1-1-2006

Efficient combinator parsing for natural-language.

Rahmatullah Hafiz
University of Windsor

Efficient Combinator Parsing for Natural-Language

by

Rahmatullah Hafiz

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Computer Science at the
University of Windsor

Windsor, Ontario, Canada

2006

Your file Votre référence
ISBN: 978-0-494-42328-8
Our file Notre référence
ISBN: 978-0-494-42328-8

ABSTRACT

Language-processors, that are constructed using top-down recursive-descent search with backtracking parsing technique, are highly modular, can handle ambiguity, and are easy to implement with clear and maintainable code. However, a widely-held, and incorrect, view is that top-down processors are inherently exponential for ambiguous grammars and cannot accommodate left-recursive productions. It has been known for many years that exponential complexity can be avoided by memoization, and that left- recursive productions can be accommodated through a variety of techniques. However, until now, memoization and techniques for handling left-recursion have either been presented independently, or else attempts at their integration have compromised modularity and clarity of the code – this leads to the fact that there exists no perfect environment for investigating many NLP-related theories. This thesis solves these shortcomings by proposing a new combinator-parsing algorithm, which is efficient, modular, accommodates all forms of CFG and represents all possible resulting parse-trees in a densely-compact format.

ACKNOWLEDGEMENT

I would like to express my gratefulness to my advisor Dr. Richard Alan Frost for his motivational and insightful suggestions, and patience. Dr. Frost's enthusiasm as a mentor and dedication as a researcher vastly inspired me to keep myself focused during my graduate studies.

I would like to thank Dr. Paul Callaghan of the University of Durham for his important remarks. I also wish to show my appreciation to my thesis committee members Dr. Jianguo Lu, Dr. Richard J. Caron and Dr. Scott Goodwin for their valuable comments regarding my thesis report and defense.

I am obliged to my wife, Sanjukta, for her continuous encouragement and kind support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: PREFACE

1.1 Introduction

Although the elegant top-down parsing method closely resembles a Natural Language Processing strategy, it has some short comings which make it less appealing for various NLP-related tasks. A naïve implementation may require exponential time and space and do not provide any support for ambiguous left-recursion (note that converting left-recursive grammar to non-left recursive form may cause missing parses). Modularity of implementation is required so that individual components of a language-processor can be tested separately and semantic-rules can be integrated naturally. As natural-language is ambiguous, it is important to ensure that the language-processor is able to process ambiguous left-recursive grammars in order to have the proper right-most and left-most derivations, which is essential to retrieve all possible semantic meanings. Also the computation-time needs to be reasonable, and the exponential number of parse-trees should be represented within polynomial space. Many attempts have been made to accomplish the above requirements but none has been able to accommodate all of them within one algorithm.

In this thesis we develop a general top-down combinator-parsing algorithm that accommodates ambiguous and left-recursive grammars, whilst maintaining polynomial time-complexity, compact (polynomial) representation of exponential number of parse trees and modularity of the implementation. We implement the algorithm in a lazy functional language, Haskell, have analyzed it theoretically and have tested it with highly ambiguous grammars to support the theoretical claims.

1.2 Structure of the Thesis Report

Chapter 2 introduces the basics of top-down parsing, the problem, the requirements and how we will prove the thesis-statement. Chapter 3 and 4 briefly describes different aspects of lazy-functional programming and combinator-parsing. Chapter 5 mentions some related and motivational previous-work. Chapter 6 and 7 describe the new algorithm in detail - for recognition and parsing respectively. Chapter 8 explains the Haskell-implementation of the algorithm. Chapter 9 and 10 analyze the termination and

complexity of the algorithm respectively. Chapter 11 presents some experimental results, which support the analytic results of chapter 10. Chapter 12 concludes the report and the appendix contains some example-output (densely packed parse-forests).

1.3 Contribution by the Candidate

In conducting the work described in this thesis-report, the candidate worked closely with Dr. Frost, his supervisor, and he also collaborated with Dr. Callaghan of the University of Durham.

The candidate and Dr. Frost jointly developed the algorithm to accommodate left-recursion with top-down parsing in polynomial time and space. The candidate was primarily responsible for implementing the algorithm in Haskell, with some suggestions from Dr. Frost and Dr. Callaghan. The candidate was responsible for conducting the experiments. The candidate also helped Dr. Frost to construct the proof of termination and complexity. The results of the collaborative work have been published in two papers co-authored by the candidate (Frost and Hafiz, 2006, [11]) and (Frost, Hafiz and Callaghan, 2006, [12]).

