limit, and if so, which recognizers caused the curtailment. 3) Store results in the memotable together with a subset of the current left-rec context corresponding to those recognizers that caused the curtailment at the current position, and 4) Whenever a stored result is being considered for reuse, the left-rec-context of that result is compared with the left-rec-context of the current node in the parse space. The result is only reused if, for each recognizer in the left-rec context of the result, the left-rec-count is smaller than or equal to the left-rec-count in the current context. This ensures that a result stored for application `P` of a recognizer at index j is only reused by a subsequent application `P'` of the same recognizer at the same position, if the left-rec context for `P'` would constrain the result more, or equally as much, as it had been constrained by the left-rec context for `P` at j. If there were no curtailment, the left-rec context of a result would be empty and that result can be reused anywhere irrespective of the current left-rec context.

This strategy extends the recognizer return type to include a set of labels that caused curtailments during that parse. Note that we only collect information about curtailment for the current position, so only collect results from `q` in the case where `p` consumed no input,

i.e. where the endpoint of `p` is the same as the starting position.

```
type CurtailingNTs  = IntSet
type UpResult       = (CurtailingNTs, PosSet)
type State nodeName = IntMap (IntMap (L_Context,UpResult))
type CLRM memoLabel = L_Context -> Pos
                                -> StateM (State memoLabel) UpResult

(*>) :: CLRM l -> CLRM l -> CLRM l
p *> q = \ctxt r -> do (cut,end_p) <- p ctxt r
                       let pass_ctxt e | e == r    = ctxt
                                       | otherwise = []
                           merge_cuts e prev new
                                        | e == r    = union prev new
                                        | otherwise = prev
                           join (prev_cut, prev_result) e
                            = do (new_cut, result) <- q (pass_ctxt e) e
                                 return ( merge_cuts e prev_cut new_cut
                                        , union prev_result result )
                       end_qs <- foldM join (cut, empty) end_p
                       return end_qs
```

The function `<+>` is modified to merge information from the subparsers:

```
(<+>) :: CLRM l -> CLRM l -> CLRM l
(p <+> q) inp cc  = do (cut1,m) <- p inp cc
                       (cut2,n) <- q inp cc
                       return ( union cut1 cut2 , union m n )
```

When retrieving results, `memoize` compares the current context with the pruned stored context. Reuse is only allowed if every label in the stored context appears in the current context and is not less constrained in the current context. Otherwise, the parser is run further in the current context to compute the results that were curtailed (and hence missing) in the earlier call.

```
pruneContext :: CurtailingNTs -> L_Context -> L_Context
```

```
pruneContext rs ctxt = [nc | nc@(n,c) <- ctxt, n `member` rs]
canReuse :: L_Context -> L_Context -> Bool
canReuse current stored
 = and [ or [ cc >= sc | (cn,cc) <- current, sn == cn ]
       | (sn,sc) <- stored ]
```

3.3.5 Building parse trees

Turning a recogniser into a parser is straightforward. A set of endpoints now becomes a map of endpoints to lists of trees that end at that point. The memotable type is altered to contain this new information: it stores tree results with their curtail set and a relevant `L_context`. The tree type is shown below.

```
data Tree l = Empty | Leaf Token | Branch [Tree l]  | ...
type ParseResult memoLabel = [(Int, [Tree memoLabel])]
type UpResult memoLabel = (CurtailingNTs, ParseResult memoLabel)
data Stored memoLabel = Stored { s_stored  :: UpResult memoLabel
                               , s_context :: L_Context
                               , s_results :: [(Int, Tree memoLabel)]}
type State memoLabel = IntMap (IntMap (Stored memoLabel))
type P memoLabel
    = L_Context -> Pos -> StateM (State memoLabel) (UpResult memoLabel)
```

`term` parsers now return a list of leaf values with suitable endpoints. The empty parser returns an empty tree. Alternative parses from `<+>` are merged by appending together the lists of trees ending at the same point. The maps are held in ascending endpoint order to give this operation an $O(n)$ cost.

Sequences require `*>` to join all results of `p` with all results of `q`, forming new branch nodes in the tree, and merging the resulting maps together. The former is achieved with `addP` which combines a particular result from `p` with all subsequent results from `q`, and with `addToBranch` which merges lists of trees from both the left and the right into a new list of trees. Notice that this operation is a cross-product: it must pair each tree from the left with each tree on the right. Tree merging or packing is done by concatenating results at the same endpoint.

```
addP :: [[Tree l]] -> ParseResult l -> ParseResult l
```

Test Set	#Input	#Parses	Our method				Tomita's method			
			G1	G2	G3	G4	G1	G2	G3	G4
Tomita's sent. set 1	19	346	0.02				4.79			
	26	1,464	0.03				8.66			
Tomita's sent. set 2	22	429	0.02	0.02	0.03	0.03	2.80	6.40	4.74	19.93
	31	16,796	0.02	0.02	0.05	0.08	6.14	14.40	10.40	45.28
	40	742,900	0.02	0.06	0.08	0.09	11.70	28.15	18.97	90.85

Figure 3.1: Informal comparison with Tomita's results (timings in seconds)

```
addP left_result right_output
 = [ (re , addToBranch left_result right_results)
                       | (re , right_results) <- right_output ]
addToBranch :: [[Tree l]] -> [[Tree l]] -> [[Tree l]]
addToBranch lts rts = [r ++ l | l <- lts, r <- rts]
```

The `memoize` function handles the rest of tree formation, both labelling and introducing sharing to avoid an exponential number of trees. Labelling attaches the memo label to the tree result. Sharing replaces the computed list of results with a single result that contains sufficient information to find the original list which will be stored in the memo table. This single result is then returned to higher parsers as a 'proxy' for the original list. To avoid recomputation, we also store the proxy in the memotable to be retrieved by subsequent parser lookups. This technique avoids exponential blow-up of the number of results propagated by parsers. An example of the resulting compact representation of parse trees has been given in Section 3.1.

It is important to note that the combinators support addition of semantics. The extension from trees to semantic values is straightforward via an "applicative functor" interface, e.g. with operator `(<*>) :: P (a -> b) -> P a -> P b`. A monadic interface may also be defined.

3.4 Experimental Results

To provide evidence of low-order polynomial costs, we conducted a small scale evaluation with respect to: a) Four practical natural-language grammars from Tomita ([3], Appendix F, pages 171 to 184); b) Four variants of an abstract highly ambiguous grammar from Aho and Ullman [6]; and c) A medium size NL grammar for an Air Travel Information System maintained by Carroll [7].

Our Haskell program was compiled using the Glasgow Haskell Compiler 6.6. We used a 3GHz/1Gb PC. The performance reported is the "MUT time" as generated in GHC runtime statistics, which is an indication of the time spent doing useful computation. It excludes time spent in garbage collection. We also run with an initial heap of 100Mb and do not fix an upper limit to heap size (apart from the machine's capacity).

Note that the grammars we have tested are inherently expensive owing to the dense ambiguity, and this is irrespective of which parsing method is used.

3.4.1 Tomita' Grammars

The grammars used were: G1 (8 rules), G2 (40 rules), G3 (220 rules), and G4 (400 rules) (Tomita 1986). We used two sets of input: a) the two most-ambiguous inputs from Tomita's sentence set 1 (page 185 App. G) of lengths 19 and 26 which we parsed with G3 (as did Tomita), and b) three inputs of lengths 4, 10, and 40, with systematically increasing ambiguity, from Tomita's sentence set 2.

Figure 1 shows our times and those recorded by Tomita for his algorithm, using a DEC-20 machine (Tomita 1986, pages 152 and 153 App. D). Clearly there can be no direct comparison against years-old DEC-20 times. However, we note that Tomita's algorithm was regarded, in 1986, as being at least as efficient as Earley's and viable for natural-language parsing using machines that were available at that time. The fact that our algorithm is significantly faster on current PCs supports the claim of viability for NL parsing.

Input Length	No. of parses	s	sm	sml	smml
6	132	1.22	-	-	-
12	208,012	*	-	-	0.02
24	1.289e+12		0.08	0.13	0.06
48	1.313e+26		0.83	0.97	0.80

Figure 3.2: Timings for highly-ambiguous grammars (time in seconds).

3.4.2 Highly ambiguous abstract grammars

We defined parsers for four variants of a highly-ambiguous grammar introduced by Aho and Ullman [6]: an unmemoized non-left–recursive parser `s`, a memoized version `ms`, a memoized left–recursive version `sml`, and a memoized left–recursive version with one sub-component also memoized `smml`:

```
s    =                term 'x' *> s  *> s     <+> empty
sm   = memoize SM   $ term 'x' *> sm *> sm    <+> empty
sml  = memoize SML  $ sml  *> sml *> term 'x' <+> empty
smml = memoize SMML $ smml *> (memoize SMML' $ smml *> term 'x') <+>empty
```

We chose these four grammars as they are highly ambiguous. The results in figure 2 show that our algorithm can accommodate massively ambiguous input involving the generation of large and complex parse forests. '*' denotes memory overflow and '-' denotes timings less than 0.005 seconds.

3.4.3 ATIS – A medium size NL grammar

Here, we used a modified version of the ATIS grammar and test inputs generated by Carroll [7], who extracted them from the DARPA ATIS3 treebank.

Our modifications include adding 634 new rules and 66 new nonterminals in order to encode the ATIS lexicon as CFG rules. The resulting grammar consists of 5,226 rules with 258 nonterminals and 991 terminals. Carroll's test input set contains 98 natural language

sentences of average length 11.4 words. An example sentence is *"i would like to leave on thursday morning may fifth before six a.m."*.

Times to parse ranged from <1 second for the 5 shortest inputs, to between 12 and 19 seconds for the 5 longest inputs. The average time was 1.88 seconds. Given that our Haskell implementation is in an early stage of development, these results suggest that it may be possible to use our algorithm in applications involving large grammars.

3.5 Related Work

Our combinators implement the algorithm of Frost, Hafiz and Callaghan [1]. The relationship of that algorithm to work by others on left recursion is discussed in detail in their paper. The following is a brief summary: As in [8], the algorithm passes information to parsers which is used in curtailment. The information passed is similar to the cancellation sets used in [9]. The algorithm uses the memoization technique of Norvig [4] to achieve polynomial complexity with parser combinators, as do Frost [10], Johnson [11], and Frost and Hafiz [12]. Note that Ford [13] has also used memoization in functional parsing, but for constrained grammars. Lickman [14] accommodates left-recursion using fixed points, based on an unpublished idea by [15], but he does not address the problem of exponential complexity. Johnson [11] integrates a technique for dealing with left recursion with memoization. However, the algorithm on which we base our combinators differs from Johnson's $O(n^3)$ approach in the technique that we use to accommodate left recursion. Also, the algorithm facilitates the construction of compact representations of parse results whereas Johnson's appears to be very difficult to extend to do this. As in [12] the algorithm integrates "left-recursion counts" with memoization, and defines recognizers as functions which take an index as argument and which return a set of indices. The algorithm is an improvement in that it can accommodate indirect as well as direct left recursion and can be used to create parsers in addition to recognizers.

Extensive research has been carried out on parser combinators. A comprehensive overview of that work can be found in [16]. Our approach owes much to that work. In particular, our combinators and motivation for their use follows from Burge [17] and Fair-

burn [18]. Also, we use Wadler's [19] notion of failure as an empty list of successes, and many of the ideas from Hutton and Meijer [20] on monadic parsing.

3.6 Concluding Comments

We have developed a set of parser combinators which allow modular and efficient parsers to be constructed as executable specifications of ambiguous left-recursive grammars. The accommodation of left recursion greatly increases what can be done in this approach, and removes the need for non-expert users to painfully rewrite and debug their grammars to avoid left recursion. We believe that such advantages balance well against any reduction in performance, especially when an application is being prototyped, and in those applications where the additional time required for parsing is not a major factor in the overall time required when semantic processing, especially of ambiguous input, is taken into account. Experimental results indicate that the combinators are feasible for use in small to medium applications with moderately-sized grammars and inputs. The results also suggest that with further tuning, they may be used with large grammars.

Future work includes proof of correctness, analysis w.r.t. grammar size, improvements for very large grammars, detailed comparison with other combinators systems such as Parsec, reduction of reliance on monads in order to support some form of "on-line" computation, comparison with functional implementations of GLR parsers, and extension of the approach to build modular executable specifications of attribute grammars.

Bibliography

[1] Frost, R., Hafiz, R., Callaghan, P.: Modular and efficient top-down parsing for ambiguous left-recursive grammars. ACL-IWPT (2007) 109 – 120

[2] Kay, M.: Algorithm schemata and data structures in syntactic processing. Technical Report CSL, XEROX Palo Alto Research (1980)

[3] Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1986)

[4] Norvig, P.: Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics **17(1)** (1991) 91 – 98

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

[6] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall (1972)

[7] Carroll, J.: Efficiency in large-scale parsing systems -parser comparison. (2003)

[8] Shiel, B.A.: Observations on context-free parsing. Technical Report, Center for Research in Computing Technology, Aiken Computational Laboratory, Harvard University. (1976)

[9] Nederhof, M.J., Koster, C.H.A.: Top-down parsing for left-recursive grammars. Technical Report, Research Institute for Declarative Systems, Department of Informatics, Katholieke Universiteit, Nijmegen. **93 - 10.** (1993)

[10] Frost, R.A.: Using memoization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers. SIGPLAN Notices **29 (4)** (1994) 23 – 30

[11] Johnson, M.: Memoization in top-down parsing. Computational Linguistics **21(3)** (1995) 405–417

[12] Frost, R., Hafiz, R.: A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. SIGPLAN Notices **41(5)** (2006) 46–54

[13] Ford, B.: Packrat parsing: simple, powerful, lazy, linear time. In: ICFP. (2002)

[14] Lickman, P.: Parsing With Fixed Points. PhD thesis, University of Cambridge (1995)

[15] Wadler, P.: Monads for functional programming. 1st Spring School on Advanced FP **925** (1995) 24 – 52

[16] Frost, R.A.: Realization of natural-language interfaces using lazy functional programming. ACM Comput. Surv. **38(4)** (2006)

[17] Burge, W.H.: Recursive Programming Techniques. Addison-Wesley. (1975)

[18] Fairburn, J.: Making form follow function: An exercise in functional programming style. Technical Report , Cambridge Comp. Lab. **89** (1986)

[19] Wadler, P.: How to replace failure by a list of successes. In: FP languages and computer architecture. Volume 201. (1985) 113 – 128

[20] Hutton, G., Meijer, E.: Monadic parser combinators. J. Funct. Program. **8(4)** (1998) 437 – 444

Chapter 4

Lazy Combinators for Executable Specifications of General Attribute Grammars

This paper was published as:

Hafiz, R. and Frost, R. (2010) *Lazy Combinators for Executable Specifications of General Attribute Grammars.* Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN, Pages : 167-182. January 2010, San Francisco, USA.

4.1 Introduction

Attribute grammar (AG, [1]) systems have been constructed primarily as compilable parser-generators for formal languages. Little work has been done where fully-general AGs have been used to offer a platform for declaratively specifying directly-executable specifications of natural languages (NL) to construct NL interfaces or NL database query processors. Although it is highly modular, general top-down parsing is often ignored as it has been traditionally categorized as expensive, and non-terminating while processing left-recursive grammars. Also, no existing approach supports arbitrary attribute dependencies (including dependencies from the right) in one-pass within a modular top-down system.

A platform that supports executable and declarative specifications of general AGs, offers two benefits. From a practical viewpoint, application developers can specify and execute their language descriptions directly without worrying about underlying evaluation methods. Individual parts of descriptions can be efficiently tested piecewise, and modularity enables systematic and incremental development. From a theoretical perspective, general AGs accommodate ambiguity and left-recursion, which are needed for natural language processing. As illustrated by Warren [2] and Frost et al. [3], transforming a left-recursive CFG to a weakly equivalent non-left-recursive form may introduce loss of parses and lack of completeness in semantic interpretation. AGs with arbitrary attribute dependencies provide unrestricted construction of declarative semantic rules, facilitating expression of complex linguistic theories such as Montague semantics.

Frost et al. explained at PADL'08 [4] how top-down parsers can be constructed as unconstrained executable CFGs. This paper describes an extension to accommodate semantics with arbitrary attribute dependencies. We have achieved our objective by defining a set of combinators for constructing modular, declarative and executable language processors, similar to the denotational semantics textbook AG notation. Our combinators (e.g., `<|>` and `*>` correspond to alternating and sequencing, `rule_s` and `rule_i` for synthesized and inherited semantic rules, `parser` and `nt` for AG formation) are pure, higher-order and lazy functions that ensure fully declarative specifications (Section 4.3.2 and 4.3.3).

We define attributes in terms of *expressions* (as our method is referentially transparent and non-strict) that represent operations on syntax symbols, and these expressions are

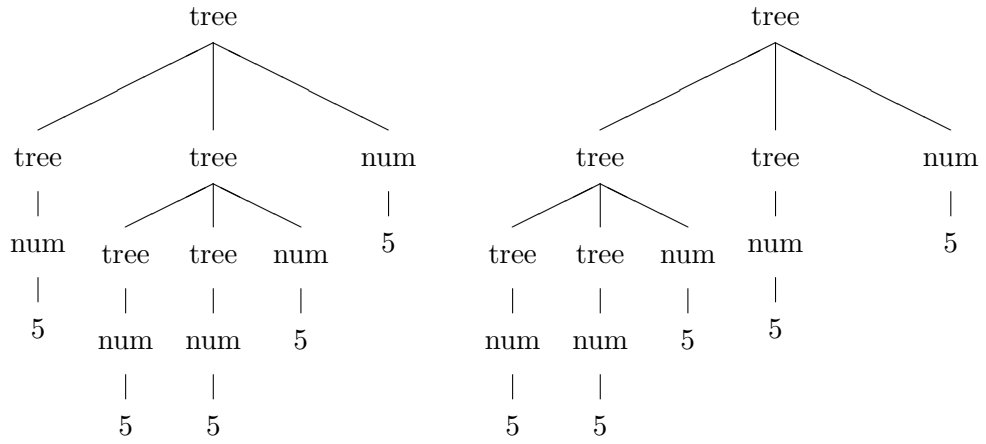
computed from the surrounding *environment* when required (Section 4.3.4). We execute syntax and semantics in polynomial time using memoization to ensure that results for a particular parser at a particular input position are computed at most once, and are reused when required. We represent potentially exponential results for ambiguous input in a compact and shared polynomial tree structure (Section 4.4).

We have provided a platform by implementing our algorithm in terms of higher-order functions in Haskell. The declarative notation, arbitrary dependencies and non-strict evaluation have the potential to allow us to discard unwanted parses using linguistic features such as grammatical, semantic and number agreements, and this could extend the AG paradigm by capturing characteristics of unification grammars, combinatory-categorical grammars and type-theoretic grammars while being computationally efficient.

An Example: We illustrate our approach with a simple artificial `repmax` example [5, 6] which we have extended to accommodate ambiguity, left-recursion and arbitrary attribute dependencies in semantic rules. Our goal is to parse inputs such as "`1 5 2 3 2`" with the ambiguous left-recursive CFG $tree ::= tree\ tree\ num \mid num$, $num ::= 1|2|3|4|5|...$, and to extract all possible trees with all terminals replaced by the maximum value of the sequence using sets of declarative semantics. The following example illustrates most of the aspects of general AGs :

$$
\begin{array}{rcl}
start(S_0) & ::= & tree(T_0) \\
\{RepVal.T_0 \downarrow & = & MaxVal.T_0 \uparrow\} \\
tree(T_0) & ::= & tree(T_1)\ \ tree(T_2)\ \ num(N_1) \\
\{MaxVal.T_0\uparrow & = & Max(MaxVal.T_1 \uparrow, MaxVal.T_2 \uparrow, MaxVal.N_1 \uparrow), \\
RepVal.T_1 \downarrow & = & RepVal.T_0 \downarrow, RepVal.T_2 \downarrow = RepVal.T_0 \downarrow, \\
RepVal.N_1 \downarrow & = & RepVal.T_0 \downarrow\} \\
 & \mid & num(N_2) \\
\{MaxVal.T_0\uparrow & = & MaxVal.N_2 \uparrow, RepVal.N_2 \downarrow = RepVal.T_0 \downarrow\} \\
num(N_0) & ::= & 1\ \ \{MaxVal.N_0 \uparrow = 1\} \mid\ ...\ \mid 5\{MaxVal.N_0 \uparrow = 5\}
\end{array}
$$

According to this AG, there are two ambiguous outputs when *start* is applied to the input sequence "`1 5 2 3 2`":



Two parse trees for one input

Using our method, an "almost verbatim" executable specification of the above AG's representation can be constructed in Haskell as follows:

```
start = memoize Start parser (nt tree T0)
 [rule_i  RepVal Of T0 Is findRep [synthesized  MaxVal Of T0]]

tree  = memoize Tree parser
 (nt tree T1 *> nt tree T2 *> nt num T3)
 [rule_s MaxVal Of LHS Is
         findMax [synthesized MaxVal Of T1
                 ,synthesized MaxVal Of T2
                 ,synthesized MaxVal Of T3]

 ,rule_i RepVal Of T1 Is findRep [inherited RepVal Of LHS]
 ........
 <|> parser (nt num N1)
 [rule_i RepVal Of N1  Is findRep [inherited RepVal Of LHS]
 ,rule_s MaxVal Of LHS Is findMax [synthesized MaxVal Of N1]]

num  = memoize Num terminal term "1" [MaxVal 1] <|> ...
               <|> terminal term "5" [MaxVal 5]
```

When the executable specification of `start` is applied to "`1 5 2 3 2`", a compact representation of ambiguous parse trees is generated with appropriate semantic values for re-

spective grammar symbols. For example, `tree` parses the whole input (starting at position 1 and ending at position 6) in two ambiguous ways. The `tree`'s inherited and synthesized attributes (represented with `I` and `S`) are associated with its start and end positions respectively. The attributes are of the form `attribute_type value` e.g. `RepVal 5`. The compact results have pointing sub-nodes (as node-name, unique-id pairs e.g. `(Tree,T1)`) with inherited and synthesized attributes:

```
Tree START at 1 ; Inherited atts:   T0 RepVal 5
     END at   6 ; Synthesized atts: T0 MaxVal 5
Branch
[SubNode (Tree,T1) ((1,[((I,T1),[RepVal 5])]),(4,[((S,T1),[MaxVal 5])]))
,SubNode (Tree,T2) ((4,[((I,T2),[RepVal 5])]),(5,[((S,T2),[MaxVal 3])]))
,SubNode (Num, T3) ((5,[((I,T3),[RepVal 5])]),(6,[((S,T3),[MaxVal 2])]))]
     END at   6 ; Synthesized atts: T0 MaxVal 5
Branch
[SubNode (Tree,T1) ((1,[((I,T1),[RepVal 5])]),(2,[((S,T1),[MaxVal 1])]))
,SubNode (Tree,T2) ((2,[((I,T2),[RepVal 5])]),(5,[((S,T2),[MaxVal 5])]))
,SubNode (Num, T3) ((5,[((I,T3),[RepVal 5])]),(6,[((S,T3),[MaxVal 2])]))]
                            ..................
Num  START at 1 ; Inherited atts:   N1 RepVal 5
     END at   2 ; Synthesized atts: N1 MaxVal 1
    Leaf (ALeaf "1",(S,N1))..................
     START at 5 ; Inherited atts:   N1 RepVal 5
     END at   6 ; Synthesized atts: N1 MaxVal 2
    Leaf (ALeaf "2",(S,N1))
```

This example illustrates that complex semantic rules can be accommodated. Our semantic rules declaratively define arbitrary actions on the syntax symbols. For example, the second semantic rule of `tree` is an inherited rule for the second parser `T2`, which depends on its ancestor `T0`'s inherited attribute `RepVal`. The `T0`'s `RepVal` is dependent on its own synthesized attribute `MaxVal`, and eventually this `RepVal` is threaded down as every `num`'s inherited attribute.