CHAPTER 2: INTRODUCTION

2.1 Grammars and Top-Down Parsing

A 'language' is a set of finite-length sequences or strings, constructed over a finite-set of entities known as alphabet. Any formal or natural-language can be specified or defined with a finite-size specification (or generator) – formally known as 'grammar'. A formal grammar G is a 4-tuple (N, Σ, P, S) where:

- N is a finite-set of non-terminals
- Σ is a finite-set of terminals (or alphabet – over which a language is defined)
- P is a finite-set of production-rules
- S is a distinguished symbol (known as *start* symbol)
- $N \cap \Sigma = \emptyset, S \in N$ and

  $(\forall p_i \in P)(p_i \in (N \cup \Sigma)^* N(N \cup \Sigma)^* \times (N \cup \Sigma)^*)$

The language $L(G)$, which is defined by G, is a set of 'strings' that consist only of the terminals from Σ and that can be derived starting from S and applying $\exists p_i$ until no non-terminal is present. This formal-grammar framework is the most expressive way to specify a language. According to Chomsky (1956, [4]), this general framework of formal or *phrase-structure* grammar can be divided into four 'types':

Grammar Types	Properties
Type 0- Unrestricted	▪ Rules don't have restrictions (in terms of number of symbols) on left and right side of the productions. ▪ Its most 'unrestricted' nature makes it less useful to linguists.
Type 1 – Context-Sensitive	▪ Rules are of form $\alpha A \beta \rightarrow \alpha \chi \beta$ where $A \in N$, $\alpha \& \beta \in (N \cup \Sigma)^*$ and $\chi \in (N \cup \Sigma)^+$. ▪ The derivation $A \rightarrow \chi$ is determined by the "context" of α & β.
Type 2 – Context-Free	▪ Rules are of form $A \rightarrow \alpha$ where $A \in N$, $\alpha \in (N \cup \Sigma)^*$. ▪ Simple and 'powerful' enough to define most of the languages.
Type 3 – Regular	▪ Rules are of form $A \rightarrow \alpha B$ or $A \rightarrow \alpha$ where $A \& B \in N$ and $\alpha \in \Sigma$. ▪ Equivalent to 'regular-expressions' and the most restricted grammar.

Table 2.1: Four types of grammars, where type 0 ⊃ type 1 ⊃ type 2 ⊃ type 3

Out of the four types, the Context-Free Grammar (CFG)[1], typically expressed in Backus-Naur Form (BNF), is the most important type in terms of application to various languages. The languages that can be specified by CFGs are known as Context-Free Languages. CFGs are considered as 'standard sets of rules' for syntax-analysis or **parsing**, which is a technique to determine whether a given input sequence's grammatical structure can be recognized and identified by the given CFG. The most natural form of parsing is **Top-Down Parsing**, which is a method of attempting to find the 'left-most derivation' of a given input sequence. The 'attempt' starts from the root-symbol `S` and keeps expanding from the left-most position of `S`'s definition. The **recursive-decent fully backtracking** parsing is the most general form of top-down parsing, where rules are implemented as 'mutually-recursive' procedures and if an alternative of a rule 'fails' or 'ends', the parser backtracks to try another rule. Top-down parsers are easy to construct and understand, compared to their bottom-up counter-parts.

An Example

In order to specify or generate a language `L(G_1) = {0, 1, 00, 01, 10, 11, 000, 001, 010, 100}`, we may use the following CFG `G_1`:

G_1 =

(N = {B_E, D},

Σ = {0, 1},

P =

$$\left\{ \begin{array}{ll} \text{B\_E} & ::= \text{D} \mid \text{D B\_E} \mid \varepsilon \\ \text{D} & ::= 0 \mid 1 \end{array} \right\},$$

S = B_E) (B_E = Binary Expression, D = Digit)

While determining the syntax-structure of an input "001" using **top-down parsing** technique with `G_1`, a parser executes the start-symbol or the *root* rule `B_E`, which has three alternatives. Each of the alternatives is individually (in a sequence from left) applied on the original-input. If the alternatives have non-terminals, then they are

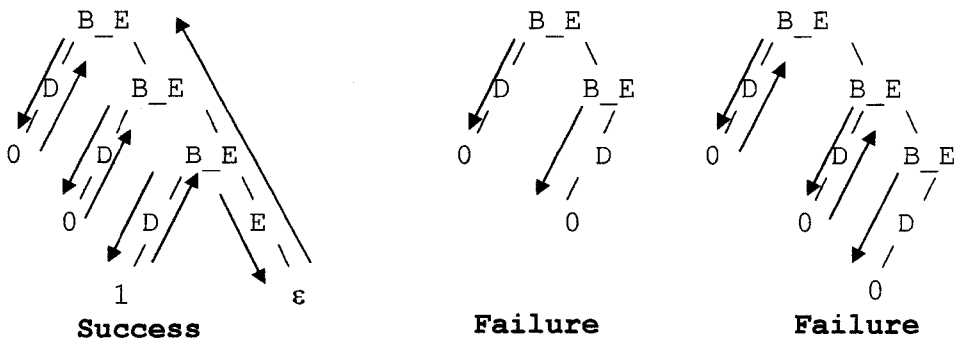
[1] In this report, we represent a production-rule of a CFG as
Non-terminal ::= ….. Non-terminal' ….*terminal*…|….. Non-terminal'' ….*terminal'*…
Where '::=' means the starting of a rule-definition, '|' separates alternatives, *terminals* are in *italic*, Non-terminals start with capital-letter and sequencing-symbols are written next to each other.

expanded again recursively and their definitions (which may consist of different alternatives) are applied to the remaining input-sequence (if some left-most tokens were previously consumed). The process ends if there are no more input-tokens left for processing. For example:

1. B_E is expanded to its first alternative B_E: := D and then D is expanded to its two alternatives D: := 0 and D: := 1. The second alternative of D is a 'failure' as it derives to '1' whereas the input sequence starts with '0'.