4.2 General AGs and Parser Combinators

In an attribute grammar, syntax rules of a context-free grammar are augmented with semantic rules to describe the meaning of the sentences of a context-free language. Although different definitions have been given [6, 7, 8, etc.], we prefer to define a general AG by imposing minimal restrictions on attribute dependencies: a CFG is 4-tuple $G = (N, T, P, S)$, where N is a finite set of non-terminals, T is a finite-set of terminals, P is a finite-set of syntax rules, S is the start non-terminal , $N \cap T = \phi$ and $(\forall p_i \in P)$ p_i is of the form $a ::= b$ where $a \in N$ and $b \in (N \cup T)^*$.

An AG can be formed from G as a 3-tuple $AG = (G, A, R)$, where A is a finite set of attributes and R is a finite set of semantic rules. Each $X \in (N \cup T)$ is associated with a set of attributes $A(X) \subset A$, and each $a \in A(X)$ can be described by a function $r \in R$. The set $A(X)$ can be partitioned into two sets $A_i(X)$ and $A_s(X)$, which represents *inherited* and *synthesized* attributes respectively. A synthesized attribute is an attribute for the LHS symbol of a production rule, and an inherited attribute is associated with a symbol that resides at the RHS of the production rule. We define the inherited and synthesized expressions r_{a_i} and r_{a_s} (w.r.t. a syntax rule $X_0 ::= X_1 X_2 \ldots X_n$), that generate each $a_i \in A_i(X)$ and $a_s \in A_s(X)$ respectively as:

$$
\begin{array}{rcl}
r_{a_i} : \mathbf{P}(\bigcup A(X)) & \rightarrow & A_i(X_x) \\
\alpha & \mapsto & \textit{operations on } \alpha \\
r_{a_s} : \mathbf{P}(\bigcup A(X) - A(X_0)) & \rightarrow & A_s(X_0) \\
\alpha & \mapsto & \textit{operations on } \alpha \\
\text{where } 0 < x \leq n & \text{and} & \mathbf{P} \text{ is the power set}
\end{array}
$$

In functional programming, parser combinators have been used extensively [9, 10, 11, etc.] to prototype top-down backtracking recognizers, which provide modular and executable specifications of grammars that accommodate ambiguity. In basic recursive-descent top-down recognition, rules are constructed as mutually-recursive functions, and after an alternative rule has been applied, the recognizer backtracks to try another rule. Such rec-

ognizers can be constructed as a set of higher order functions, each of which takes an index j as argument and returns a set of indices. Each index in the result set corresponds to a position at which the parser successfully finished recognizing a sequence of tokens (*input*) that began at position j. An empty result set indicates that the recognizer has failed. The result for an ambiguous input contains repetition of one or more ending indices. Using the following basic combinators ($term_{rec}$ and $empty_{rec}$ for terminals and empty symbols, and $\texttt{<|>}_{rec}$ and $\texttt{*>}_{rec}$ for alternative rules and sequencing of symbols respectively) as infix operators, recognizers for a subset of CFGs can be constructed as executable specifications:

$$
\begin{aligned}
term_{rec}\ (t, j) &= \begin{cases} \{\} & , j \geq \#input \\ \{j+1\}, j^{th} \text{ token of } input = t & \\ \{\} & , \text{otherwise} \end{cases} \\
empty_{rec}\ j &= \{j\} \\
(p\ \texttt{<|>}_{rec}\ q)\ j &= (p\ j)\ \cup\ (q\ j) \\
(p\ \texttt{*>}_{rec}\ q)\ j &= \bigcup (map\ q\ (p\ j))
\end{aligned}
$$

However, recognizers constructed with these basic combinators share the shortcomings of naive top-down parsing: 1) they do not terminate for the left-recursive grammars 2) they require exponential time and space for ambiguous input in the worst case. These problems have been addressed in [3, 4] by use of memoization and a technique that restricts the depth of left-recursion.

4.3 Executable Specifications of General AGs

4.3.1 Preliminaries

We have extended the work of Frost et al. [4] to declaratively construct modular and executable specifications of fully-general AGs by providing new combinators. Our executable specifications map an input's start position to a set of end positions with tree structures. We

also thread attributes (i.e. purely-functional and lazy expressions) along with the start and end positions so that they are available for dependencies that are specified in the semantic rules.

We begin by defining some fundamental data structures. Note that from now on, we use the term *parser* for an executable specification of an attribute grammar rule. At any point in the computation, a parser may have a list of synthesized and inherited attributes *Atts* of any user-defined type. A parser, represented by a label (e.g. $Tree$, Num etc.), may have multiple occurrences in a syntax rule, and each occurrence may have different synthesized and/ or inherited attributes. For correct identification, we declare each multiple occurrence as an $Instance$, which is a pair of synthesize/inherited indicator and a unique parser id. For example, an instance of parser $Tree$ could be the pair $(Synthesized\ or\ Inherited, T0)$.

All parsers except the root parser may have a list of inherited attributes for a start position j, and a list of synthesized attributes associated with each successful end position. To accommodate these attributes, we define data-type *Start* and *End* for a parser by pairing the respective indices with a list of instances and attributes. By definition, a parser produces parse-trees based on syntax rules to indicate correct derivations. We use a recursive data-type $PTree$ that compactly represents parse-trees with each component's attribute values and pointer for *where to go next*. The *Result* of a parser's execution is a mapping from *Start* to a list of *End*s where each of the *End* results a list of $PTree$ structures. A memoization technique (section 4.4) is used to prevent redundant computations in order to achieve polynomial time for ambiguous input. The memo-table *State* represents a memory space with *Result*s for parsers which have succeeded or failed. This table is systematically threaded through parser executions using the standard state-monad [12].

```
data Atts       = MaxVal    {getAVAL :: Int}
                | Binary_OP {getB_OP :: (Int -> Int -> Int)} ...

type InsAttVals = [(Instance, [Atts])]
type Start/End  = (Int,InsAttVals)
type Result     = [((Start, End),[PTree Label])]

data PTree v    = Leaf (v,Instance) | Branch [PTree v]
```

```
| SubNode ((Label, Instance), (Start,End))
```

In the following sections, we describe our approach by defining some higher-order functions with segments of Haskell code. The definitions' syntax is straightforward in nature, and can be followed by using a standard literature on Haskell syntax e.g., [13]. We have defined the functions in a declarative manner so that it would be easier to follow for general audience. The full prototype Haskell implementation can be found at the website mentioned in section 4.6.

4.3.2 Combinators for Syntax

We use two basic concepts from [4] to accommodate syntax rules including direct and indirect left-recursion :

- To accommodate direct left-recursion, a left-recursive *Context* is used, which keeps track of the number of times a parser has been applied to an input position j. For a left-recursive parser, this count is increased on recursive descent, and the parser is curtailed whenever the "*left-recursive count* of parser at j exceeds the number of remaining input tokens".
- To accommodate indirect left-recursion, a parser's result is paired with a set of curtailed non-terminals at j within its current parse path, which is used to determine the context at which the result has been constructed at j.

To maintain the flow of attributes when a parser is re-written by its definition, in addition to being executed on the current *Start* and *Context*, we require that it must pass down its unique id and a list of its own inherited attributes so that they can be used when executing the succeeding parsers' semantic definitions. These inherited attributes are defined in terms of semantic rules when the current parser is part of its predecessor's syntax definition.

The current parser's alternative definitions are formed with the combinator `<|>`, which not only accommodates alternative syntax rules but also a list of semantic rules associated with each syntax rule. The semantic rules include synthesized rules for the current parser

and inherit rules for parsers in alternative syntax rules. Threading appropriate rules to appropriate parsers is carried out by a combinator called *parser* (section 4.3.3). Both alternative rules p and q are applied to the current position j and the current context, and the id and inherited attributes of the current parser are also passed down so that they are available to the parsers in both alternatives. All results from p and q are merged together at the end. The operation of the combinator `<|>` can be expressed with the type `(<|>) :: NTType -> NTType -> NTType`, where :

```
type M a           = Start -> Context -> StateMonad a
type ParseResult   = (Context, Result)
type NTType        = Id -> InsAttVals -> M ParseResult
```

In each of the alternative of the current parser, multiple parsers can be sequenced with the sequencing combinator `*>`. In the definition of `*>` for parsers p and q, p is first applied to the current start position and the current context. Then `*>` enables p to compute its inherited attributes using a combinator *nt* (section 4.3.3) from an environment of type *SemRule* that consists of p's precursor's attributes, and the results of all parsers in sequence with p. This *Result* contains sequencing parsers' synthesized and inherited attributes that are embedded in *PTree* structures. Because the attributes are treated as lazy and pure expressions, p's inherited attribute (or any other parser's synthesized or inherited attribute) computations take place only when they are required somewhere else. The next parser q is then sequentially applied to the set of end positions returned by p. q also computes inherited attributes from the same environment. A result from p is joined with all subsequent results from q to form new branch nodes in the tree. The combinator `*>`'s input-output relation can be expressed as type `(*>) :: SeqType -> SeqType -> SeqType`, where :

```
type SemRule = (Instance,(InsAttVals, Id) -> InsAttVals)
type SeqType = Id -> InsAttVals -> [SemRule] -> Result -> M ParseResult
```

The definitions of the AG combinators *term token* and *empty* that define the terminals in the AG rules are analogous to their basic recognizer definitions (section 4.2.2). The only difference is that the terminals are provided with static synthesized attributes. The

term token makes sure that these attributes are passed up with the end positions with a tree of type $Leaf$, only if the terminal successfully consumes an input token. In case of *empty*, the synthesized attributes are passed upwards regardless.

4.3.3 Accommodating Arbitrary Dependencies in Semantics

Our syntax-directed evaluation allows semantic rules for a parser to be defined in terms of potentially unevaluated attributes from the current parser, and its predecessor, successors and sibling parsers. We map synthesized and inherited semantic rules associated with parsers in a syntax rule to the starting and ending positions respectively in the parsers' result-sets. Our method of constructing a result for a parser allows us to establish full call-by-need based arbitrary dependencies between attributes - including dependencies from the right and top. For example, when a parser p_i with a syntax $p_i = p_m$ `*>` p_n is applied to position 1 and successfully ends at position 5, one of p_i's input/output attribute relations could be :

$$SubNode\ p_i\ (inh_{p_i1}, syn_{p_i5})\ =$$
$$[..Branch[SubNode\ p_m\ (inh_{p_m1}, syn_{p_m3})\ , SubNode\ p_n\ (inh_{p_n3}, syn_{p_n5})]..]$$

where, assuming p_m starts at 1 and ends at 3, p_n starts at 3 and ends at 5, inh_{xy} and syn_{xy} represent inherited and synthesized attributes of parser x at position y respectively. From this structure, semantic functions with arbitrary attribute dependencies such as $inh_{p_mi} \leftarrow f(inh_{p_nj}, syn_{p_ik})$, (where f is a desired operation on the attributes) can derive input arguments when required. Note that the output of the example AG from section 4.1 shows actual result structure. An approach based on strict evaluation, rather than lazy, would not achieve this as it maintains a strict evaluation order.

Each AG rule is formed with a higher order wrapper function $parser$, which primarily maps current parser's synthesized rules to all ending points of the syntax result, and assists each parser in the syntax rule to pass down their inherited rules for future use. A parser's synthesized rules are grouped with the identifier (Syn, LHS) from a set of *semantics* that is associated with the current *syntax*. Assuming the syntax would eventually produce a result-set $newRes$, the grouped synthesized rules are mapped to this result using a function

mapSynthesize. This function computes synthesized attributes by applying the semantic specifications on the succeeding parsers' inherited and/ or synthesized attributes for all $PTree$ entries in the result:

```
parser :: SeqType -> [SemRule] -> Id -> InsAttVals -> M ParseResult
parser syntax semantics id inhAtts j context
 = do s <- get
      let ((e,res),s')           =
          let sRule              = groupRule (Syn, LHS) semantics
              tempRes            = syntax id inhAtts semantics res
              ((l,newRes),st)    = unState (tempRes j context) s
              groupRule id rules = [rule | (ud,rule) <- rules, id == ud]
          in  ((l, mapSynthesize  sRule newRes inhAtts id),st)
      put s'
      return (e,res)
```

All parsers in a rule pass down their own identification and a list of inherited attributes so that they can be computed or used in their own definition's semantic rules, if required. This task is done with a higher order function nt, which groups the inherited rules for the current parser based on the pair (Inh, idx) (where idx is the unique Id of the parser x) from *semantics* of current syntax. Then nt facilitates computations of inherited attributes with a mapping function *mapInherited* by applying the grouped rules on a *parser*-provided environment that consists of the predecessor *idp*'s and surrounding parsers' synthesized and inherited attributes. These attributes are to be collected from *newRes*. A parser may have more than one inherited attribute for a particular starting position, which may result from different alternatives. When these attributes are used in any succeeding parser's semantic calculation, they are grouped together under the current parser's single identification so that they are available to carry out desired tasks in the semantic definitions that may require inter-alternative or local result dependencies.

```
nt :: NTType -> Id -> SeqType
nt currentParser idx idp inhAtts semantics newRes
 = let inhRules            = groupRule (Inh, idx) semantics
       ownInAtts           = mapInherited inhRules newRes inhAtts idp
```

```
      groupRule id rules  = [rule | (ud,rule) <- rules, id == ud]
   in  currentParser idx ownInAtts
```

4.3.4 Declarative Executable Specifications of Semantic Rules

We follow a declarative format for the semantic specification which states that synthesized or inherited attribute expressions of a parser can be formed by applying a desired operation on any of the synthesized and/or inherited attributes of any of its surrounding parsers. We define synthesized and inherited semantic expressions with a higher-order function *rule*, which eventually applies user-defined function $userFunction$ on lists of attribute values. *rule* is the generalized version of the synthesized and inherited expression constructing combinators `rule_s` and `rule_i` respectively, and would ultimately return a value of type `SemRule = (Instance,(InsAttVals, Id) -> InsAttVals)` after attaching appropriate type and id. The argument attributes for $userFunction$ are also declaratively specified as synthesized or inherited expressions in $listOfExpr$. These expressions are evaluated with the help of a function $valueOf$, which identifies specified parsers in the user-defined function's argument-expressions either by LHS (i.e., when the current parser's attribute is used in the semantics) or by any other parser's unique id in the syntax.

```
rule sORi typ idp userFunction listOfExp
 = let formAtts  id spec = (id, forNode id . spec)
       forNode   id atts = [(id, atts)]
       newVal            = userFunction (map valueOf listOfExp)
   in  formAtts (sORi,idp) (setAtt typ. newVal)


valueOf sORi typ id_specified  id_current environment
 | pIDspec == LHS   = getAttVals (sORi , id_current  ) environment typ
 | otherwise        = getAttVals (sORi , id_specified) environment typ
```

The user-defined function's argument-expressions are applied to an *environment* of attributes using a recursive function $getAttVals$ to collect the specified parsers' respective attributes. As mentioned in the previous section the environment is formed and provided

with the help of combinators *parser* and *nt*. The *getAttVals* function collects these attributes by comparing the specified parser's id, synthesized/inherited instance and the desired attribute's type with the similar categories from the environment. These comparison factors are threaded down through the current syntax-directed execution path as unevaluated instructions, and the actual comparison takes place only when the attribute values are requested through user-defined functions.

```
getAttVals :: Instance -> InsAttVals -> (a -> AttValue) -> [AttValue]
getAttVals x ((i,v):ivs) typ =
 let getAtts typ (t:tvs) = if (typ undefined) == t
                           then (t :getAtts typ tvs)
                           else getAtts typ tvs
     getAtts typ []      = []
 in  if (i == x) then getAtts typ v else getAttVals x ivs typ
getAttVals x [] typ      = [ErrorVal "ERROR no id found"]
```

The returned attributes are fed into the operations mentioned in the original semantic rules. These operations are straightforward to define. The only requirement for the construction is that these functions perform the desired task on a list of specifications, which are eventually transformed to a list of attribute values. One example of these functions could be `findRep`, which converts the specified synthesized maximum value (computed from the predecessor's alternatives' result-set) to the current parser's inherited replacement value:

```
findRep  specs  = \(atts,i) ->
                  RepVal  (foldr (max) 0 (map (applyMax atts i) (x:xs)))
applyMax y i x  = getAVAL (foldr (getMax)(MaxVal 0) (x y i))
getMax   x   y  = MaxVal  (max (getAVAL x) (getAVAL y))
```

Using these combinators and functions, we can now declaratively construct executable language specifications as fully general attribute rules. For instance, all rules for the section 4.1's example AG are formed with combinators `*>, <|>, parser, nt` and `rule`. One alternative syntax for $tree(T_0)$ ::= $tree(T_1)$ $tree(T_2)$ $num(T_3)$ is expressed with `tree  = parser (nt tree T1 *> nt tree T2 *> nt num T3)`, and one of the inherited semantics for this syntax $RepVal.T_1 \downarrow = RepVal.T_0 \downarrow$ is represented with `rule_i RepVal Of T1 Is findRe`

4.4 Use of Memoization

Norvig [14] first showed that Earley-like [15] polynomial time complexity can be achieved in mutually-recursive top-down parsing by using memoization. Frost et al. [16, 3, 4] also employed similar techniques to parser combinators. We utilize a related memoization technique to achieve polynomial time complexity for recursive grammars. We use a state-monad [12] to systematically thread a memo-table of type `[(Label,[(Start, (Context,Result))])]` through all parser executions whilst maintaining pure functionality.

All of our parsers are executed with a wrapper function *memoize*. If the current parser passes the direct left-recursion depth-check test then a *lookup* is performed based on the parser's *Label* and current position j (which resides in *Start*) to retrieve the previously saved *Result*. If there exists a saved result, then this result is returned if the indirect left-recursion context-comparison test is satisfied. Otherwise, a new result-set is constructed by applying the parser at j with an increased *context* and its own inherited semantics so that they are available for succeeding parsers. The *memoize* function *updates* the memo-table with this new result, inherited semantics and a subset of the current left-rec context corresponding to curtailed non-terminals at the current j. The update operation overwrites any previous entry for the current *Label* and j, since the current entry would subsume all of the previously computed entries. *memoize* also groups local syntactic ambiguities under j in a newly-formed result for a Tomita-like [17] polynomial compact representation, and only returns a reference to this packed entry to the caller, instead of the complete result.

The other task of *memoize* is that, whenever a memoized parser returns a result (either through a lookup or by constructing a new result), it makes sure that the parser's inherited attributes are integrated with the starting point and the synthesized attributes are accompanied with a correct parser *id* at the ending points in the result-set. When we group the local syntactic ambiguities, we also merge synthesized attributes under the current parser's identifier.

4.5 Complexity Analysis

Here we informally discuss the worst-case time and space requirements of our algorithm with respect to the length of the input n. Memoization ensures that a non left-recursive parser is applied to a start position only once. But a left-recursive parser can be applied to the same start position at most n times due to the depth-check. According to [3], the sequencing combinator `*>` performs $O(n^2)$ operations when applying the second parser to every end position returned by the first parser. Therefore, if there were no semantics involved, then a non left-recursive and a left-recursive parser would require $O(n^3)$ and $O(n^4)$ time in the worst-case. While accommodating semantics, we have altered the ambiguity-grouping requirement by collecting distinct attributes resulting in a common end position. This assures the fact the syntactic ambiguity may not necessarily represent semantic ambiguity. In theory, a semantic rule may result in unambiguous attribute values when applied to a group of syntactically ambiguous results, each of whose identical syntactic component may have distinct attribute values. One of the alternative syntax rules $r ::= p$ `*>` q may have at most n syntactic ambiguities, because two parsers' ending positions can be chosen from n start positions in n ways. Overall, the number of multiset results for r is increased from n to n^2. The number of ambiguities arising from a single alternative rule with multiple parsers would depend on the number of parsers in sequence, not only on n. Hence this factor has not been considered in our analysis. If the above parser r is associated with m semantic rules, then `*>` needs to perform extra $m * n^2$ operations. Although p or q's all start-end position pairs may be partitioned into multiple multisets, they depend on p or q's syntactic definitions, which are not considered here as operations related to current parser r. Given a fixed number of semantics, and the highest degree of operation under r is still n^2, the time complexities of non left-recursive and left-recursive parsers remain at $O(n^3)$ and $O(n^4)$ respectively.

Our $PTree$ structure allows us to save results as a list of one-level-depth branches with attribute values attached to pointing sub-nodes. In the memo-table, for each parser's n input positions, we can store n branches corresponding to n end positions. As mentioned earlier, for a branch p `*>` q, there are n possible ambiguities. Hence, we need $O(n^3)$ space in the worst-case w.r.t. the length of the input. The time and space requirements can be

reduced further if we generate only the final semantic value, instead of all possible decorated parse trees, because lazy evaluation would only evaluate those parts of the parse space that are required by the current semantic expression. We suspect that many applications (similar to the one in the next section) would fall under this category.

4.6 Implementation and an Example Application

We have implemented our one-pass top-down AG evaluation algorithm by constructing a set of combinators (as discussed in section 4.3) in a lazy and purely functional language - Haskell. Using these combinators, declarative specifications can be constructed and executed directly without knowing much about Haskell. To test the usability of our system, we have developed a simplified natural language interface. The syntax of the underlying AG is a fully general CFG that has 15 non-terminals and 32 AG rules, and all syntax rules are associated with a subset of a set-theoretic version of Montague semantics that we have extracted from Frost and Fortier [18]. Our interface is able to answer hundreds of thousands of questions about a particular domain - the *solar system*. More information about the implementation and this application, and a version of demo code can be found at `http://cs.uwindsor.ca/~hafiz/fullAg.html`.

We define an attribute type as a set of alternative attributes, where each has its own function type. These attributes are the type-definitions of semantic expressions, which propagate up or down during parser executions. For example:

```
data Att = TERMPHJOIN_VAL {getTJVAL :: ((ES -> Bool) ->
                                        (ES -> Bool)  -> (ES -> Bool))}
         | QUEST_VAL      {getQUVAL :: String}....
```

Next we construct a dictionary to define syntactic categories and their meanings e.g.,

```
dictionary = [("man",   Cnoun, [NOUNCLA_VAL set_of_men])
             ,("orbit", Tverb, [VERB_VAL (tran_verb rel_orbit)]),
             ,("human", Cnoun, meaning_of nouncla "man or woman" Nouncla)
             ,...]
```

Then we modularly define a complete AG specification for the solar system application. For example, part of the definitions of term-phrase and noun-clause are:

```
jointermph =
 memoize Jointermph
 parser (nt jointermph S1 *> nt termphjoin S2 *> nt jointermph S3)
        [rule_s TERMPH_VAL  Of LHS Is
         appjoin1 [synthesized TERMPH_VAL Of S1
                  , synthesized TERMPHJOIN_VAL Of S2
                  , synthesized TERMPH_VAL Of S3]]
 <|>
 parser (nt termph S4)
        [rule_s TERMPH_VAL  Of LHS Is
         copy [synthesized TERMPH_VAL  Of S4]]

snouncla =
 memoize Snouncla
 parser (nt adjs S1  *> nt cnoun S2)
        [rule_s NOUNCLA_VAL Of LHS Is
         intrsct1 [synthesized ADJ_VAL Of S1
                  , synthesized NOUNCLA_VAL Of S2]]
 <|> ...
```

Being right and left recursive, the parser `jointermph` expands to both right and left. The semantic expressions are declaratively defined e.g., `jointermph`'s first semantic rule expresses that `jointermph`'s synthesized attribute `TERMPH_VAL` is formed by joining the synthesized attributes of the r.h.s parsers `S1,S2` and `S3`. The operations, which are applied to syntactic symbols' attributes, are defined based on a set-theoretic version of Montague semantics. For example, `snouncla`'s attribute `NOUNCLA_VAL` is obtained by intersecting sets of adjectives and common-nouns.

An example session with our interface is as follows:

```
which moons that were discovered by hall orbit mars   => [phobos deimos]
every planet is orbited by a moon                     => [false]
```

```
how many moons were discovered by hall or kuiper      => [4]
did hall discover deimos or phobos and miranda        => [no, yes]
etc.
```

Note that the last answer is ambiguous due to the right and left branching of `jointermph`, hence are separated by a comma.

4.7 Related Work

The primary use of AGs has mostly been the specification and construction of compilable parser-generators for programming languages [19]. The classical definition of an AG has often been modified to support the needs of such languages. Swierstra et al. introduced the idea of *higher order attributes* [20, 21] by treating syntax as a part of semantic functions' input and output in a semantics-driven analysis. De Moor et al. [6] achieved semantic modularity by treating attributes as first-class objects. Boyland [22], described an efficient method - *collections* for remote attribute dependencies. *JustAdd* [23] is a compiler-compiler AG system for Java that supports circular referential dependencies with conditional rewriting of ASTs using lazy-evaluation. The *Silver* specification language [24] has been developed primarily based on *forwarding* (a concept similar to higher-order attributes) and other extensions mentioned above. Kats et al. [25] describe *attribute decorators* that support many AG extensions. Similar to *JustAdd*, they use memoization for efficient attribute evaluation.

Our approach differs from these approaches by offering a platform that strictly preserves the syntactic structure of ambiguous CFGs (which includes direct and indirect left-recursions). Our top-down syntax-driven parsing strategy provides a set of non-strict combinators for constructing fully declarative semantic expressions with arbitrary dependencies. In addition to eliminating redundant computations, our use of memoization technique has been specialized to perform extra tasks such as keeping track of non-terminals' context information, merging syntactic ambiguity, mapping and grouping attributes etc.

Even though use of lazy-evaluation to build AG systems has been around for a long time [26, 27, etc.], little work has been done using AGs for natural language processing tasks: Levison and Lessard [28] used AGs to impose some degree of grammatical and semantic

agreement by propagating only inherited attributes downwards while generating natural language text. In the template-based natural language generating system *YAG* [29], AGs have been used to correct partially-specified input by imposing grammatical/number restrictions [30]. Their multi-pass evaluating process begins by initializing inherited attributes with values from the input, then evaluating the rest of the input.

Our approach differs from the last two approaches by being a complete one-pass parsing system that can return either compactly-represented parse trees with attribute values in nodes or just the final answer(s). This is in contrast to the template-based text generators which receive structured input, not natural languages sentences, and don't use AGs for full-blown parsing. By being lazy, we achieve general attribute dependencies by providing more flexible input/output attribute relations. Also, along with declarative semantics, our syntax is highly modular because of the systematic use of parser-combinators as basic building blocks.

4.8 Concluding Comments

We have developed a framework where general CFGs (including ambiguous and left-recursive grammars) can be integrated with semantic rules with arbitrary attribute dependencies as directly-executable and modular specifications. Our approach is based on a top-down parsing method implemented as a set of non-strict combinators resulting in declarative specifications. We utilize a memoization technique for polynomial time and space complexities. In the future we aim to process syntactic and semantic ambiguities based on grammatical and number agreement, type checking and conditional restrictions. By taking advantage of arbitrary attribute dependencies, we plan to model NL features that can be characterized by other grammar formalisms such as unification grammars, combinatory-categorical grammars and type-theoretic grammars. We are constructing formal correctness proofs, and optimizing the implementation for using with very large grammars. We believe that our work will help computational linguists build and test their theories and specifications without worrying about the underlying computational methods, and will also help non-experts create NL interfaces to their applications.

Bibliography

[1] Knuth, D.: Semantics of context-free languages. Theory of Computing Systems, Springer New York **2**(2) (1968) 127–145

[2] Warren, D.: Programming the ptq grammar in xsb. In: Workshop on Programming with Logic Databases. (1993) 217–234

[3] Frost, R., Hafiz, R., Callaghan, P.: Modular and efficient top-down parsing for ambiguous left-recursive grammars. 10th IWPT, ACL. (2007) 109 – 120

[4] Frost, R., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. 10th PADL, ACM. **4902** (2008) 167–181

[5] Bird, R.: Intro. to Functional Programming using Haskell. Prentice Hall (1998)

[6] De Moor, O., Backhouse, K., Swierstra, D.: First-class attribute grammars. In: Third Workshop on Attribute Grammars and their Applications. (2000) 245–256

[7] Tienari, M.: On the definition of attribute grammar. Semantics-Directed Compiler Generation **94** (1980) 408 – 414

[8] Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: PLDI, ACM (1989) 131–145

[9] Frost, R., Launchbury, J.: Constructing natural language interpreters in a lazy functional language. The Computer Journal **32(2)** (1989) 108–12

[10] Hutton, G., Meijer, E.: Monadic parser combinators. J. Funct. Program. **8(4)** (1998) 437 – 444

[11] Wadler, P.: How to replace failure by a list of successes. In: Functional programming languages and computer architecture. Volume LNCS 201. (1985) 113 – 128

[12] Wadler, P.: Monads for functional programming. First International Spring School on Advanced Functional Programming Techniques **925** (1995) 24 – 52

[13] Hudak, P., Peterson, J., Fasel, J.: A gentle introduction to haskell 98. Technical report (1999)

[14] Norvig, P.: Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics **17(1)** (1991) 91 – 98

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

[16] Frost, R., Szydlowski, B.: Memoizing purely functional top-down backtracking language processors. Science of Computer Programming **27(3)** (1996) 263–288

[17] Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Boston, MA. (1986)

[18] Frost, R., Fortier, R.: An efficient denotational semantics for natural language database queries. In: Applications of NLDB. (2007) 12–24

[19] Paakki, J.: Attribute grammar paradigms a high-level methodology in language implementation. ACM Comput. Survey **27**(2) (1995) 196–255

[20] Swierstra, S.D., Alcocer, P., Saraiva, J.: Designing and implementing combinator languages. In: 3rd Summer School on Advanced FP. (1998) 150 – 206

[21] Swierstra, S.D., Vogt, H.: Higher order attribute grammars. In: Attribute Grammars, Applications and Systems. Volume 545. (1991) 256–296

[22] Boyland, J.: Remote attribute grammars. Journal of the ACM **52(4)** (2005) 627 – 687

[23] Ekman, T.: Extensible Compiler Construction. PhD thesis, Comp Science, Lund University (2006)

[24] Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. In: LDTA. (2007) 103 – 116

[25] Kats, L., Sloane, A., Visser, E.: Decorated attribute grammars. attribute evaluation meets strategic programming. In: 18th International Conference on Compiler Construction. (2009) 142 – 157

[26] Augusteijn, L.: The elegant compiler generator system. In: Attribute Grammars and their Applications. (1990) 238–254

[27] Johnsson, T.: Attribute grammars as a functional programming paradigm. In: FP languages and computer architecture. Volume 274. (1987) 154 – 173

[28] Levison, M., Lessard, G.: Application of attribute grammars to natural language sentence generation. AGs and their Applications **461** (1990) 298–312

[29] Mcroy, S., Channarukul, S., Ali, S.: An augmented template-based approach to text realization. Natural Language Engineering. **9(4)** (2003) 381 – 420

[30] Channarukul, S., Mcroy, S., Ali, S.: Enriching partially-specified representations for text realization using an attribute grammar. In: 1st International Natural Language Generation Conference. (2000) 163 – 170

Chapter 5

A System for Modularly Constructing Efficient Natural Language Processors

This paper has been accepted for publication as:

Hafiz, R. and Frost, R. (2011) *A System for Modularly Constructing Efficient Natural Language Processors*. Computational linguistics Applications Conference, Jachranka, Poland, 2011

5.1 Introduction

Traditionally top-down processing has been used for generative grammars (e.g., definite clause grammars (DCGs)[1], and context free grammars (CFGs)) to model natural languages. From a computational point of view, top-down analysis of languages showed early potential by being modular, and by accommodating a wide range of linguistic phenomenon that are essential for deep analysis. Moreover, modular top-down analysis allows construction of larger language processors by piece-wise combination of smaller components. This type of construction is especially useful when processors (e.g., natural language interfaces) compute meanings of sentences using compositional semantics such as Montagovian semantics. Specifying syntax and semantics to describe formal languages using denotational notation of attribute grammars (AGs) has been practiced extensively. However, very little work has shown the usefulness of declarative AGs for computational models for natural language. Previous work falls short in accommodating ambiguous CFGs with left-recursive rules, and providing a declarative syntax-semantics interface that can take full advantages of dependencies between syntactic constituents to model linguistically-motivated cases.

In this paper we show that a unique top-down parsing approach can be used to build a system [1] in a purely functional language Haskell [2] where application developers can specify syntactic and semantic descriptions of natural languages using a general notation of AGs as directly executable specifications. The underlying top-down analysis method parses ambiguous sentences with general CFGs efficiently and allows coupling semantic rules with syntax declaratively. This system can be used not only to produce compact representation of the potentially exponential number of parses using polynomial time and space, and to compute meanings of sentences using compositional semantics, but also to model linguistic properties that may require more restricted generative formalisms than context-free grammars. Some of these properties we demonstrate by only using upward propagating synthesized attributes include characteristics of unification-based formalisms (e.g., subject-verb agreements), unbounded syntactic dependencies in sentences for some form of disambiguation, and generative formalisms that can process cross-serial dependencies and duplicate languages. We also emphasize how executable language processors can

[1] Visit `cs.uwindsor.ca/~hafiz/xsaiga/fullAg.html` for the system code.

be constructed by integrating compositional semantics with a context-free backbone.

Our technique relies on 1) constructing a set of higher-order pure functions that allow the construction of modular and directly-executable specifications of syntax and semantics rules, 2) a context and depth-aware top-down parsing algorithm for general CFGs, and 3) a variation of explicit memorization for time and space efficiency. The lazy-evaluation procedure is the necessary technique for us to impose constraints to model targeted languages, which implicitly makes sure that computations are carried out when they are asked for (call-by-need) and, a particular computation is performed at most once and its result is recalled whenever it is needed again. Our declarative notation allows designers to specify *what* the description of the language is, *what* relationships exist between components, and *what* restrictions are needed to be imposed, rather than *how* these procedures actually work.

5.2 General Notation

Here we introduce the notation that we will use in our system to represent directly-executable specifications of AGs.

5.2.1 Operators for Syntactic Analysis

Consider the following context-free grammar (in BNF) for a segment of English that contains sentences such as `bob saw a nightingale`:

$$
\begin{array}{lll}
sent & ::= & tp\ vp \\
tp & ::= & pnoun\ |det\ np \\
vp & ::= & verb\ tp \\
np & ::= & noun \qquad pnoun ::= \ 'bob' \\
noun & ::= & 'nightingale' \\
verb & ::= & 'saw' \qquad det ::= \ 'a'
\end{array}
$$

Example CFG 1

In the above grammar, a nonterminal sentence (*sent*) is a term phrase(*tp*) followed by a verb phrase (*vp*). A *tp* can be expanded using two alternative rules - one rewrites *tp* to a proper noun *pnoun*, and in the other one *tp* is formed by a determiner (*det*) followed by a noun phrase (*np*). The alternatives are separated with a `|` and terminals are in single quot. When *sent* is applied to `bob saw a nightingale`, it will unambiguously produce a single parse tree.

In our system for executable specifications, we provide a set of higher order functions or combinators to modularly denote syntactic descriptions similar to the above example. We use the combinators `<|>` and `*>` to denote alternating and sequencing of syntax symbols, and `term` to represent a terminal. We will gradually increase this set of combinators or operators to accommodate other linguistic properties. Using these basic combinators, we can represent CFG 1 as the following executable specification:

```
sent  = memoize Sent (tp  *> vp)
tp    = memoize Tp   (pnoun <|> det *> np)
vp    = memoize Vp   (verb*> tp)
np    = memoize Np noun
pnoun = term "bob"
noun  = term "nightingale"
verb  = term "saw"
det   = term "a"
```

In this executable specification, each expression is a function for a non-terminal that maps an index representing the current input position (*Start*) to a parse-tree of the recursive data-type *PTree* (see below) that indicates the structure of the current parse that successfully ends at the position *End*. The final result of complete parsing is a list of these *PTree*s with their starting and ending positions. The embedded $(Start, End)$ pairs in trees work as as pointers to indicate *where to go next*, which allows the trees in the result set to be one-level-depth branches, sub-nodes or leaves. The trees that start and end at identical positions are shared between all non-terminal entries. Note that each non-terminal's functional definition is *memoized* with a wrapper function *memoize* that ensures that a non-terminal (identified by a unique label e.g., `Sent`) is executed at a input position at

most once.

$$
\begin{aligned}
\mathit{type}\ \ \mathit{Start/End} &= \mathit{Int} \\
\mathit{data}\ \ \mathit{PTree}\ \ \mathit{Label} &= \mathit{Leaf}\ \ \mathit{Label} \\
&\mid\ \mathit{Branch}[\mathit{PTree}\ \ \mathit{Label}] \\
&\mid\ \mathit{SubNode}(\mathit{Label}, (\mathit{Start}, \mathit{End})) \\
\mathit{type}\ \ \mathit{Result} &= [((\mathit{Start}, \mathit{End}), [\mathit{PTree}\ \ \mathit{Label}])]
\end{aligned}
$$

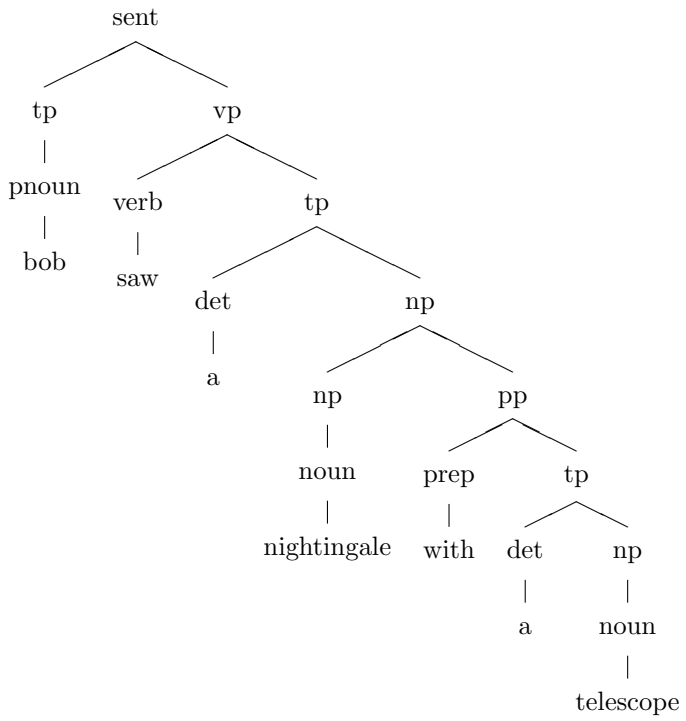
The compact and memoized result for all successful parses is systematically threaded through parser-executions. As pure functional combinators do not have side effects, the threading of the memoized table is done with a *state monad* [3].

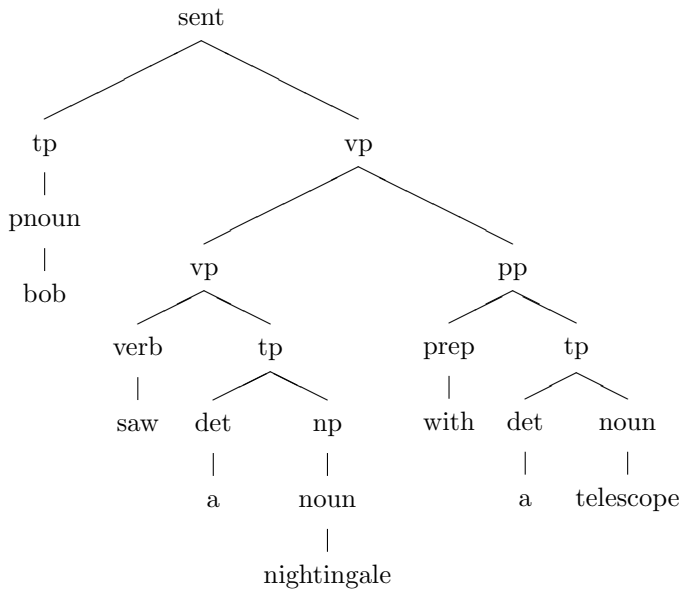
We can extend CFG 1 by adding rules for prepositional phrases *pp* so that we can parse a wide range of sentences such as `bob saw a nightingale with a telescope`. We can also add more terminals according to the target language that we want to model:

$$
\begin{aligned}
sent &::= tp\ vp \\
tp &::= pnoun\ |det\ np \\
pp &::= prep\ tp \\
vp &::= vp\ pp\ |verb\ tp \\
np &::= np\ pp\ |noun \\
pnoun &::= 'bob' \\
noun &::= 'nightingale'\ |'telescope' \\
prep &::= 'with'\ |'on'\ |etc. \\
verb &::= 'saw'\ |'see'\ |etc. \\
det &::= 'the'\ |'an'\ |'a'
\end{aligned}
$$

Example CFG 2

This extended CFG 2 now contains left-recursive rules (e.g., vp::=vp pp), and it can accommodate ambiguity. For example, when we parse the sentence `bob saw a nightingale with a telescope` starting from the non-terminal *sent* using a general parsing algorithm we get the following two parses:





Ambiguous input results two parse trees

As our system allows piecewise integration of smaller components to form a larger description, we can seamlessly extend our previous executable specification of syntax to model CFG 2 (as shown below). This form of modular construction also allows testing and editing of individual components separately without affecting other parts.

```
sent  = memoize Sent (tp   *> vp)
tp    = memoize TP   (pnoun       <|> det *> np)
pp    = memoize PP   (prep *> tp)
vp    = memoize VP   (verb *> tp  <|> vp  *> pp)
np    = memoize NP   (noun        <|> np  *> pp)
pnoun = memoize "pnoun"term "bob"
noun  = memoize "noun" term "nightingale" <|> term "telescope"
prep =  memoize "prep" term "with"<|> etc.
verb  = memoize "verb" term "saw" <|> etc.
det   = memoize "det"  term "a"   <|> etc.
```

This specification can be executed directly using our general top-down parsing algorithm ([4]) to accommodate a subset of English that contains ambiguity. When the starting nonterminal `sent` is applied at the first token position of the sentence `bob saw a nightingale`

`with a telescope`, it produces two parses that are embedded in the following pretty-printed compact format. Two different parses can be extracted if we follow the $(Start, End)$ pointing pairs starting from the `Sent`'s entry. Notice that ambiguities arise from two of the vp's entries where the verb phrase starts at position 2 and ends at position 8 for the sub-string `saw a nightingale with a telescope` using two different sub-tress. Each entry in this compact representation is shared i.e., they are entered exactly once in the result set. For example, the prepositional phrase pp that starts at position 5 and ends at position 8, is shared by the vp and the np.

```
"Sent" (Start: 1,End: 8) Branch [SubNode ("tp",  (1,2)), SubNode ("vp",  (2,8))]

"VP"   (Start: 2,End: 8)[Branch [SubNode ("verb",(2,3)), SubNode ("tp",  (3,8))]
                        ,Branch [SubNode ("vp",  (2,5)), SubNode ("pp",  (5,8))]]
       (Start: 2,End: 5) Branch [SubNode ("verb",(2,3)), SubNode ("tp",  (3,5))]

"TP"   (Start: 1,End: 2) SubNode ("pnoun", (1,2))
       (Start: 3,End: 8) Branch [SubNode ("det",(3,4)), SubNode ("np", (4,8))]
       (Start: 3,End: 5) Branch [SubNode ("det",(3,4)), SubNode ("np", (4,5))]
       (Start: 6,End: 8) Branch [SubNode ("det",(6,7)), SubNode ("np", (7,8))]

"NP"   (Start: 4,End: 8) Branch [SubNode ("np", (4,5)), SubNode ("pp", (5,8))]

"PP"   (Start: 5,End: 8) Branch [SubNode ("np", (5,6)), SubNode ("pp", (6,8))]

"pnoun" (Start: 1,End: 2) Leaf "bob"
"verb"  (Start: 2,End: 3) Leaf "saw"
"det"   (Start: 3,End: 4) Leaf "a"
"noun"  (Start: 4,End: 5) Leaf "nightingale"
"prep"  (Start: 5,End: 6) Leaf "with"
"det"   (Start: 6,End: 7) Leaf "a"
"noun"  (Start: 8,End: 9) Leaf "telescope"
```

5.2.2 Operators for Semantic Analysis

5.2.2.1 Computing Meaning

Our system is ideally suited to incorporate compositional semantics that have a one-to-one correspondence between the rules defining syntactic constructs and the rules stating how the meaning of phrases are constructed from the meanings of their constituents. Richard Montague, who was one of the first to develop a compositional semantics for English, suggested to treat natural language semantics similar to formal language semantics as purely-functional lambda(λ) expressions where larger expressions are constructed by composing smaller expressions. As the original proposal is computationally intractable, we use an efficient set-theoretic version [5].

Alongside with piecewise syntactic extension, we can glue together semantics rules for corresponding syntax rules to compute meanings of a larger set of languages. We extend the set of combinators to integrate semantics with syntax as directly-executable specifications of general attribute grammars. Consider a subset of CFG 2 $sent ::= tp\ vp,\ vp ::= verb\ tp,\ tp ::= pnoun|det\ noun$ that can parse the sentence `bob saw a nightingale`. Following the suggestions of [5], we define an AG where syntax rules are accompanied by sets of semantic rules, which are Montague-style compositional semantics as (λ) expressions:

$$
\begin{array}{lll}
sent(S_0) & ::= & tp(T_0)\ vp(V_0) \\
\quad \{S_0.VAL & \uparrow & = (\lambda p\ p\ T_0.VAL \uparrow) V_0.VAL \uparrow\} \\
tp(T_0) & ::= & pnoun(P_0) \\
\quad \{V_0.VAL & \uparrow & = P_0.VAL \uparrow\} \\
 & | & det(D_0)\ noun(N_0) \\
\quad \{T_0.VAL & \uparrow & = (\lambda p\ D_0.VAL \uparrow\ p)\ N_0.VAL \uparrow\} \\
vp(V_0) & ::= & verb(V_1)\ tp(T_1) \\
\quad \{V_0.VAL & \uparrow & = (\lambda p\ V_1.VAL \uparrow\ p) T_1.VAL \uparrow\} \\
pnoun(P_0) & ::= & 'bob' \\
\quad \{P_0.VAL & \uparrow & = (\lambda p\ 'bob' \in p)\{human\}\} \\
noun(N_0) & ::= & 'nightingale' \\
\quad \{V_0.VAL & \uparrow & = \{nightingales\}\}
\end{array}
$$

$$
\begin{array}{lll}
verb(V_0) & ::= & 'saw' \\
\quad \{V_0.VAL & \uparrow & = (\lambda z\ z(\lambda x \lambda y\ applytransvp(y,x)))\} \\
det(D_0) & ::= & 'a' \\
\quad \{D_0.VAL & \uparrow & = \lambda p \lambda q(p\ \wedge\ q) \neq \phi\}
\end{array}
$$

Attribute Grammar 1.

In the above AG, non-terminals are identified with labels (e.g. V_0, N_0 etc.), and these non-terminals have synthesized attributes (e.g. $V_0.VAL\uparrow$) that propagate upward. If inherited attributes are needed that propagate downward then we can mark them with $\downarrow$s. The semantics for non-terminals are correctly-typed pure functions, and some of them are higher-order functions. For example, the semantic rule for the transitive verb `saw`, which is identified with the $V_0.VAL$, is a function that receives two functions - one defines the term phrase `a nightingale` and the other defines the proper noun `bob` as input arguments. The semantics for `saw` eventually passes the input arguments into a user-defined function *applytransvp* that determines the authenticity of the transitive verb's relation between the subject and the object.

We extended our set of combinators to accommodate AGs like the above, which are descried in section 5.4. We have built combinators `rule_s` and `rule_i` to form synthesized and inherited semantic rules, wrapper functions `parser` and `nt` to form a complete AG expressions, and we refer to syntactic constructs by their unique labels (e.g., `V1, P1` etc.) in order to use them as semantic functions' arguments. By allowing any syntactic construct's attributes that are available in a syntax rule as a semantic functions' arguments, we can form arbitrary dependencies between these constructs that are necessary for many linguistic phenomena. The above AG can be described declaratively with our notation in Haskell's syntax to form a directly-executable specification. Below we show only a few AG rules (the rules for nonterminals *sent*, *tp*, and *vp*) to introduce our notation for integrating semantic rules with syntax rules:

```
1. sent = memoize Sent
2.  (parser (nt tp T0 *> nt vp V0)
```

```
3.  [rule_s val OF LHS  EQ applytp
4.     [synthesized VAL OF V0, synthesized VAL OF T0]]
5. tp   = memoize TP
6.  (parser (nt det D0 *> nt noun N0)
7.  [rule_s val OF LHS  EQ applydet
8.      [synthesized VAL OF D0, synthesized VAL OF N0]]
9.  <|>
10.  parser (nt pnoun P0)
11.  [rule_s Ref OF LHS EQ copy
12.    [synthesized Ref OF P0]])
13. vp  = memoize VP
14.  (parser (nt verb V1 *> nt tp T1)
15. [rule_s val OF LHS  EQ applyvp
16.    [synthesized VAL OF V1, synthesized VAL OF T1]]
        ......
```

In the above specification, lines 2, 6, 10, and 14 are syntax rules for *sent*, *tp*, and *vp*, and the rest are sets of associated declarative semantics. For example, the synthesized `VAL` attributes of non-terminals (which are identified by `LHS`s) are computed with user functions (such as `applytp`) by supplying the required syntactic constructs' attributes (such as `synthesized VAL OF V0`). These user functions perform similar tasks that are mentioned as λ expressions in Attribute Grammar 1. We have constructed sample NL interfaces using our declarative AG notation that uses context-free syntax and compositional semantics. These applications can answer hundreds of thousands of questions about particular domains (see the web-link at footnote 1 for a sample application implementation).

5.2.2.2 Using Semantics for Disambiguation

Here we provide a simple example to show an innovative use of our declarative semantic notation to perform one form of natural language disambiguation.

The sentence `bob saw a nightingale with a telescope` that we parsed using CFG 2, which produced two syntax trees, can be interpreted in more than one way. In a prepositional phrase *pp*, a preposition demonstrates an adjectival or adverbial property by quantifying either the noun phrase *np* or the verb phrase *vp*. According to the first parse, the *np*'s

head noun *nightingale* is quantified by the preposition *with*, which can be misinterpreted as "a telescope carrying nightingale". But in the second parse, the *with* quantifies the verb *saw* from the *vp*, which is semantically more meaningful. Based on this argument, we can make sure that the first parse is discarded while generating only the second parse by using the following attribute grammar:

$$
\begin{array}{lll}
sent(S_0) & ::= & tp(T_0)\ vp(V_0) \\
tp(T_0) & ::= & det(D_0))\ np(N_0) |\ pnoun(N_1) \\
pp(P_0) & ::= & prep(P_1)\ tp(T_0)\ \{P_0.Ref\uparrow\ =\ T_0.Ref\uparrow\} \\
vp(V_0) & ::= & vp(V_1)\ pp(P_0) \\
\ \ \{V_0.Ref & \uparrow & =\ V_1.Ref\uparrow \\
\ \ ,V_0.Kill & \uparrow & =\ V_1.Ref\uparrow\ \notin\ P_0.Ref\uparrow\} \\
 & & |verb(V_1)\ tp(T_1)) \\
\ \ \{V_0.Ref & \uparrow & =\ V_1.Ref\uparrow\} \\
np(N_0) & ::= & np(N_1)\ pp(P_0) \\
\ \ \{N_0.Ref & \uparrow & =\ N_1.Ref\uparrow \\
\ \ ,N_0.Kill & \uparrow & =\ N_1.Ref\uparrow\ \notin\ P_0.Ref\uparrow\} \\
 & & |noun(N_1) \\
\ \ \{N_0.Ref & \uparrow & =\ N_1.Ref\uparrow\} \\
pnoun(N_0) & ::= & bob \\
noun(N_0) & ::= & nightingale\ \{N_0.Ref\uparrow\ =\ \text{bird}\} \\
 & & |telescope\ \ \{N_0.Ref\uparrow\ =\ \text{used to see}\} \\
verb(V_0) & ::= & saw\ \ \ \ \ \ \ \ \ \ \ \{N_0.Ref\uparrow\ =\ \text{to see}\} \\
prep(V_0) & ::= & with \\
det(D_0) & ::= & a
\end{array}
$$

Attribute Grammar 2.

In the above AG, we added semantic rules for agreements with the verb phrase *vp* and the noun phrase *np* that result in rejection of the sub-parse that produces an attribute *Kill* of the value *true*. According to our approach, when a sub-parse is rejected, then the entire

parse is rejected too. These rejection rules check to see whether prepositional phrase *pp* is meaningfully quantifying the neighbouring verb phrase and noun phrase by matching their upward propagating Ref attribute values. In this artificial example, we wanted to enforce the requirement that the noun *telescope* has a meaningful match with the verb *saw* (i.e., the *saw*'s attribute belongs to the *telescope*'s attribute), but not with the noun *nightingale*. In subsequent sections we describe the working mechanism of our approach. The above AG can be described modularly with our notation to create the following declarative specification (we only show the rule for verb phrase *vp*):

```
vp = memoize VP
 (parser (nt vp V1 *> nt pp P1)
 [rule_s Ref OF LHS  EQ copy [synthesized Ref OF V1]
 ,rule_s Kill of LHS EQ notElemOf
    [synthesized Ref OF, synthesized Ref OF P1]]
 <|>
  parser (nt verb V1 *> nt tp T1)
  [rule_s Ref OF LHS EQ copy [synthesized Ref OF V1]
  ,rule_s Kill of LHS EQ False ])
```

Here the second semantic rule for *vp* can discard a parse by checking whether the `Ref OF V1` is a member of `Ref OF P1` with the function $notElemOf$. When this *vp* rule is placed with other rules to form the complete executable AG, and when the root non-terminal *sent* is applied to our example sentence, then it will only generate one compactly-represented parse tree similar to the "Parse Tree 2" on the previous page.

5.3 Background

In this section we briefly describe some related grammar formalisms [6] and the general top-down parsing technique.

5.3.1 Grammar Formalisms

Our approach is based on a context-free backbone. That means the targeted natural language can be generated, recognized, parsed and computed using context-free generative syntax rules. A context-free grammar (CFG) G is 4-tuple $G = (N, T, P, S)$, where N is a finite set of non-terminals, T is a finite-set of terminals, P is a finite-set of syntax rules, S is the start non-terminal , $N \cap T = \phi$ and $(\forall p_i \in P)$ p_i is of the form $a ::= b$ where $a \in N$ and $b \in (N \cup T)^*$.

In an attribute grammar (AG, [7]), syntax rules of CFGs are augmented with semantic rules to describe the meaning of the sentences of a context-free language. Although different definitions are given in [8, 9, 10], we prefer to define a general AG by imposing minimal restrictions on attribute dependencies. An AG is a 3-tuple $AG = (G, A, R)$, where G is the founding CFG, A is a finite set of attributes and R is a finite set of semantic rules. Each $X \in (N \cup T)$ is associated with a set of attributes $A(X) \subset A$, and each $a \in A(X)$ can be described by a function/expression $r \in R$. The set $A(X)$ can be partitioned into two sets $A_i(X)$ and $A_s(X)$, which represent *inherited* and *synthesized* attributes respectively. A synthesized attribute is an attribute for the LHS non-terminal of a production rule, and it can be computed using any attributes from the RHS terminals/non-terminals. Whereas an inherited attribute is associated with a terminal/non-terminal that resides at the RHS of the production rule, which can be computed attributes of terminals/non-terminals that are on the right or on the left of the current non-terminal. In conventional term, inherited attributes propagate downwards and synthesized attributes propagate upwards.

From the viewpoint of pure functions and lazy-evaluation, semantics rules and attributes are interchangeable. When we refer to an attribute we either refer to a value - *static semantics*, or to an unevaluated expression - *dynamic semantics*.

5.3.2 Top-Down Parsing with Parser Combinators

Parser combinators have been used extensively ([11, 12, 13] etc.) to prototype top-down backtracking functional recognizers and parsers to provide modular and piecewise construction of executable specifications of grammars that can accommodate ambiguity. In basic

recursive-descent recognition, syntax rules are constructed as mutually-recursive functions, each of which maps an starting *input* position to a set of indices corresponding to a set of ending positions at which the parser successfully finished recognizing a sequence of *input* tokens. An empty result set indicates that the recognizer has failed. After an alternative rule has been applied, the recognizer backtracks to try another rule. The result for an ambiguous input contains one or more ending indices. In the case of parsing, indices in the result set are replaced by tree structures to show successful parsing structures.

However, top-down recognizers constructed with basic combinators do not terminate for left-recursive grammars, and when extended to parsers they require exponential time and space for ambiguous input in the worst case. We have addressed these problems [14] by the use of a technique that restricts the depth of left-recursive calls, curtails a parse when left-recursive call exceeds the number of remaining input tokens, and that tracks curtailed indirect left-recursive non-terminals to determine the context at which the result has been constructed. We also use memoization techniques for parsers (a technique similar to [15]) that ensures $O(n^3)$ and $O(n^4)$ worst-case time complexities for non left-recursive and left-recursive CFGs respectively. For space efficiency, the potentially exponential number of ambiguous results are represented compactly in a directed acyclic graph with polynomial size space where entries are one-level-depth branches or sub-nodes, and ambiguous entries are grouped under common nodes. These nodes have referencing pointers to construct complete parse trees whenever needed (similar to [16]).

5.4 Declarative and Executable Attribute Grammars

5.4.1 Forming Arbitrary Dependency

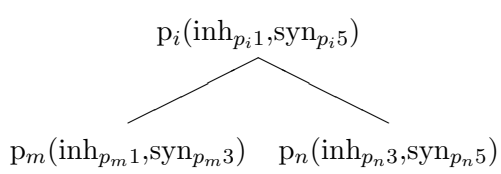
In [17], we demonstrated how declarative semantics rules can be integrated with CFGs' syntax rules in top-down parsing by describing the implementation of combinators in the purely functional language Haskell that enable the formation of language specifications

with modular syntax and semantic. The new parser[2] now has a set of attributes, which are computed from other terminals/non-terminals' attributes that belong to the current parser's syntax definition(s).

Even if a parser contains semantic rules as well as semantic rules, the result of a parser's execution is still a mapping from a start position of input to a set of parse-tree structures with end points. A recursive parse-tree can be a single leaf (to represent a terminal), a sub-node (to represent a non-terminal), or a branch (to represent combinations of terminals and non-terminals, which models more complex syntax). The primary change in the tree data structure is how attributes are represented and threaded so that they are available for dependencies that are specified in the semantic rules. These attributes are essentially purely-functional lazy expressions. For example, the `BinaryOP` attribute in the example below is a function that computes an integer by applying a binary operation to two input integers.

$$
\begin{aligned}
&\mathit{type}\ \ \mathit{Start/End} && = \ (\mathit{Int}, [\mathit{Attributes}]) \\
&\mathit{data}\ \ \mathit{Attributes} && = \mathit{MaxVal}\,\{\mathit{getAVAL} :: \mathit{Int}\} \\
& && \mid\ \mathit{BinaryOP}\,\{\mathit{getBOP} :: (\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int})\}
\end{aligned}
$$

Synthesized and inherited semantic rules associated with a parser are mapped onto the starting and ending positions respectively in the parser's result-set. This facilitates the overall syntax-directed evaluation and allows semantic rules to be denoted in terms of potentially unevaluated attributes from the environment of the current parser, its predecessor, successors, and sibling parsers. For example, when a parser p_i with a syntax rule $p_i = p_m$ `*>` p_n is applied to position 1 and successfully ends at position 5, one of p_i's input/output attribute relations might be:



where, assuming p_m starts parsing at 1 and ends at 3, p_n starts parsing at 3 and ends at 5, inh_{xy} and syn_{xy} represent inherited and synthesized attributes of parser x at

[2]From now on we shall refer non-terminals of attribute grammars as parsers.

position y respectively. From this structure, semantic functions with arbitrary attribute dependencies (like the one below) can acquire input arguments whenever required while they are integrated with the syntax rule:

$$
\begin{aligned}
p_i \quad &= p_m \texttt{ *> } p_n \\
inh_{p_m i} &\leftarrow function(inh_{p_n j}, syn_{p_i k})
\end{aligned}
$$

5.4.2 Construction of Combinators for Executable AGs

Here we informally discuss the construction of the combinators. The alternative combinator `<|>`'s inputs p and q are alternative syntax with lists of respective semantic rules. p and q are executed with the current *start* position and a *context* (for left-recursive calculation), and they pass down their ids and inherited rules so that they can be used in a succeeding parsers' semantics. The associated semantic rules include synthesized rules for the parent parser (referred as LHS) and inherited rules for parsers that are in the associate syntax rule. Threading appropriate semantic rules to the appropriate syntax symbol is done with the combinator *parser*. All results from alternative parsers are merged together at the end.

Terminals or non-terminals (say p and q) can be sequenced with the combinator `*>` to form compound a syntax. This combinator enables p to be applied to the current *start* position and *context*, and to compute its inherited attributes non-strictly using the combinator *nt* from an environment that consists of p's precursor's attributes, and the *result* of the current syntax. This *result* (a tree structure) contains all sequencing parsers' synthesized and inherited attributes. The next parser q is then sequentially applied to the set of end positions returned by p, and computes inherited attributes from the same environment using the combinator *nt*. A result from p is united with all subsequent results from q to form new branches in the tree.

The higher-order function *nt* forms a non-terminal of an AG that enables parsers to pass down their own identification and a list of inherited semantic rules so that they can be used in subsequent AG definitions. These rules are evaluated by applying them on an environment that consists of the predecessor's and surrounding parsers' attributes. This function ultimately maps the current parser's inherited attributes to the starting point of a result so that other inquiring parsers know where to look if needed in the future.

The wrapper function *parser* forms a complete AG rule by mapping the current parser‘s synthesized rules to the ending points of the result, and by assisting each parser in the current syntax rule to have an access to a common environment for their own semantic rules' future needs.

Static synthesized attributes are provided within the definitions of the AG combinators *terminal* and *empty* that define the terminals in the AG rules, and these attributes are passed upward with the end positions in a tree of type $leaf$ only if the terminal successfully consumes an input token. Declarative synthesized and inherited rules are constructed with the help of the function *rule* by applying a desired user operation to a list of synthesized and/or inherited expressions from surrounding parsers. This function helps to find matching expressions from the $parser$ and nt provided environment of attributes by threading some comparison factors through the current syntax-directed execution path as unevaluated instructions. These comparison factors are collected from the user-defined function's argument list of expressions either as LHS (i.e., when the current parser's attribute is used in the semantics) or as any other parser's unique id in the syntax.

Using the combinators described above, general attribute grammars that we mentioned in this paper can be constructed as executable specifications of language descriptions.

5.5 Modeling Unification-Based Formalisms

As with CFGs, unification-based formalisms [18] are language describing machines with added capabilities to impose linguistically motivated restrictions. These formalisms use an information-domain of *feature structures*, which are partial functions from features to their values where the values can be atomic or feature structures. A NL phrase is associated with a feature structure to guarantee its category and properties. *Unification* is a declarative process of restrictive combinations of information from two feature structures to form a new structure, and if the combination fails then the formalism does not recognize the current input. Using unification, phrases such as *a group of boys are attending the seminar for boy scouts* is accepted, while *James are sleeping* is rejected.

The general notation of AGs that we introduced earlier can be utilized for modular

and declarative modeling of unification. We can compare *feature structures* with a set of synthesized or inherited expressions. The unification takes place by comparing evaluated values of these attributes according to the restriction that is needed to model linguistic correctness. We introduce a special-purpose synthesized attribute *Kill* of type *bool* for all syntax symbols in an AG. The one-pass syntax and semantic analysis algorithm is modified so that when the set of synthesized expressions for a non-terminal has an expression dedicated for the *Kill* attribute, and when this attribute is evaluated to *True* then the entire current result is discarded. This step is required to ensure that if a condition or restriction in the semantics fails then the current parse should not be a member of the result. When the expression for *Kill Bool* is mapped onto the current result, a predefined operation *nomatch* tries to determine whether all input argument expressions are evaluated to the same value or not:

$$
\begin{aligned}
&nomatch(e1 : e2 : inputExps, environment)\\
&\quad = \textbf{do let } att1 \leftarrow e1(environment),\ att2 \leftarrow e2(environment)\\
&\qquad\qquad \textbf{if } att1 /= att2 \textbf{ then return } Kill\ True\\
&\qquad\qquad \textbf{else return } nomatch(inputExps, environment)
\end{aligned}
$$

$$
\begin{aligned}
&nomatch([e1, e2], environment)\\
&\quad = \textbf{do let } att1 \leftarrow e1(environment),\ att2 \leftarrow e2(environment)\\
&\qquad\qquad \textbf{if } att1 /= att2 \textbf{ then return } Kill\ True\ \ \textbf{else return } Kill\ False
\end{aligned}
$$

These input expressions form restricted dependencies that can be used to impose linguistically motivated conditions in conjunction with compositional semantics. The following simple directly executable AG shows that phrases such as *moons that spin* is recognized as a grammatically correct sentence, but if we test the input is the phrase *moons that spins* then the parse fails because of the grammatical disagreement that sets *Kill True* for the sentence.

```
sent = memoize Sent
 (parser (nt termph S1 *> nt relpro S2*> nt vbph S3)
  [rule_s Kill OF LHS  EQ nomatch
   [synthesized VAL  OF S1, synthesized VAL  OF S3]
```

```
   ,rule_s SENT_VAL OF LHS EQ apply_termphrase
    [synthesized TERMPH_VAL OF S1
    ,synthesized RELPRO_VAL OF S2
    ,synthesized VERBPH_VAL OF S3]])

termph = memoize Termph (terminal (term "moons")
                     [TERMPH_VAL set_of_moons,   VAL "plural"]
 <|> terminal (term "planets")
                     [TERMPH_VAL set_of_planets, VAL "plural"])

vbph   = memoize Vbph   (terminal (term "spin")
                     [VERBPH_VAL set_of_spin,    VAL "plural"]
 <|> terminal (term "spins")
                     [VERBPH_VAL set_of_spin,    VAL "singular"])

relpro = memoize Relpro
    (terminal (term "that") [RELPRO_VAL intersect]
 <|> terminal (term "who")  [RELPRO_VAL intersect])
```

Note that the other semantic rules are constructed based on the set-theoretic version of Montague's compositional semantics. Our declarative notation for compositional semantics allows us to establish unbounded dependency that exists in relative clause sentences. According to Montague, relative pronouns (*relpro*) act as a conjunction (i.e. *and*) for two surrounding phrases p and q (i.e., $and = \lambda p \lambda q(\lambda z(p\ z\ \wedge\ q\ z)))$. In the set-theoretic version, the *and* is defined as set intersection between two phrases, which are computed as sets too. In the above example, relative pronouns (*that*, *who*) have a synthesized attribute *intersect*, which is a function that expects two sets as input to perform the set intersection operation. This attribute is propagated upwards to the *sent*'s definition as `RELPRO_VAL`. The *sent* computes the final `SENT_VAL` through an *apply_termphrase* function that provides two sets (`TERMPH_VAL` and `VERBPH_VAL`) to `RELPRO_VAL` from two surrounding phrases - term phrase and verb phrase.

In [19] Correa systematically compared properties of attribute grammars and unification grammars, and demonstrated that attribute grammars are more general than unification

grammars in terms of expressive power and computational efficiency. He also mentioned difficulties in implementing generalized attribute grammars because of problems related to attribute evaluations. As our parsing strategy is non-strict, our attribute evaluation order is demand driven and untangled by nature. Moreover, unlike unification-based notations, we can refer to unevaluated expressions as attributes in semantic dependencies, which enriches the overall declarativeness and modularity of the language description.

5.6 Accommodating Phenomena Beyond Context-Free

It has been well-documented that a few instances of natural language (e.g., some sentences from Dutch or Swiss German) may not be accommodated by context-free grammars; rather they need a stronger formalisms that are often referred to as mildly context-sensitive grammars (MCSGs) [20]. MCSGs are strictly stronger than CFGs and strictly weaker than context sensitive grammars in terms of generative power. Many formal formalisms strictly or weakly satisfy characteristics of MCSG such as tree adjoining grammar (TAG), combinatory categorial grammar (CCG), linear indexed grammar (LIG) etc. (see [21] for the proof of their equivalence). Common characteristics that define MCSG formalisms include 1) that they contain all context-free languages, 2) that the membership problem is solvable in polynomial time, 3) that a member language grows constantly, and 4) that these formalisms contain the following non context-free languages [22] -

$L1$. *multiple agreement*
$\{a^n b^n c^n | n \geq 0\}$
$L2$. *crossed agreement*
$\{a^n b^m c^n d^m | n, m \geq 0\}$
$L3$. *duplication*
$\{ww | w \in \{a, b\}^*\}$

We now show how to take advantage of arbitrary attribute dependencies to model above languages. As AGs generate languages that are founded on context free grammars, it is

not directly possible to generate the above non context-free languages, but it is possible to non-strictly filter out sentences that do not belong to the target language by imposing semantic restrictions that guarantee only acceptance of members of $L1$, $L2$, and $L3$. A CFG S ::= As*> Bs*> Cs, As ::= As*> $a|a$, Bs ::= Bs*> $b|b$, Cs ::=Cs*>$c|c$ that generates a set of sentences (in the form of any number of as followed by any number of bs then any number of cs) can also potentially include sentences generated by $L1$. But, if we can enforce that the As, Bs, and Cs can be expanded or recursively re-write themselves exactly the same number of times (i.e., n) then this CFG is restricted to a grammar only for $L1$. This restriction is shown with the following example AG by checking As, Bs, and Cs's *count* attribute value. Each time the As, Bs, or Cs accepts a token (e.g., a, b, or c respectively), their synthesized *count* attribute's values increase, and at the root non-terminal S these *count*s are compared with *nomatch*. If any mismatch is found then the S's $Kill$ attribute is set to $True$, and the current parse is entirely discarded.

```
s = memoize Ss
    (parser (nt as S0  *> nt bs S1 *> nt cs S2 )
    [rule_s Kill OF LHS EQ  nomatch
    [synthesized Count OF S0, synthesized Count OF S1
    ,synthesized Count OF S2]])

as = memoize As (parser (nt as S0 *> nt term_a S1 )
                [rule_s Count OF LHS  EQ increment
                [synthesized Count OF S0]]
            <|>  parser (nt term_a S1)
                [rule_s Count OF LHS  EQ copy
                [synthesized Count OF S1]])

bs = memoize Bs (parser (nt bs S0 *> nt term_b S1 )
                [rule_s Count OF LHS  EQ increment
                [synthesized Count OF S0]]
           <|>   parser (nt term_b S1)
                [rule_s Count OF LHS  EQ copy
                [synthesized Count OF S1]])

cs = memoize Cs (parser (nt cs S1 *> nt term_c S1 )
```

```
                [rule_s Count OF LHS  EQ increment
                [synthesized Count OF S1]]
           <|>   parser (nt term_c S1)
                [rule_s Count OF LHS  EQ copy
                [synthesized Count OF S1]])

term_a = memoize TA (terminal (term "a") [Count 1])
term_b = memoize TB (terminal (term "b") [Count 1])
term_c = memoize TC (terminal (term "c") [Count 1])
```

Note that in the above example *increment* and *copy* are user-defined functions (like *nomatch*) that increment and make a copy of an attribute value whenever demanded. Using a similar concept, an AG for $L2$ can be constructed declaratively by using another user defined function *crossmatch* to impose cross serial dependencies. The *crossmatch*'s four argument expressions' *count* values are now used to decide whether the root non-terminal's $Kill$ value is $True$ or $False$:

$$
\begin{aligned}
&crossmatch([e1, e2, e3, e4], environment) \\
&\quad = \textbf{do let } count1 \leftarrow e1(environment),\ count2 \leftarrow e2(environment) \\
&\qquad\qquad\quad count3 \leftarrow e3(environment),\ count4 \leftarrow e4(environment) \\
&\quad \textbf{if } count1 == count3 \textbf{ and } count2 == count4 \\
&\quad \textbf{then return } Kill\ False \textbf{ else return } Kill\ True
\end{aligned}
$$

The application of *crossmatch* in AG the for the $L2$ can be shown with the following partial executable specification:

```
s = memoize Ss
 (parser (nt as S0 *> nt bs S1 *> nt cs S2 *> nt ds S3 )
 [rule_s Kill OF LHS EQ  crossmatch
  [synthesized Count OF S0, synthesized Count OF S1
  ,synthesized Count OF S2, synthesized Count OF S3]]).....
```

To model the duplicate or 2-place copy language $L3$ with our notation of AGs we can use a stack or list-like data structure as an attribute type. The backbone CFG that potentially

can contain members of $L3$ along with other unwanted sentences can be represented as follows:

$S ::= W{*>}W$

$W ::= a{*>}W \mid b{*>}W \mid \epsilon$

The definition for W satisfies generating $\{a, b\}^*$, but one W following another does not guarantee generating $L3$. If W had an attribute as a list of indicators representing all accepted as and bs by the current W, and if we can set the root S's $Kill$ value to $True$ whenever two lists of indicators for two consecutive Ws are not identical then unwanted sentences will be discarded. We can accomplish this objective with our notation for AG as shown with the partially-complete example below. Assume that new user function *nomatchList* checks equality of two operand lists, and *addToList* attaches the first argument to the second, which is a list originated as an empty list from the alternative *empty*.

```
s = memoize Ss (parser (nt w S0  *> nt w S1)
    [rule_s Kill OF LHS EQ  nomatchList
    [synthesized List OF S0, synthesized List OF S1]])

w = memoize W  (parser (nt term_a S0 *> nt W S1 )
    [rule_s Count OF LHS  EQ addToList
    [synthesized Val  OF S0, synthesized List OF S1]]

 <|> parser (nt term_b S0 *> nt W S1 )
    [rule_s Count OF LHS  EQ addToList
    [synthesized Val  OF S0, synthesized List OF S1]])

 <|> terminal (empty) [List []].....
```

The above example specifications of executable AGs for languages $L1$, $L2$, and $L3$ could also have been constructed differently by utilizing different variations of semantics constrains by introducing new inherited and/or synthesized attributes.

5.7 Concluding Comments

We have demonstrated how the declarative notation for general AGs can be used to create executable specifications to modularly prototype many NLP tasks. Our approach is founded on a top-down parsing technique that accommodates any form of CFGs coupled with declarative semantics. The overall evaluation technique is non-strict and treats attributes as unevaluated functions, and as a result arbitrary dependencies between syntactic components can be described in the semantics. We have shown that restrictions can be imposed using special-purpose attributes and customizable semantic functions to model natural language properties such as unification, and phenomena that cannot be accommodated with only CFGs. In the future we plan to investigate further incorporating dependencies involving inherited attributes along with synthesized attributes to construct general-purpose natural language interfaces where compositional semantics would compute meanings. We are intending to investigate many other forms of natural language disambiguation using our declarative restrictive capability. We also believe that our transparent syntax-semantic interface can used effectively for syntax-based machine translation applications.

Bibliography

[1] Pereira, F.C.N., Warren, D..: Definite clause grammars for language analysis. Artificial Intelligence **13** (1980)

[2] Hudak, P., Peterson, J., Fasel, J.: A gentle introduction to haskell 98. Technical report (1999)

[3] Wadler, P.: Monads for functional programming. 1st Spring School on Advanced FP **925** (1995) 24 – 52

[4] Frost, R., Hafiz, R., Callaghan, P.: Modular and efficient top-down parsing for ambiguous left-recursive grammars. ACL-IWPT (2007) 109 – 120

[5] Frost, R., Fortier, R.: An efficient denotational semantics for natural language database queries. In: Applications of NLDB. (2007) 12–24

[6] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall (1972)

[7] Knuth, D.: Semantics of context-free languages. Theory of Computing Systems, Springer New York **2**(2) (1968) 127–145

[8] De Moor, O., Backhouse, K., Swierstra, D.: First-class attribute grammars. In: 3rd Workshop on AGs and their Applications. (2000)

[9] Tienari, M.: On the definition of attribute grammar. Semantics-Directed Compiler Generation **94** (1980) 408 – 414

[10] Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: PLDI, ACM (1989) 131–145

[11] Wadler, P.: How to replace failure by a list of successes. In: FP languages and computer architecture. Volume 201. (1985) 113 – 128

[12] Frost, R., Launchbury, J.: Constructing natural language interpreters in a lazy functional language. The Computer Journal **32(2)** (1989)

[13] Hutton, G., Meijer, E.: Monadic parser combinators. J. Funct. Program. **8(4)** (1998) 437 – 444

[14] Frost, R., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. ACM-PADL (2008) 167–181

[15] Norvig, P.: Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics **17(1)** (1991) 91 – 98

[16] Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1986)

[17] Hafiz, R., Frost, R.A.: Lazy combinators for executable specifications of general attribute grammars. In: ACM-PADL. (2010) 167–182

[18] Shieber, S.: An introduction to unification-based theories of grammar,. CSLI Lecture Notes Series, University of Chicago Press. (1986)

[19] Correa, N.: Attribute and unification grammar: A review and analysis of formalisms. Annals of Mathematics and AI **873-105** (1993)

[20] Joshi, A.K.: Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? NL Parsing, Cambridge University Press (1985)

[21] Vijay-Shanker, k., Weir, D.J.: The equivalence of four extensions of context-free grammars. Theory of Computing Systems **27 (6)** (1994) 511–546

[22] Kudlek, M., Martn-Vide, C., Mateescu, A., MitranaSource, V.: Contexts and the concept of mild context-sensitivity. Linguistics and Philosophy **26 (6)** (2003) 703–725

Chapter 6

Conclusion

6.1 Concluding Comments

This thesis shows that it is possible to construct modular natural language processors as directly-executable specifications. Notation is introduced that facilitates the declarative specification of languages as context-free grammars (CFGs), and meaning as compositional semantics that can define arbitrary dependencies among syntactic constructs. The resulting language description is realized using of a set of purely-functional and non-strict combinators, which are implemented in the modern functional language Haskell (`haskell.org`). When language specifications are constructed with these combinators and applied to a (potentially ambiguous) natural language sentence or query, the underlying processor generates either a compact representation of parses (with semantic values attached with respective syntactic constructs), or just the final result. Besides deep syntactic and semantic analysis of natural languages, the overall declarative approach also accommodates linguistic features (such as long-distance and cross-serial dependencies) that may need a more constrained formalism than CFGs.

This thesis has presented four peer-reviewed papers, which address various aspects of the overall problem. **Chapter 2** describes a general top-down parsing algorithm that incorporates techniques for accommodating ambiguity, and direct and indirect left-recursive grammars in polynomial time and space. Even though problems related to accommodating

ambiguity, left-recursion in polynomial time and space have been previously addressed separately, this is the first approach that incorporates all of these features within one algorithm. The algorithm keeps track of the depth of left-recursive parses and curtails the search when there is no possibility of constructing a valid parse, records the context for computing parses with indirect left-recursion in order to prevent missing parses, employs a memoization technique for time efficiency, and represents the potentially exponential number of ambiguous parses in a compact space as a directed acyclic graph (DAC).

Chapter 3 describes a modular approach to implement the parsing algorithm of **Chapter 2** for building executable syntactic descriptions of natural languages. Pure functional languages enable the construction of executable top-down recursive descent parsers using *combinators* (higher-order functions) as infix operators. This approach has been extended to incorporate properties described in **Chapter 2** by defining a set of combinators, which can be used to prototype any form of CFG. The combinators are used to modularly sequence grammar symbols for forming a rule with multiple symbols, and to define multiple alternative rules. The method for accommodating left-recursive grammars, which has been described in Chapters 2 and 3, has already been applied in practice. The technique has been adopted by the developers of the widely-used LEPL parser library which is written in the Python programming language. More can be found at `http://www.acooke.org/lepl/implementation.html` about the LEPL parser library.

As pure functions do not permit side effects, an abstraction technique called a *monad* is used to systematically thread the table containing parsing results through parsers' executions so that the memoization technique is properly employed. Recursive data structures are used to represent parsing results compactly. The combinators are implemented in the pure functional programming language Haskell (visit `cs.uwindsor.ca/~hafiz/xsaiga/imp.html` for the implementation), and using the combinators executable syntactic descriptions can be constructed without mastering the underlying details of the parsing algorithm. The implementation has been tested successfully to support the claim of polynomial time of space complexities by experimenting with practical natural-language grammars (e.g., ambiguous grammar with 5,226 rules), and with some massively-ambiguous artificial grammars.

Chapter 4 describes a new technique to integrate declarative semantics with the gen-

eral top-down syntax analysis in one pass. In order to achieve the declarativeness of general attribute grammars (AGs), the extended approach uses a non-strict evaluation technique to enable the specification of semantics by establishing arbitrary dependencies among the syntactic categories. The previous set of combinators was extended to integrate the declarative semantics with the syntax so that a large language-description can be built by piecewise combination of smaller syntactic and semantic specifications. The dependencies in semantics, which produces upward or downward propagating *meanings*, are especially useful for modelling compositional semantics of natural language. Prototype database query processors are constructed (visit `cs.uwindsor.ca/~hafiz/xsaiga/fullAg.html` or the **Appendix B** for the implementation) with Montague-style compositional semantics as executable specifications. These processors can answer thousands of questions.

An innovative technique is introduced in **Chapter 5** that describes how to impose restrictions on the generative power of the target language using declarative semantics. This technique non-strictly uses conditional pruning so that many linguistically-motivated requirements can be accommodated. This chapter is based on a paper that has been accepted in Computational Linguistics - Applications 2011 conference. An improved and modified version of this paper has recently been accepted for presentation in a peer-reviewed international conference - The 10th Mexican International Conference on Artificial Intelligence, the proceedings of which will be published in a special issue of the Springer LNAI series.

Appendix A discusses some formal properties of the approach. Termination of the approach is established (Appendix A.1) by showing that there exists an well-founded ordering throughout the syntactic and semantics analysis with a least value and a finite upper limit. Appendix A.2 represents the syntax-analysis algorithm as a deduction system with a few new inference rules (for accommodating left-recursion) in order to prove the correctness of the algorithm. Appendix A.3 discusses time and space complexities of the parsing algorithm w.r.t. the input length and the size of a the grammar. The changes in complexity of the approach when semantics are integrated, is discussed in the Section 4.5 of the Chapter 4.

Below is a list of features that distinguish this thesis from previous work -

- This is the first general top-down recursive-descent parsing algorithm that accommodates direct and indirect left-recursive ambiguous grammars.

- Memoization has been used with to-down parsing for time efficiency. However, this work is the first that uses shared forest (compact representation technique) with a general top-down parsing for polynomial space complexity.
- This work accommodates fully-general attribute dependencies among syntactic components in declarative semantics (including inheritance from the right) in one pass.
- This thesis now contains formal proofs of correctness, termination and complexity for the general top-down parsing.
- The top-down analysis system has been implemented in Haskell and made available to the Haskell community.
- This thesis includes experimental results that were conducted using as set of diverse test cases, which support theoretical claims.

In the future, I plan to further investigate the usefulness of arbitrary attribute dependencies to build general-purpose (potentially web-based) natural-languages interfaces and database query processors. I believe that the notion of Montague's compositional semantics can be implemented, and expanded efficiently utilizing my current work. I am also interested in expanding the syntax-semantic interface for building syntax-based machine-translation applications. I have just begun exploring a limited form of the disambiguation task. I plan to take advantage of the context-awareness and restrictive capability of my approach to incorporate a wide range of syntactic, semantic and word-sense disambiguation in natural language processing applications. Another important NLP task is error detection and correction. My approach fails for erroneous input, my approach simply returns an empty list indicating a failure. However, as our parsing process builds a memo-table that is filled with information, potentially it can be used to detect errors and to make suggestions for correction. The correctness analysis proof (Appendix A) of top-down parsing does not explicitly consider the context-matching condition when dealing with indirect left-recursion. In future I intend to provided a more complete proof. I have shown that the declarative semantics can be used to impose restrictions to model some languages that are beyond context-free. I plan to perform a formal analysis about the restrictive capability for the wide coverage of grammars.

Appendix A

Formal Properties of Our Approach

A.1 Termination

We discuss termination by adopting a technique for *termination analysis of recursive functions* where the key idea is to ensure that there exits an well-founded ordering so that the argument of each recursive call establishes an inequality with the corresponding inputs. This comparison is done in terms of a *measure* (an element of the well-founded set), which changes after each recursive call. A *measure-function* maps a data-object (which is related to the corresponding recursive-functions input) into a member of a well-founded ordered set. For example, consider a recursive function definition $f(x^1) = ...f(x^i)....$ To show that f terminates, the first task would be to define a measure function $|.|$ that maps some data related to the input (here, the input to f) to a *measure* (the output of $|.|$, which is a natural number). The next step is to define a well-founded order ($\succ$, with the least element 0) of decreasing *measures* for all executions of f until $f(x^k)$, which is the last recursive call - $|x^1|... \geq ...|x^i|.. \geq ...|x^k|$. If the above inequality holds, then the function f terminates. Note that a well-formed order can be also formed with the $\leq$ inequality too as long as there is a least element and a finite upper bound exist in the ordering.

In [1], Giesl described the following four general steps to prove termination of an algorithm of the form $f(x^*) = ...f^{'}(y^*)...$, where f and $f^{'}$ are mutually and/ or nested recursive functions:

1. Generating a measure-function $|.|$ and a well-founded ordering $\succ$.
2. Generating an Induction Lemma IL: $|\exists y_i \in y^*| \succ |f^{'}(y^*)|$.
3. Proving the Induction Lemma.
4. Proving the inequality $|x^*| \geq |y^*|$ or $|y^*| \geq |x^*|$

For our purpose, we recall a few of the terms and definitions that we described in earlier chapters. We represent a left-rec counter (Chapter 2) for a parser P_i at position j with c_{ij}, where $1 \leq i \leq P_{\#}$ (the $P_{\#}$ is the length of the set of parsers) and $1 \leq j \leq input_{\#}$ (the $input_{\#}$ is the length of the input). We also recall that an entry of a parser's *Result* is of the form $((Start, End), [PTree\ Label])$ where *Start*s and *End*s are list of starting and ending indices with associated inherited and synthesized attributes respectively, and that the memo-table is a list of parse names along with respective *Result*s, which also include a *Context* that keeps track of c_{ij} (Chapter 4).

Definitions 2.1: Our **measure function** $|.|$ maps a parser's input arguments to a natural number. Without loss of generality, we can state $|.| :: (Start, (Context, memo\ table)) \rightarrow starting\ index_j\ in\ Start + \#\ of\ attributes\ in\ Start{+}c_{ij} \in Context$.

Lemma 1.1: From Chapters 2 and 3 we recall that the starting position j ranges from $1 \leq j \leq input_{\#}$, and the range of left-rec counter c_{ij} is $0 \leq c_{ij} \leq (input_{\#} - j + 1)$. From these ranges we can identify that the least value of $|.|$ is $1 + att_i + 0$ (where att_i is the number of inherited attributes for a parser), and maximum value is also finite - $1 + att_i + input_{\#}$.

Induction Lemma IL:
$[((Start^{'}, End), [Tree])], (Context^{'}, memo\ table)) \leftarrow \forall p_x \in P,$
$p_x\ (Start, (Context, memo\ table))$

$\Rightarrow Result = [] \vee$
$\forall(i, att_i) \in Start, \forall(j, att_s) \in End, \forall(i^{'}, att_{i^{'}}) \in Start^{'},$

1. $j \geq i \wedge$
2. $(j + \#\ of\ att_s) + (i^{'} + \#\ of\ att_{i^{'}}) \geq (i + \#\ of\ att_i) \wedge$
3. $\forall c_{xj} \in Context^{'} \geq \forall c_{xj} \in Context$

where att_s and att_i are sets of synthesized and inherited attributes of p_x respectively.

Proof by Induction on $p_x \in P$:
In the *base case*, if p_x is an *empty* symbol or just a terminal, then

1. $j > i$ (definition of *terminal*, increases starting positions is upon successful acceptances), or $j = i \wedge$ (definition of *empty*, which does not increases is upon successful acceptances).
2. By definition, there are no inherited semantic rules for *terminal* or *empty* (i.e., $\#\ of\ att_i = 0$). This condition and the **IL 1** proves **IL 2**.
3. In cases of *terminal* or *empty*, no recursion takes place, which results no changes in left-rec counts (Chapter 2). Hence **IL 3** holds.

As the *inductive hypothesis*, we assume that **IL** holds for a set of parsers P. Now, we have to show that **IL** also holds for $P \cup \{p_y\}$.

Case 1: Let $p_y = memoize\ parser(nt\ p_m$ `*>` $nt\ p_n)\ (att_{s_{p_y}} \cup att_{i_{p_m}} \cup att_{i_{p_n}})$, for some $p_m, p_n \in P$, where $att_{s_{p_y}}$ is a set of list of synthesized semantics for p_y, and $att_{i_{p_m}}$ and $att_{i_{p_n}}$ are sets of inherited semantics associated with parsers p_m and p_n. **IL** 1, 2, and 3 stay valid as follows:

1. From the hypothesis p_m's any output position $\geq p_m$'s input position (which is also p_y's input position i), and from the definition of `*>` (Chapter 3) and the hypothesis

p_n's output positions ($\forall j$, which is also output positions of p_y) $\geq p_n$'s input positions (which are also p_m's output position). Hence, $\forall j \geq i$, for the p_y.

2. The definitions of *parser* and *nt* (Chapter 4) ensure the mapping of synthesized and inherited semantic rules to respective parsers' starting and ending positions in their result sets. Definitions of pure functions and lazy evaluation allow us to consider # of the set of semantic rules = # of the set of attributes. Note that, this equivalence is only possible when the attribute grammar is *well-formed* i.e., there is no cyclic dependency in the semantics. This equivalence and **IL 1** proves **IL 2**.

3. Regardless of whether p_y is left, right or non recursive, as **IL 3** holds for p_m and p_n (from the hypothesis) and from the definition of *memoize*, **IL 3** also holds for p_y. In general, if p_y is left-recursive then it's c_{yj} will be increased for the same input position j, and this change is recorded in the *Context*. Otherwise, c_{yj} will will remain the same.

Case 2: Let $p_y = memoize\ parser(p_m\ att$ `<+>` $p_n\ att^{'})$, for some $p_m, p_n \in P$, and where *att* and $att^{'}$ are sets of semantic rules for p_y that are associated with alternatives p_m and p_n respectively. **IL** 1,2, and 3 for p_y can be proven using similar logic from *the case 1*, and from the definitions of *parser*, `<+>`, and the inductive hypothesis.
Hence, **IL** for $P \cup \{p_y\}$ holds from **IL** for P and **IL** for p_y.

Proof of Termination:
We have to show that an inequality holds for consecutive recursive parser-executions that ensure an well-founding ordering according to the definition of the measure function $|.|$. Let some $p_x \in P$ rewrites to (p_k `*>` $p_{x'}$) $(att_{s_{p_x}} \cup att_{i_{p_k}} \cup att_{i_{p_{x'}}})$ in its definition. Assuming, p_k is applied with $(Start, (Context, memo\ table))$ at i and returns
$([Result],(Context^{'}, memo\ table^{'}))$, and from the definition of `*>`, p_x's recursive call, $p_{x'}$ is executed on $\forall j$ (the end-points of p_k that are in the *Result*). According to the induction lemma, each of these recursive calls results in the following cases:
Case 1: Regardless of the whether p_x's calls are left-recursive or right-recursive, **IL 2** always holds as as the number of attributes and number of semantic rules are constant and equal for a parser.

Case 2: When $j > i$ and $\forall c_{xj} \in Context' \geq \forall c_{xj} \in Context$, p_x's recursive call's are right-recursive as some left-most tokens are consumed before the next recursive call and the the left-rec context for input-positions does not increase.
Case 3: When $j = i$ and $\forall c_{xj} \in Context' \geq \forall c_{xj} \in Context$, the p_x's recursive call's are left-recursive as no input is consumed and the left-rec context continues to increase until the left-recursive parse is correctly curtailed.

From the above cases and the definition of $|.|$, for all recursive executions of p_x, the following holds - $|\text{arguments to } p_{x'}| \succ |\text{arguments to } p_x|$, and according to the *Lemma 1.1* this is a well-founded ordering.

A.2 Correctness Analysis

In order to prove that our top-down parsing algorithm is correct, we have to show that it is sound and complete. As mentioned in Chapter 3 of [2], for a grammar G (that describes a language $L(G)$) and every input sentence w, a parsing algorithm is *sound* if the algorithm accepts input sentence w, then w is in $L(G)$; and is *complete* if the algorithm accepts all sentences w is in $L(G)$, then .

Pereira and Warren [3] and Shieber et al. [4] demonstrated how traditional parsing algorithms, including the top-down recursive-descent parsing algorithm, can be expressed as deduction systems. The primary advantage of expressing a parsing algorithm as a deduction system is that a deduction system excludes control techniques, data-structures etc. that are primarily needed for implementation. Hence, deduction systems are very useful for discussing formal properties of parsing algorithms. As the conventional top-down parsing does not terminate while parsing with left-recursive grammars, here we extend the deduction system to incorporate our technique (which accommodates left-recursive grammar rules).

In a deduction system, an inference rule specifies a single parsing step that is used to derive statements about the grammatical status of input strings from other such statements. The general form of an inference rule is:

$$\frac{antecedents}{consequent} \text{ side conditions}$$

A grammatical deduction system is defined by a set of rules of inference and a set of axioms given by appropriate formula schemata. A derivation of a formula B (consequent) from assumptions $A_1, .. , A_m$ (antecedents), when side conditions hold, is a sequence of formulas $S_1, .. , S_n$ such that $B = S_n$. Shieber et al. [4] expressed a top-down recursive-decent algorithm as the following deductive system:

`Item form:`	$[\bullet\beta, j]$
`Axioms:`	$[\bullet S, 0]$
`Goals:`	$[\bullet, n]$
`Inference Rules`	
`Scanning:`	$\dfrac{[\bullet w_{j+1}\beta, j]}{[\bullet\beta, j+1]}$
`Prediction:`	$\dfrac{[\bullet B\beta, j]}{[\bullet\gamma\beta, j]}\ \mathrm{B} \rightarrow \gamma$

Given a context-free grammar (CFG) $G = \langle N, \Sigma, S, P\rangle$ and an input string $w = w_1..w_n$ to be parsed, item $[\bullet\beta, j]$ asserts that a sub-string of the input up to the length j $(1 \leq j \leq n)$ forms a sentential form of the language $S \overset{*}{\Rightarrow} w_1 w_2 ... w_j \beta$ by subsequent applications of the scanning and the prediction rules starting from the axiom $[\bullet S, 0]$. The scanning and prediction rules show how to logically infer a new token and a new rule respectively. The goal item $[\bullet, n]$ asserts that the complete input (i.e., $S \overset{*}{\Rightarrow} w_1 w_2 ... w_n \beta$) has been accepted by the given grammar. This deduction system is not capable of producing a goal item when the given CFG contains left-recursion. This is owing to the fact that, in the inference rule for a production, the side-condition will be $B \rightarrow B...$, which results non-termination. Following the technique discussed in Chapters 2 and 3, we add two more inference rules for the production (we call them curtailment rules) that ensure proper accommodation of the left-recursive grammars:

`Additional Inference Rules`

`Curtailment 1:` $\dfrac{[\bullet B_c\beta, j-1]}{[\bullet B_{c+1}\beta, j-1]}\ \mathrm{B} \rightarrow \mathrm{B}\gamma'|\gamma'', \mathrm{c} < \mathrm{n} - \mathrm{j}$

`Curtailment 2:` $\dfrac{[\bullet B_c\beta, j-1]}{[\bullet\gamma''\beta, j-1]}\ \mathrm{B} \rightarrow \mathrm{B}\gamma'|\gamma'', \mathrm{c} \geq \mathrm{n} - \mathrm{j}$

`where,` c `represents how many times a left-recursion (e.g.,` $B \rightarrow B\gamma^{'}|\gamma^{''}$`)`
`has been applied at the current position` j`, and` c`'s initial value is` 0`.`

Below we discuss how these curtailment inference rules also generate the same item by satisfying the parsing assumptions when $\beta^{'}$ is left-recursive.

Lemma 2.1: If some rule $B \rightarrow B\gamma^{'}|\gamma^{''}$ is left-recursive then in the case of applying a production inference rule, either curtailment inference rule 1 or 2 applies. We inductively show (on the length of the input left to parse $l = n - j$) that these rules are just enough to allow a left-recursive non-terminal's alternatives to compute results. Note that n is the total length of the input.

When $n - j = l = 1$ (i.e., only one token is left), curtailment rule 1 is applied, and it increases the left-rec counter c for the current left-recursive non-terminal B from the initial 0 to 1). This recursion lets the recursive depth grow one level only for this one token. The increment of c satisfies the condition $c \geq n - j$ of the curtailment inference rule 2, and allows the alternatives ($\gamma^{''}$) of B to be applied at the current position for the left token only once. Allowing more recursive depth will cause excessive applications of alternatives, and less recursive depth will cause missing parses that could generate from the alternatives.

From the inductive hypothesis, if the lemma holds for $l = n - 1$, then we just have to show that for $l = n$, the alternatives are given at most one chance to parse the last token. This last statement directly follows from the base case of the induction. Hence, Lemma 2.1 states that the added curtailment inference rules ensure right recursive depth for left-recursive parses for an input of any length.

Correctness of Our General Top-Down Parsing: Given a CFG $G = \langle N, \Sigma, S, P \rangle$, and an input $w_1..w_j..w_n$ with any $w_j \in \Sigma$, to prove that our parsing algorithm is **correct**, we have to show that the algorithm is **sound** i.e., $[\bullet\beta, j] \Rightarrow S \overset{*}{\Rightarrow} w_1 w_2 ... w_j \beta$, and the algorithm is **complete** i.e., $S \overset{*}{\Rightarrow} w_1 w_2 ... w_j \beta$, $\Rightarrow [\bullet\beta, j]$. In other words, we have to show $S \overset{*}{\Rightarrow} w_1 w_2 ... w_j \beta$, $\Leftrightarrow [\bullet\beta, j]$. Below, we provide a sketch proof (in accordance with [4]) of these properties using induction on input length (i.e., on j).

In the base case, when $j = 0$, we have to show that $S \overset{*}{\Rightarrow} \beta \Leftrightarrow [\bullet\beta, 0]$. We have $[\bullet S.0]$ as an axiom, and applying one $\overset{*}{\Rightarrow}$ on S, we have $[\bullet\beta, 0]$ (*completeness*). On the other hand, $[\bullet\beta, 0]$ states that there is a scanning rule that does not consume any token, and from the axiom $[\bullet S.0]$ we infer that $S \overset{*}{\Rightarrow} \beta$ (*soundness*).

As the inductive hypothesis, we assume that for some $j^{'}$ (where $0 \leq j^{'} < j$), $S \overset{*}{\Rightarrow} w_1 w_2 ... w_{j^{'}} \beta$, $\Leftrightarrow [\bullet\beta, j^{'}]$ holds.

For *completeness*, if $S \overset{1}{\Rightarrow} w_1 w_2 ... w_j \beta$, then the rule $S \rightarrow w_1 w_2 ... w_j \beta$ must be in the grammar G, and one application of this prediction rule and j applications of the scanning rules on the axiom $[\bullet S.0]$ will generate the item $[\bullet\beta, j]$. If $S \overset{+}{\Rightarrow} w_1 w_2 ... w_j \beta$, then assume that $S \overset{*}{\Rightarrow} w_1 w_2 ... w_{j^{'}} \beta^{'} \overset{1}{\Rightarrow} w_1 w_2 ... w_{j^{'}} \beta$, where $\beta^{'} \rightarrow w_{j^{'}+1} .. w_j \beta$. From the induction hypothesis, from $S \overset{*}{\Rightarrow} w_1 w_2 ... w_{j^{'}} \beta^{'}$ the item $[\bullet\beta^{'}, j^{'}]$ will be generated. If the definition for $\beta^{'}$ is not left-recursive then after applying the production inference rule for $\beta^{'}$ in $[\bullet\beta^{'}, j^{'}]$ and $j - (j^{'} + 1)$ scanning applications the item $[\bullet\beta, j]$ will be generated. But, if the definition for $\beta^{'}$ is left-recursive then after $j - (j^{'} + 1)$ applications of the curtailment inference rule 1, one application of the curtailment inference rule 2, and $j - (j^{'} + 1)$ applications of the scanning rules the item $[\bullet\beta, j]$ will be generated (from the Lemma 2.2).

On the other hand, for *soundness*, we have to show $[\bullet\beta, j] \Rightarrow S \overset{*}{\Rightarrow} w_1 ... w_{j^{'}} .. w_j \beta$. From the hypothesis, for the item $[\bullet w_j \beta, j - 1]$, a grammatical sentential form $S \overset{*}{\Rightarrow} w_1 ... w_{j-1} \beta$ exists. In order to apply the scanning inference rule on β in $[\bullet w_j \beta, j - 1]$ to consume the token w_j, there must be a grammatical rule in the CFG G for β that can be used as a production inference rule's side condition. This rule can be left or right recursive, or a non-recursive rule that accepts w_j at the left-most position. Hence, after applying one production and one scanning inference rules in $[\bullet w_j \beta, j - 1]$, we find $S \overset{*}{\Rightarrow} w_1 ... w_{j^{'}} .. w_j \beta$.

A.3 Complexity Analysis

In this section we used the big O notation to represent the worst case upper bound. Theoretically the O is a set of functions i.e., $O(g(n)) = \{f(n)|0 \leq f(n) \leq cg(n)\}$. Here we have slightly abused the notation for simplicity i.e., we used $O(g(n)) = cg(n) = f(n)$ where $f(n)$ is the worst performing function from the set.

A.3.1 Time Complexity w.r.t. the length of input n

We show that the worst-case time complexity of parsing an input (of length n) is $O(n^3)$ for non left-recursive grammars and $O(n^4)$ for left-recursive grammars where n is the length of the input. We begin with a few assumptions that are drawn from Chapters 2 and 3:

Assumptions:

1. Merging two result-sets, curtailment-condition check, and incrementing left-rec counter requires $O(n)$ time.
2. Operations related to manipulating context and reason (to decide reusing for non-terminals that are in the path of indirect left-recursion) need $O(n)$ time.
3. Parsers constructed with only terminals require $O(n)$ time.
4. Memo-table update and lookup require $O(n)$ time.
5. Functionalities for *creating pointers* and *grouping ambiguity* need $O(n)$ and $O(n^2)$ time respectively.
6. Creating n-ary branches requires $O(n)$ time.

Lemma 3.1: The alternative combinator requires $O(n^2)$ time.

Application of $memoize(p_p$ <|> $p_q)$ at start-position j involves the following steps (assuming parsers p_p and p_q have already been applied on j and their results are available):

1. One memo-table lookup + create pointer requires $O(n)$
2. If the lookup fails:
   2.1 Condition for curtailment check requires $O(n)$
   2.2 If 2.1 permits:
      2.2.1 Merging two results returned by p_p and p_q) requires $O(n)$ (merging reasons de-

pends on # of non-terminals)

2.2.2 Ambiguity packing of new result + updating the packed result to memo-table + create pointer requires $O(n^2)$

The above time complexities directly follow from the assumptions. Hence, the worst case complexity remains $O(n^2)$.

Lemma 3.2 The sequencing combinator requires $O(n^2)$ time.

In case of $memoize(p_p$ `*>` $p_q)$ at start-position j, in the worst-case p_p may returns a set of results of length n and according to the definition of `*>` (Chapter 2), p_q has to be applied to every (end-position +1) of p_p's result-set and each pointer of p_p's result-set needs to create an n-ary branch with a pointer-set returned by p_q's application to each (end-position + 1) of pp. Application of `*>` involves the following steps (assuming p_p and p_q had already been applied and their results are available):

1. One memo-table lookup + creating pointer requires $O(n)$
2. If the lookup fails:

2.1 Condition for curtailment check requires $O(n)$

2.2 If 2.1 permits:

2.2.1 Application of p_q on p_p's result set + forming the n-ary branching between each pointer of p_p with the corresponding pointer-set of p_q + merging their results and reasons to form new result requires $O(n * n) = O(n^2)$.

2.2.2 Ambiguity packing of the new result + updating the packed result in the memo-table + creating the pointer requires $O(n^2)$.

The above time complexities follow from the assumptions. Hence, the worst case complexity remains at $O(n^2)$.

Applying the arguments below, we can conclude that a non left-recursive parser and a left-recursive parser require $O(n^3)$ and $O(n^4)$ time respectively:

Non left-recursive parsers need $O(n^3)$ time: Given an input of length n and a parser-set (i.e., nonterminals in the grammar) P of size $P\#$, each non left-recursive parser, $p \in P$ is applied to a particular start-position $j \in n$ at most once, as at least one left-most token of current input would be consumed before recursive execution of p again. Irrespective of how many times the `*>` combinator is used in each of the parser's alternative definition, all parsers in the sequence can be applied sequentially to at most n start positions. So, from Lemma 3.2, each alternative definition formed with the `*>` combinator would require at most $n * O(n^2)$ or $O(n^3)$ time. Also, multiple occurrences of alternative rules (formed with `<+>` combinator) need $O(n^2)$ time (from Lemma 3.1). Hence, the worst case time complexity for non left-recursive parsers is $O(n^3)$.

Left-recursive parsers need $O(n^4)$ time: On the other hand, a direct left-recursive parser is applied to a particular start-position $j \in n$ at most n times (follows from the condition for curtailment), and an indirect left-recursive parser at most $n * P\#$ times. Therefore, the total number of times a left-recursive parser can be executed is n^2. Hence, following Lemmas 3.1 and 3.2., a left-recursive parser needs $n^2 * O(n^2)$ or $O(n^4)$ time for the worst case.

A.3.2 Parsing complexity w.r.t. the size of the grammar p

Here we informally discuss how the size of the grammar affects the worst case complexities. We consider the number of non-terminals (p) of a context-free grammar as the size of the grammar, because we need to have production rules for all of these non-terminals.

Consider a production rule for a non-terminal p_i of the form $p_i = (p_a ...$ `*>` $...p_m)$ (where $1 \geq a, i, m \geq p$), which has k number of alternatives that are constructed with the combinator `<|>`. In theory, any finite number of terminals and nonterminals can be sequenced together to form each of p_i's alternatives. However, in the worst case, each of the grammatical symbol that participates in the formation of the rule must consume at least one input token. Hence, there can be at most n non-terminals in a sequence (i.e., $m = n$ in p_i's definition), where n is the length of the input.

According to the definition of *memoization*, one non-terminal computes results at any input position j at most once, and considering p (total number of non-terminals) in the calculation, `*>` and `<|>` need $O(pn^2)$ time in the worst case. So, if p_i's alternative definition is non left-recursive and if it has at most n nonterminals then, from the discussion in the previous section, the worst case time requirement is $n * O(pn^2)$ or $O(pn^3)$. Whereas, if p_i's one alternative definition is left-recursive and if it has at most n nonterminals then, from the discussion in the previous section, the worst case time requirement is $n*n*O(n^2)$ or $O(pn^4)$.

The number of alternatives k for a non-terminal p_i is independent of the length of the input or the grammar size as we have defined it. Hence, for the number of non-terminals p, the worst case time complexity is $p * O(pn^3) * k$ or $O(kp^2n^3)$ for the non left-recursive case, and $p * O(pn^4) * k$ or $O(kp^2n^4)$ for the left-recursive case.

A.3.3 Space Complexity w.r.t. the length of the input n

As mentioned in Chapters 2 and 3, the memo-table used for parsing is of type $\{(parser\ name, \{(start\ position, (left\ rec\ context, \{result\}))\}\}$, and the *result* is of type $(start\ position, end\ position + 1), \{tree\})$. Each parser has at most n memo-table entries and each of them has a result-set of size at most n. Hence, the total number of entries is $p * n^2$, where n is the length of the input, and p is the number of parser. However, each entry of the result-set can be paired with a tree of size at most $n * c$. Here c is a constant that depends on number of symbols on the right-hand side of a rule, or the type of the definition of the right-hand side symbols (e.g., whether they rewrite to an empty terminal) etc., but not on the length of the input. Hence, the final size of the memo-table is $O(p * n^3)$.

Bibliography

[1] Giesl, J.: Termination of nested and mutually recursive algorithms. Journal of Automated Reasoning. **19 (1)** (1997) 1 – 29.

[2] Kallmeyer, L.: Parsing Beyond Context-Free Grammars. Springer Science and Business Media (2010)

[3] Pereira, F.C.N., Warren, D.: Parsing as deduction. In: Proceedings of the 21st annual meeting on Association for Computational Linguistics. Association for Computational Linguistics Morristown, NJ, USA. (1983) 137144.

[4] Shieber, S.M., Schabes, Y., Pereira, F.C.N.: Principles and implementation of deductive parsing. Harvard University as Technical Report TR-11-94. (1995)

Appendix B

Example NL Query-Processor Using the New Combinators

Here we demonstrate the usefulness of our approach by building a directly-executable language processor. We use the modular and declarative notation of our approach, which we described in the previous chapters, by piece-wise integration of smaller syntactic and semantic components to form the larger NL query processor. This is a domain-specific sample application, which is capable of answering numerous questions about the solar system.

B.1 The Dictionary

We begin by constructing a dictionary to define members of syntactic categories and their meanings. These are the terminals or words that build an input natural-language query. For example, a `planet` is a common noun that shares meaning with a *set of planet*. It is also possible to define meaning of some words by relating them to phrases or words that have equivalent meanings. For example, the meaning of the common noun `discoverer` can be assigned by computing the phrase `person who discovered something`.

```
dictionary =
 [("man",              Cnoun,     [NOUNCLA_VAL set_of_men]),
  ("men",              Cnoun,     [NOUNCLA_VAL set_of_men]),
  ("thing",            Cnoun,     [NOUNCLA_VAL set_of_things]),
  ("things",           Cnoun,     [NOUNCLA_VAL set_of_things]),
  ("planets",          Cnoun,     [NOUNCLA_VAL set_of_planet]),
  ("planet",           Cnoun,     [NOUNCLA_VAL set_of_planet]),
  ("woman",            Cnoun,     [NOUNCLA_VAL set_of_woman]),
  ("women",            Cnoun,     [NOUNCLA_VAL set_of_woman]),
  ("sun",              Cnoun,     [NOUNCLA_VAL set_of_sun]),
  ("moon",             Cnoun,     [NOUNCLA_VAL set_of_moon]),
  ("moons",            Cnoun,     [NOUNCLA_VAL set_of_moon]),
  ("satellite",        Cnoun,     [NOUNCLA_VAL set_of_moon]),
  ("satellites",       Cnoun,     [NOUNCLA_VAL set_of_moon]),
  ("atmospheric",      Adj,       [ADJ_VAL     set_of_atmospheric]),
  ("blue",             Adj,       [ADJ_VAL     set_of_blue]),
  ("blue",             Adj,       [ADJ_VAL     set_of_depressed]),
  ("solid",            Adj,       [ADJ_VAL     set_of_solid]),
  ("brown",            Adj,       [ADJ_VAL     set_of_brown]),
  ("gaseous",          Adj,       [ADJ_VAL     set_of_gaseous]),
  ("green",            Adj,       [ADJ_VAL     set_of_green]),
  ("red",              Adj,       [ADJ_VAL     set_of_red]),
  ("ringed",           Adj,       [ADJ_VAL     set_of_ringed]),
  ("vacuumous",        Adj,       [ADJ_VAL     set_of_vacuumous]),
  ("exist",            Intransvb, [VERBPH_VAL  set_of_things, VAL 2]),
  ("exists",           Intransvb, [VERBPH_VAL  set_of_things, VAL 1]),
  ("spin",             Intransvb, [VERBPH_VAL  set_of_spin, VAL 2]),
  ("spins",            Intransvb, [VERBPH_VAL  set_of_spin, VAL 1]),
  ("the",              Det,       [DET_VAL function_denoted_by_a]),
  ("a",                Det,       [DET_VAL function_denoted_by_a]),
  ("one",              Det,       [DET_VAL function_denoted_by_one]),
  ("an",               Det,       [DET_VAL function_denoted_by_a]),
  ("some",             Det,       [DET_VAL function_denoted_by_a]),
  ("any",              Det,       [DET_VAL function_denoted_by_a]),
  ("no",               Det,       [DET_VAL function_denoted_by_none]),
  ("every",            Det,       [DET_VAL function_denoted_by_every]),
```

```
("all",              Det,       [DET_VAL function_denoted_by_every]),
("two",              Det,       [DET_VAL function_denoted_by_two]),
("mars",             Pnoun,     [TERMPH_VAL (test_wrt 12), VAL 1]),
("bernard",          Pnoun,     [TERMPH_VAL (test_wrt 55)]),
("bond",             Pnoun,     [TERMPH_VAL (test_wrt 67)]),
("venus",            Pnoun,     [TERMPH_VAL (test_wrt 10)]),
("cassini",          Pnoun,     [TERMPH_VAL (test_wrt 65)]),
("dollfus",          Pnoun,     [TERMPH_VAL (test_wrt 63)]),
("Fouuntain",        Pnoun,     [TERMPH_VAL (test_wrt 62)]),
("galileo",          Pnoun,     [TERMPH_VAL (test_wrt 56)]),
("hall",             Pnoun,     [TERMPH_VAL (test_wrt 54)]),
("herschel",         Pnoun,     [TERMPH_VAL (test_wrt 64)]),
("huygens",          Pnoun,     [TERMPH_VAL (test_wrt 66)]),
("kowal",            Pnoun,     [TERMPH_VAL (test_wrt 57)]),
("kuiper",           Pnoun,     [TERMPH_VAL (test_wrt 69)]),
("larsen",           Pnoun,     [TERMPH_VAL (test_wrt 61)]),
("lassell",          Pnoun,     [TERMPH_VAL (test_wrt 70)]),
("melotte",          Pnoun,     [TERMPH_VAL (test_wrt 60)]),
("nicholson",        Pnoun,     [TERMPH_VAL (test_wrt 59)]),
("perrine",          Pnoun,     [TERMPH_VAL (test_wrt 58)]),
("pickering",        Pnoun,     [TERMPH_VAL (test_wrt 68)]),
("almathea",         Pnoun,     [TERMPH_VAL (test_wrt 21)]),
("ariel",            Pnoun,     [TERMPH_VAL (test_wrt 47)]),
("callisto",         Pnoun,     [TERMPH_VAL (test_wrt 25)]),
("charon",           Pnoun,     [TERMPH_VAL (test_wrt 53)]),
("deimos",           Pnoun,     [TERMPH_VAL (test_wrt 20)]),
("dione",            Pnoun,     [TERMPH_VAL (test_wrt 40)]),
("earth",            Pnoun,     [TERMPH_VAL (test_wrt 11)]),
("enceladus",        Pnoun,     [TERMPH_VAL (test_wrt 38)]),
("europa",           Pnoun,     [TERMPH_VAL (test_wrt 23)]),
("ganymede",         Pnoun,     [TERMPH_VAL (test_wrt 24)]),
("hyperion",         Pnoun,     [TERMPH_VAL (test_wrt 43)]),
("iapetus",          Pnoun,     [TERMPH_VAL (test_wrt 44)]),
("io",               Pnoun,     [TERMPH_VAL (test_wrt 22)]),
("janus",            Pnoun,     [TERMPH_VAL (test_wrt 36)]),
("jupiter",          Pnoun,     [TERMPH_VAL (test_wrt 13)]),
```

```
("jupitereighth",      Pnoun,     [TERMPH_VAL (test_wrt 32)]),
("jupitereleventh",    Pnoun,     [TERMPH_VAL (test_wrt 31)]),
("jupiterfourteenth",  Pnoun,     [TERMPH_VAL (test_wrt 34)]),
("jupiterninth",       Pnoun,     [TERMPH_VAL (test_wrt 33)]),
("jupiterseventh",     Pnoun,     [TERMPH_VAL (test_wrt 29)]),
("jupitersixth",       Pnoun,     [TERMPH_VAL (test_wrt 27)]),
("jupitertenth",       Pnoun,     [TERMPH_VAL (test_wrt 28)]),
("jupiterthirteenth",  Pnoun,     [TERMPH_VAL (test_wrt 26)]),
("jupitertwelfth",     Pnoun,     [TERMPH_VAL (test_wrt 30)]),
("luna",               Pnoun,     [TERMPH_VAL (test_wrt 18)]),
("mercury",            Pnoun,     [TERMPH_VAL (test_wrt  9)]),
("mimas",              Pnoun,     [TERMPH_VAL (test_wrt 37)]),
("miranda",            Pnoun,     [TERMPH_VAL (test_wrt 46)]),
("neptune",            Pnoun,     [TERMPH_VAL (test_wrt 16)]),
("nereid",             Pnoun,     [TERMPH_VAL (test_wrt 52)]),
("oberon",             Pnoun,     [TERMPH_VAL (test_wrt 50)]),
("phobos",             Pnoun,     [TERMPH_VAL (test_wrt 19)]),
("phoebe",             Pnoun,     [TERMPH_VAL (test_wrt 45)]),
("pluto",              Pnoun,     [TERMPH_VAL (test_wrt 17)]),
("rhea",               Pnoun,     [TERMPH_VAL (test_wrt 41)]),
("saturn",             Pnoun,     [TERMPH_VAL (test_wrt 14)]),
("saturnfirst",        Pnoun,     [TERMPH_VAL (test_wrt 35)]),
("sol",                Pnoun,     [TERMPH_VAL (test_wrt  8)]),
("tethys",             Pnoun,     [TERMPH_VAL (test_wrt 39)]),
("titan",              Pnoun,     [TERMPH_VAL (test_wrt 42)]),
("titania",            Pnoun,     [TERMPH_VAL (test_wrt 49)]),
("triton",             Pnoun,     [TERMPH_VAL (test_wrt 51)]),
("umbriel",            Pnoun,     [TERMPH_VAL (test_wrt 48)]),
("uranus",             Pnoun,     [TERMPH_VAL (test_wrt 15)]),
("venus",              Pnoun,     [TERMPH_VAL (test_wrt 10)]),
("discover",           Transvb,   [VERB_VAL (trans_verb rel_discover)]),
("discovers",          Transvb,   [VERB_VAL (trans_verb rel_discover)]),
("discovered",         Transvb,   [VERB_VAL (trans_verb rel_discover)]),
("orbit",              Transvb,   [VERB_VAL (trans_verb rel_orbit)]),
("orbited",            Transvb,   [VERB_VAL (trans_verb rel_orbit)]),
("orbits",             Transvb,   [VERB_VAL (trans_verb rel_orbit)]),
```

```
("is",                Linkingvb, [LINKINGVB_VAL  id]),
("was",               Linkingvb, [LINKINGVB_VAL  id]),
("are",               Linkingvb, [LINKINGVB_VAL  id]),
("were",              Linkingvb, [LINKINGVB_VAL  id]),
("that",              Relpron,   [RELPRON_VAL    intersect]),
("who",               Relpron,   [RELPRON_VAL    intersect]),
("which",             Relpron,   [RELPRON_VAL    intersect]),
("and",               Verbphjoin,[VBPHJOIN_VAL   intersect]),
("or",                Verbphjoin,[VBPHJOIN_VAL   union]),
("and",               Nounjoin,  [NOUNJOIN_VAL   intersect]),
("or",                Nounjoin,  [NOUNJOIN_VAL   union]),
("by",                Prep,      [PREP_VAL       id]),
("and",               Termphjoin,[TERMPHJOIN_VAL termph_and]),
("or",                Termphjoin,[TERMPHJOIN_VAL termph_or]),
("and",               Sentjoin,  [SENTJOIN_VAL   sand]),
("does",              Quest1,    [QUEST1_VAL     yesno]),
("did",               Quest1  ,  [QUEST1_VAL     yesno]),
("do",                Quest1,    [QUEST1_VAL     yesno] ),
("what",              Quest2,    [QUEST2_VAL     whatq]),
("who",               Quest2,    [QUEST2_VAL     whoq]),
("which",             Quest3,    [QUEST3_VAL     whichq]),
("what",              Quest3,    [QUEST3_VAL     whichq]),
("how",               Quest4a,   [QUEST3_VAL     howmanyq]),
("many",              Quest4b,   [QUEST3_VAL     howmanyq])  ]
++
[("human",       Cnoun,    meaning_of nouncla Nouncla "man or woman"),
 ("person",      Cnoun,    meaning_of nouncla Nouncla "man or woman"),
 ("discoverer",  Cnoun,    meaning_of nouncla
                              Nouncla "person who discovered something"),
 ("discoverers", Cnoun,    meaning_of nouncla
                              Nouncla "person who discovered something"),
 ("humans",      Cnoun,    meaning_of nouncla Nouncla "man or woman"),
 ("people",      Cnoun,    meaning_of nouncla Nouncla "man or woman"),
 ("orbit",       Intransvb,meaning_of verbph  Verbph  "orbit something"),
 ("orbits",      Intransvb,meaning_of verbph  Verbph  "orbit something"),
 ("anyone",      Indefpron,meaning_of detph   Detph   "a person"),
```

```
  ("anything",    Indefpron,meaning_of detph   Detph   "a thing"),
  ("anybody",     Indefpron,meaning_of detph   Detph   "a person"),
  ("someone",     Indefpron,meaning_of detph   Detph   "a person"),
  ("something",   Indefpron,meaning_of detph   Detph   "a thing"),
  ("somebody",    Indefpron,meaning_of detph   Detph   "a person"),
  ("everyone",    Indefpron,meaning_of detph   Detph   "every person"),
  ("everything",  Indefpron,meaning_of detph   Detph   "every thing"),
  ("everybody",   Indefpron,meaning_of detph   Detph   "every person"),
  ("nobody",      Indefpron,meaning_of detph   Detph   "no person"),
  ("noone",       Indefpron,meaning_of detph   Detph   "no person")]
```

B.2 Executable Attribute Grammars

Next, we specify the syntax of the query language about the solar system with context-free rules. These syntax rules are accompanied by semantic rules to compute meanings of the phrases that are accepted by the corresponding syntax. Using the declarative notation we described in Chapters 3 and 4, we build this language description as an executable specification of an attribute grammar as follow:

```
snouncla
 = memoize Snouncla
 (parser
  (nt cnoun S3) [rule_s NOUNCLA_VAL OF LHS EQ
                copy [synthesized NOUNCLA_VAL OF S3]]
  <|>
  parser (nt adjs S1  *> nt cnoun S2)
  [rule_s NOUNCLA_VAL OF LHS EQ
       intrsct1 [synthesized ADJ_VAL      OF  S1
                ,synthesized NOUNCLA_VAL  OF  S2]]  )

-------------------------------------------------------------------
relnouncla
 = memoize Relnouncla
   (parser (nt snouncla S1  *> nt relpron S2  *> nt joinvbph S3)
```

```
    [rule_s NOUNCLA_VAL OF LHS EQ
      apply_middle1[synthesized NOUNCLA_VAL  OF S1
                   ,synthesized RELPRON_VAL  OF S2
                   ,synthesized VERBPH_VAL   OF S3]]
    <|>
    parser (nt snouncla S4)
    [rule_s NOUNCLA_VAL OF LHS EQ
      copy [synthesized NOUNCLA_VAL OF S4]])
----------------------------------------------------------------
nouncla
 = memoize Nouncla
   (parser (nt relnouncla S1 *> nt nounjoin S2 *> nt nouncla S3)
    [rule_s NOUNCLA_VAL OF LHS EQ
      apply_middle2 [synthesized NOUNCLA_VAL  OF S1
                    ,synthesized NOUNJOIN_VAL OF S2
                    ,synthesized NOUNCLA_VAL  OF S3]]
    <|>
    parser (nt relnouncla S1 *> nt relpron S2 *> nt linkingvb S3 *> nt nouncla S4)
    [rule_s NOUNCLA_VAL  OF LHS EQ
      apply_middle3 [synthesized NOUNCLA_VAL  OF S1
                    ,synthesized RELPRON_VAL  OF S2
                    ,synthesized NOUNCLA_VAL  OF S4]]
    <|>
    parser (nt relnouncla S1)
    [rule_s NOUNCLA_VAL  OF LHS EQ
      copy [synthesized NOUNCLA_VAL  OF S1]])
----------------------------------------------------------------
adjs
 = memoize Adjs
   (parser (nt adj S1 *> nt adjs S2)
    [rule_s ADJ_VAL  OF LHS EQ
      intrsct2 [synthesized ADJ_VAL  OF S1
               ,synthesized ADJ_VAL  OF S2]]
    <|>
    parser (nt adj S3)
    [rule_s ADJ_VAL  OF LHS EQ
```

```
      copy [synthesized ADJ_VAL  OF S3]])
-----------------------------------------------------------------
detph
 = memoize Detph
   (parser (nt indefpron S3)
    [rule_s TERMPH_VAL OF LHS EQ
      copy [synthesized TERMPH_VAL OF S3]]
    <|>
    parser (nt det S1 *> nt nouncla S2)
    [rule_s TERMPH_VAL OF LHS EQ
      applydet [synthesized DET_VAL      OF S1
               ,synthesized NOUNCLA_VAL  OF S2]])
-----------------------------------------------------------------
transvbph
 = memoize Transvbph
   (parser (nt transvb S1 *> nt jointermph S2)
    [rule_s VERBPH_VAL OF LHS EQ
      applytransvb [synthesized VERB_VAL    OF S1
                   ,synthesized TERMPH_VAL  OF S2]]
    <|>
    parser (nt linkingvb S1 *> nt transvb S2 *> nt prep S3 *> nt jointermph S4)
    [rule_s VERBPH_VAL  OF LHS EQ
      drop3rd [synthesized LINKINGVB_VAL  OF  S1
              ,synthesized VERB_VAL       OF  S2
              ,synthesized PREP_VAL       OF  S3
              ,synthesized TERMPH_VAL     OF  S4]])
-----------------------------------------------------------------
verbph
 = memoize Verbph
   (parser (nt transvbph S4)
    [rule_s VERBPH_VAL OF LHS EQ copy [synthesized VERBPH_VAL OF S4]]
   <|>
    parser (nt intransvb S5)
    [rule_s VERBPH_VAL OF LHS EQ copy [synthesized VERBPH_VAL OF S5],
     rule_s VAL  OF LHS EQ     copy [synthesized VAL  OF S5]]
   <|>
```

```
    parser (nt linkingvb S1 *> nt det S2 *> nt nouncla S3)
    [rule_s VERBPH_VAL OF LHS EQ applyvbph [synthesized NOUNCLA_VAL OF S3]])
-----------------------------------------------------------------
termph
 = memoize Termph
   (parser (nt pnoun S1)
   [rule_s TERMPH_VAL OF LHS EQ copy [synthesized TERMPH_VAL OF S1],
    rule_s VAL  OF LHS EQ copy [synthesized VAL  OF S1]]
   <|>
   parser (nt detph S2)
   [rule_s TERMPH_VAL OF LHS EQ copy [synthesized TERMPH_VAL OF S2]])
-----------------------------------------------------------------
jointermph
 = memoize Jointermph
   (parser (nt jointermph S1 *> nt termphjoin S2 *> nt jointermph S3)
    [rule_s TERMPH_VAL  OF LHS EQ
      appjoin1 [synthesized TERMPH_VAL     OF S1
               ,synthesized TERMPHJOIN_VAL OF S2
               ,synthesized TERMPH_VAL     OF S3]]
   <|>
    parser (nt termph S4)
    [rule_s TERMPH_VAL  OF LHS EQ copy [synthesized TERMPH_VAL  OF S4],
     rule_s NUMBA  OF LHS EQ copy [synthesized NUMBA  OF S4]])
-----------------------------------------------------------------
joinvbph
 = memoize Joinvbph
   (parser (nt verbph S1  *> nt verbphjoin S2  *> nt joinvbph S3)
   [rule_s VERBPH_VAL  OF LHS EQ
     appjoin2 [synthesized VERBPH_VAL    OF S1
              ,synthesized VBPHJOIN_VAL  OF S2
              ,synthesized VERBPH_VAL    OF S3]]
    <|>
    parser (nt verbph S4)
    [rule_s VERBPH_VAL  OF LHS EQ copy [synthesized VERBPH_VAL  OF S4],
     rule_s NUMBA  OF LHS EQ copy [synthesized NUMBA  OF S4]])
-----------------------------------------------------------------
```

```
sent
 = memoize Sent
   (parser (nt jointermph S1  *> nt joinvbph S2)
    [rule_s SENT_VAL OF LHS EQ
      apply_termphrase [synthesized TERMPH_VAL  OF  S1
                        ,synthesized VERBPH_VAL  OF  S2],
     rule_s VAL OF LHS  EQ
       nomatch [synthesized VAL  OF S1
                ,synthesized VAL  OF S2]])
two_sent
 = memoize Two_sent
   (parser (nt sent S1 *> nt sentjoin S2 *> nt sent S3)
    [rule_s SENT_VAL OF LHS EQ
      sent_val_comp [synthesized SENT_VAL      OF  S1
                     ,synthesized SENTJOIN_VAL  OF  S2
                     ,synthesized SENT_VAL      OF  S3]] )
-----------------------------------------------------------------
question
 = memoize Question
   (parser (nt quest1 S1  *> nt sent S2 )
    [rule_s QUEST_VAL  OF LHS EQ
      ans1 [synthesized QUEST1_VAL  OF  S1
            ,synthesized SENT_VAL    OF  S2]]
    <|>
    parser (nt quest2 S1 *> nt joinvbph S2)
    [rule_s QUEST_VAL  OF LHS EQ
      ans2 [synthesized QUEST2_VAL    OF  S1
            ,synthesized VERBPH_VAL    OF  S2]]
    <|>
    parser (nt quest3 S1 *> nt nouncla S2 *> nt joinvbph S3)
    [rule_s QUEST_VAL  OF LHS EQ
      ans3 [synthesized QUEST3_VAL  OF S1
            ,synthesized NOUNCLA_VAL OF S2
            ,synthesized VERBPH_VAL  OF  S3]]
    <|>
    parser (nt quest4 S1 *> nt nouncla S2 *> nt joinvbph S3)
```

```
    [rule_s QUEST_VAL  OF LHS EQ
      ans3 [synthesized QUEST3_VAL  OF  S1
           ,synthesized NOUNCLA_VAL OF  S2
           ,synthesized VERBPH_VAL  OF  S3]]
    <|>
    parser (nt two_sent S1)
    [rule_s QUEST_VAL OF LHS EQ
      truefalse [synthesized SENT_VAL OF  S1]]
    <|>
    parser (nt sent S1)
    [rule_s QUEST_VAL  OF LHS EQ
      truefalse [synthesized SENT_VAL OF  S1]])

quest4 = memoize Quest4
   (parser (nt quest4a S1 *> nt quest4b S2)
   [rule_s QUEST3_VAL  OF LHS EQ copy [synthesized QUEST3_VAL  OF  S1]] )
-----------------------------------------------------------------

query = memoize Query
        (parser (nt question S1)
        [rule_s QUEST_VAL  OF LHS EQ copy [synthesized QUEST_VAL  OF  S1]])
```

B.3 Functions for Attribute Evaluation

To compute meanings of larger phrases from the meaning of smaller phrases in the above AG's semantic rules, we define some application-specific operators and functions. For the current application, these functions are constructed following an efficient set-theoretic version of Montague's compositional semantics:

```
intrsct1         [x, y]
 = \atts -> NOUNCLA_VAL (intersect     (getAtts getAVALS atts x)
                                       (getAtts getAVALS atts y))
intrsct2         [x, y]
```

```
 = \atts -> ADJ_VAL (intersect         (getAtts getAVALS atts x)
                                       (getAtts getAVALS atts y))
applydet         [x, y]
 = \atts -> TERMPH_VAL                ((getAtts getDVAL atts x)
                                       (getAtts getAVALS atts y))
applytransvb     [x, y]
 = \atts -> VERBPH_VAL ((make_trans_vb (getAtts getBR atts x))
                                       (getAtts getTVAL atts y))

applyvbph        [z] = \atts -> VERBPH_VAL (getAtts getAVALS atts z)

appjoin1         [x, y, z]
 = \atts -> TERMPH_VAL                ((getAtts getTJVAL atts y)
                                       (getAtts getTVAL atts x)
                                       (getAtts getTVAL atts z))
appjoin2         [x, y, z]
 = \atts -> VERBPH_VAL                ((getAtts getVJVAL atts y)
                                       (getAtts getAVALS atts x)
                                       (getAtts getAVALS atts z))
apply_middle1    [x, y, z]
 = \atts -> NOUNCLA_VAL              ((getAtts getRELVAL atts y)
                                      (getAtts getAVALS atts x)
                                      (getAtts getAVALS atts z))
apply_middle2    [x, y, z]
 = \atts -> NOUNCLA_VAL             ((getAtts getNJVAL atts y)
                                     (getAtts getAVALS atts x)
                                     (getAtts getAVALS atts z))
apply_middle3    [x, y, z]
 = \atts -> NOUNCLA_VAL             ((getAtts getRELVAL atts y)
                                     (getAtts getAVALS atts x)
                                     (getAtts getAVALS atts z))
drop3rd          [w, x, y, z]
 = \atts -> VERBPH_VAL ((make_trans_vb (invert
                                     (getAtts getBR atts x)))
                                     (getAtts getTVAL atts z))
apply_termphrase [x, y]
```

```
 = \atts -> SENT_VAL                ((getAtts getTVAL atts x)
                                     (getAtts getAVALS atts y) )
sent_val_comp    [s1, f, s2]
 = \atts -> SENT_VAL                ((getAtts getSJVAL atts f)
                                     (getAtts getSV atts s1)
                                     (getAtts getSV atts s2))
ans1             [x, y]
 = \atts -> QUEST_VAL               ((getAtts getQU1VAL atts x)
                                      (getAtts getSV atts y) )
ans2             [x, y]
 = \atts -> QUEST_VAL               ((getAtts getQU2VAL atts x)
                                     (getAtts getAVALS atts y))
ans3             [x, y, z]
 = \atts -> QUEST_VAL               ((getAtts getQU3VAL atts x)
                                     (getAtts getAVALS atts y)
                                     (getAtts getAVALS atts z))
truefalse        [x]
 = \atts -> if (getAtts getSV atts x) then (QUEST_VAL "true.")
                                      else (QUEST_VAL "false.")

-- Functions used to define meanings of transitive verbs in terms of relations
make_trans_vb rel p = [x | (x, image_x) <- collect rel, p image_x]

trans_verb x = x
passtr_verb x = x

-- Functions used to define meanings of term phrase conjoiners
termph_and p q       = g where g x = (p x) && (q x)
termph_or  p q       = g where g x = (p x) || (q x)

-- Functions used to define sentence conjoiners
sand True True = True
sand any any'  = False

-- Functions denoted by determiners
function_denoted_by_a xs ys       = length( intersect xs ys ) > 0
```

```
function_denoted_by_every xs ys   = includes xs ys
function_denoted_by_none xs ys    = length( intersect xs ys ) == 0
function_denoted_by_one xs ys     = length( intersect xs ys ) == 1
function_denoted_by_two xs ys     = length( intersect xs ys ) == 2

-- Set and List operators
collect []          = []
collect ((x,y) : t) = (x, y:[e2 | (e1, e2) <- t, e1 == x]) :
                        collect [(e1, e2) | (e1, e2) <- t, e1 /= x]

invert           = map swap where swap (x, y) = (y, x)
includes  as bs  = (as \\ bs) == []
```

The semantic attributes or meanings, which can be computed using the above attribute grammar's semantic rules, are the type-definitions of semantic expressions, which propagate up or down during parser executions. For example:

```
data AttValue =
          NOUNCLA_VAL     {getAVALS   ::   ES}
      | VERBPH_VAL      {getAVALS   ::   ES}
      | ADJ_VAL         {getAVALS   ::   ES}
      | TERMPH_VAL      {getTVAL    ::   (ES -> Bool)}
      | DET_VAL         {getDVAL    ::   (ES -> ES -> Bool)}
      | VERB_VAL        {getBR      ::   Bin_Rel}
      | RELPRON_VAL     {getRELVAL  ::   (ES -> ES -> ES)}
      | NOUNJOIN_VAL    {getNJVAL   ::   (ES -> ES -> ES)}
      | VBPHJOIN_VAL    {getVJVAL   ::   (ES -> ES -> ES)}
      | TERMPHJOIN_VAL  {getTJVAL   ::   ((ES -> Bool)
                                      -> (ES -> Bool)
                                      -> (ES -> Bool))}
      | SENTJOIN_VAL    {getSJVAL   ::   (Bool -> Bool -> Bool)}
      | DOT_VAL         {getDOTVAL  ::   String}
      | QM_VAL          {getQMVAL   ::   String}
      | QUEST_VAL       {getQUVAL   ::   String}
          etc.
```

B.4 The Database

We use the following definitions of data (where the integers representing entities are linked to strings) for use in the set-theoretic compositional semantics in AGs:

```
-- THE UNIVERSE OF DISCOURSE
set_of_things      = [8..70]

-- SETS DENOTED BY COMMON NOUNS
set_of_sun        = [8]      set_of_planet     = [9..17]
set_of_moon       = [18..53] set_of_men        = [54..70]
set_of_woman      = []

-- SETS DENOTED BY ADJECTIVES
set_of_red         = [12, 13, 14, 22] set_of_blue    = [11, 14, 15, 16]
set_of_green       = [11, 15, 16]     set_of_brown   = [9, 10, 17]
set_of_ringed      = [13, 14, 15, 16] set_of_gaseous = [13, 14, 15, 16]
set_of_solid       = (union set_of_planet set_of_moon)
set_of_atmospheric = [ 10, 11, 12, 22, 42 ]
set_of_vacuumous   = (union set_of_planet set_of_moon)

-- SETS DENOTED BY INTRANSITIVE VERBS
set_of_spin      = [8..53]

-- BINARY RELATIONS
rel_orbit   =
  [(9,8),(10,8),(11,8),(12,8),(13,8), (14,8),(15,8),(16,8),(17,8),(18,11)
  ,(19,12),(20,12),(21,13),(22,13),(23,13),(24,13),(25,13),(26,13),(27,13)
  ,(28,13),(29,13),(30,13),(31,13),(32,13),(33,13),(34,13),(35,14),(36,14)
  ,(37,14),(38,14),(39,14),(40,14),(41,14),(42,14),(43,14),(44,14),(45,14)
  ,(46,15),(47,15),(48,15),(49,15),(50,15),(51,16),(52,16),(53,17)]

rel_discover =
  [(54,19),(54,20),(55,21),(56,22),(56,23),(56,24),(56,25),(57,26),(57,34)
  ,(58,27),(58,29),(59,28),(59,30),(59,31),(59,33),(60,32),(61,35),(62,35)
  ,(63,36),(64,37),(64,38),(64,49),(64,50),(65,39),(65,40),(65,41),(65,44)
```

```
 ,(66,42),(67,43),(68,45),(69,46),(69,52),(70,47),(70,48),(70,51)]

name_list =
 [("bernard",    55),("bond",       67), ("venus",      10),("cassini",    65)
 ,("dollfus",    63),("Fouuntain",  62), ("galileo",    56),("hall",       54)
 ,("herschel",   64),("huygens",    66), ("kowal",      57),("kuiper",     69)
 ,("larsen",     61),("lassell",    70), ("melotte",    60),("nicholson",  59)
 ,("perrine",    58),("pickering",  68), ("almathea",   21),("ariel",      47)
 ,("callisto",   25),("charon",     53), ("deimos",     20),("dione",      40)
 ,("earth",      11),("enceladus",  38), ("europa",     23),("ganymede",   24)
 ,("hyperion",   43),("iapetus",    44), ("io",         22),("janus",      36)
 ,("jupiter",    13)
 ,("jupitereighth",      32),("jupitereleventh",    31)
 ,("jupiterfourteenth",  34),("jupiterninth",       33)
 ,("jupiterseventh",     29),("jupitersixth",       27)
 ,("jupitertenth",       28),("jupiterthirteenth",  26)
 ,("jupitertwelfth",     30),("luna",               18)
 ,("mars",               12),("mercury",             9)
 ,("mimas",              37),("miranda",            46)
 ,("neptune",            16),("nereid",             52)
 ,("oberon",             50),("phobos",             19)
 ,("phoebe",             45),("pluto",              17)
 ,("rhea",               41),("saturn",             14)
 ,("saturnfirst",        35),("sol",                 8)
 ,("tethys",             39),("titan",              42)
 ,("titania",            49),("triton",             51)
 ,("umbriel",            48),("uranus",             15)
 ,("venus",              10) ]
```

B.5 Sample NL Input Query

The example NL query processor we described here is capable of answering hundreds of thousands of questions asked in natural-language. For example:

```
i1 = "which moons that were discovered by hall orbit mars"
```

```
i2 = "who discovered a moon that orbits mars"
i3 = "did hall discover every moon that orbits mars"
i4 = "how many moons were discovered by hall and kuiper"
i5 = "how many moons were discovered by hall or kuiper"
i6 = "every moon was discovered by a man"
i7 = "which planets are orbited by a moon that was discovered by galileo"
i8 = "which moons were discovered by nobody"
i9 = "is every planet orbited by a moon"
i10 = "which planets are orbited by two moons"
i11 = "who was the discoverer of phobos"
i12 = "hall discovered a moon that orbits mars"
i13 = "which moons that orbit mars were discovered by hall"
i14 = "every moon orbits every planet"
i15 = "every planet is orbited by a moon"
i16 = "a planet is orbited by a moon"
i17 = "does phobos orbit mars"
i18 = "did hall discover deimos or phobos and miranda"
i19 = "did hall discover deimos or phobos and miranda or deimos and deimos"
```

When these questions are applied as the input of the top-most definition of the attribute grammar, the language processor returns only the computed answers of the questions, instead of the complete parser trees. For example:

```
main  inputNo = meaning_of question Question inputNo

main i1  => [phobos deimos]
main i9  => [false]
main i5  => [4]
main i18 => [no, yes]
etc.
```

Appendix C

List of Refereed Papers Related to the Thesis

Below is a list of peer-reviewed papers that I have authored / co-authored, which are related to this thesis. Electronic versions of these papers can be found at `http://cs.uwindsor.ca/~hafiz/xsaiga/pub.html`.

1. Hafiz, R. and Frost, R. (2011) *Modular Natural Language Processing Using Declarative Attribute Grammars.* To be published in the Proceedings of the 10th Mexican International Conference on Artificial Intelligence, MICAI-2011, Puebla, Mexico 2011.

2. Hafiz, R. and Frost, R. (2011) *A System for Modularly Constructing Efficient Natural Language Processors.* To be published in the Proceedings of theComputational linguistics-Applications Conference, Poland, 2011.

3. Hafiz, R. and Frost, R. (2010) *Lazy Combinators for Executable Specifications of General Attribute Grammars.* Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN. January 2010, Madrid, Spain.

4. Hafiz, R. (2009) *Executable Specifications of Fully General Attribute Grammars with Ambiguity and Left-Recursion.* Proceedings of the 22nd Canadian Conference on AI 2009 Extended Abstract : 274-278

5. Frost, R., Hafiz, R. and Callaghan, P. (2008) *Parser Combinators for Ambiguous Left-Recursive Grammars.* Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN. January 2008, San Francisco, USA.

6. Frost, R., Hafiz, R. and Callaghan, P. (2007) *Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars.* Proceedings of the 10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE. Pages: 109 - 120, June 2007, Prague.

7. Frost, R. and Hafiz, R. (2006) *A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time.* ACM SIGPLAN Notices, Volume 41 Issue 5, Pages: 46 - 54.

Appendix D

Copyright Releases

1. **Chapter 2** includes a refereed paper, which was published as:

   Frost, R., Hafiz, R. and Callaghan, P. (2007) *Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars.* Proceedings of the 10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE. Pages: 109 - 120, June 2007, Prague.

   Dr. Alon Lavie (`alavie@cs.cmu.edu`), the chair of the IWPT ACL-SIGPARSE group, which maintains the publication of the paper, has personally sent the following e-mail to the authors regarding the copyright:

```
Dear Rahmatullah,

There should be absolutely no issue with including any of your IWPT
-published work within your dissertation, as long is there is
explicit citation of where this work was first published.

Best regards,
- *Alon*
```

2. **Chapter 3** includes a refereed paper, which was published as

   Frost, R., Hafiz, R. and Callaghan, P. (2008) *Parser Combinators for Ambiguous Left-Recursive Grammars.* Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN, Pages: 167-181. January 2008, San Francisco, USA.

   Below is the copyright release information from the Springer publisher:

   ```
   Dear Mr. Rahmatullah Hafiz,

   Thank you for placing your order through Copyright Clearance Center's
   RightsLink service. Springer has partnered with RightsLink to license
   its content. This notice is a confirmation that your order was
   successful.

   Your order details and publisher terms and conditions are available
   by clicking the link below:
   http://s100.copyright.com/CustomerAdmin/PLF.jsp?lID=2011091
   _1316638365433

   Order Details
   Licensee: Rahmatullah Hafiz   License Date: Sep 21, 2011
   License Number: 2753830069433 Publication: Springer eBook
   Title: Parser Combinators for Ambiguous Left-Recursive Grammars
   Type Of Use: Thesis/Dissertation
   ```

3. **Chapter 4** includes a refereed paper, which was published as

   Hafiz, R. and Frost, R. (2010) *Lazy Combinators for Executable Specifications of General Attribute Grammars.* Proceedings of the 12th International Symposium on

Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN, Pages : 167-182. January 2010, San Francisco, USA.

Below is the copyright release information from the Springer publisher:

```
Dear Mr. Rahmatullah Hafiz,

Thank you for placing your order through Copyright Clearance Center's
RightsLink service. Springer has partnered with RightsLink to license
its content. This notice is a confirmation that your order was
successful.

Your order details and publisher terms and conditions are available
by clicking the link below:
http://s100.copyright.com/CustomerAdmin/PLF.jsp?lID=2011091
_1316638075839

Order Details
Licensee: Rahmatullah Hafiz   License Date: Sep 21, 2011
License Number: 2753821291839 Publication: Springer eBook
Title: Lazy Combinators for Executable Specifications of General
       Attribute Grammars
Type Of Use: Thesis/Dissertation
```

4. **Chapter 5** includes a refereed paper, which has been accepted for publication as:

   Hafiz, R. and Frost, R. (2011) *A System for Modularly Constructing Efficient Natural Language Processors.* Computational linguistics Applications Conference, Jachranka, Poland, 2011.

   As this paper has recently been accepted for publication, and is yet to be published, the copyright release information is not available at the time when the thesis report is being prepared. Therefore at the time of the submission of this thesis, the copyright still belongs to the author and his supervisor, Dr. Richard A. Frost.

Vitae Auctoris

Rahmatullah Hafiz was born in 1979 in Dhaka, Bangladesh. He graduated from Government Laboratory High School, Dhaka, Bangladesh and Dhaka City College, Dhaka, Bangladesh in 1995 and 1997 respectively. He obtained B.Sc. (Honours) in Computer Science in 2004 and M.Sc. in Computer Science in 2006 at the University of Windsor. He is currently a candidate for the Doctoral degree in Computer Science.