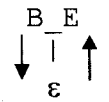
The first alternative finds a match for the leftmost position of "001" and as there are no other non-terminals left for expanding, it is a valid parse.

2. When the previous phase is done, B_E 'backtracks' to try its second alternative B_E: := D B_E, then D's two alternatives are applied on "001" (as we know, only the first one succeeds and recognizes "0") and the rest of the input "01" is processed by the non-terminal B_E (which is a part of root B_E's recursive definition). In a similar way, B_E is expanded again to its three alternatives and eventually returns a successful parse-tree and also reports all failed attempts:



(Note that the arrows are indicating the control-flow of the parsing-process.)

3. When the above phase is done, the first application of `B_E` 'backtracks' to try its third alternative `B_E::= ε,` which always succeeds:



Success

In terms of functional-programming, **combinators** or higher-order functions are ideally-suited for constructing top-down recursive-descent (with backtracking) parsers. The combinators are 'operators' which are used to construct basic parsers from terminals, and compound parsers from simpler parsers. Before examining any input-token, a combinator-parser tries to execute a 'rule' – an executable-specification, to identify the token. If this attempt fails, the parser recursively tries another rule and so on. Language-processors, constructed using this parsing-technique, are able to provide many advantages such as:

1. They are easy to implement in most of the programming languages that support 'recursion'.
2. Associating semantic rules for recursive syntax-rules is straightforward (Frost [8]).
3. They are highly modular (Koskomies [22]), re-useable and each components can be tested individually.
4. The structure of the code is closely related to the structure of the grammar of the language to be processed and can be implemented as executable-specifications of grammars, as shown by Frost and Launchbury (1989, [13]). Definite Clause Grammars (DCGs) of logic-programming can also be used to achieve this.

A simplistic implementation of a top-down combinator-parser normally requires exponential computation time, may sacrifice 'ambiguity' and it is not capable of handling grammars having left-recursive production-rules, such as `S::=S a | a`. Despite having many advantages, these drawbacks of a general top-down parser turn it to a less-attractive choice for practical uses. Consequently, all of the benefits of top-down parsing have not been available to researchers working on natural-language processing.

2.2 Natural-Language Parsing

2.2.1 Ambiguity

As **syntactic-ambiguity** is a property of 'human-spoken' languages, a grammar that defines a natural-language is naturally ambiguous. It is important that any parsing system, which parses a natural-language, is able to identify all possible derivations or 'parse-trees' for a given sentence. For example, the following grammar (similar to Charniak, 1991) is ambiguous and a parsing-system that attempts to generate the syntactic structure of the sentence "John sells the dog food" according to this CFG, should identify more then one parse-tree:

```
Sent  ::= NP VP
VP    ::= Verb NP | Verb NP NP
NP    ::= Del Noun | Noun | Del Noun Noun | NP NP
Noun  ::= food | dog | cat | John | Liz
Det   ::= the | a | an
Verb  ::= sells | buys | plays
```

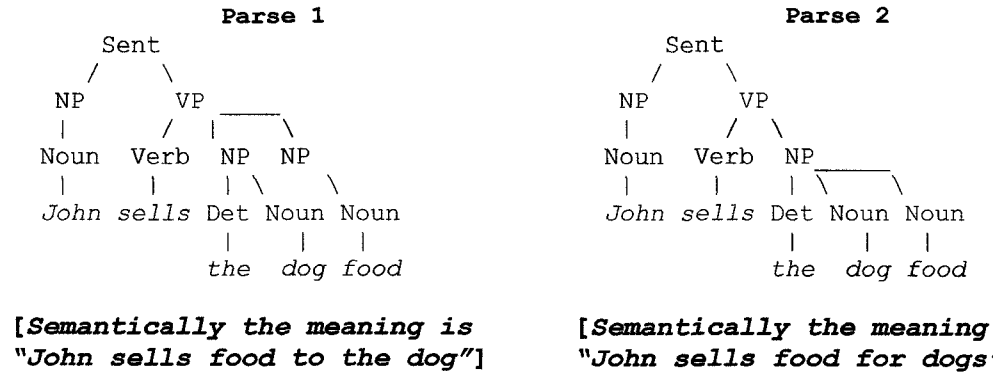


Figure 2.1: An ambiguous grammar and some possible parses

2.2.2 Left-Recursion

There are several reasons why it is important for an NL-parsing system to accommodate left-recursive CFGs:

1. If the parsing system can not process any grammar-rule written in left-recursive form 'directly', then the syntax-structure of the derivation changes and as a result, causes semantic misinterpretation. The text-book solution (Aho, Shethi and Ullman, 1986) for

the left-recursion problem of top-town recursive-descent parsing is 'eliminating a left-recursive production-rule by converting it into a non-left-recursive one'. The following example demonstrates missing valid-parses and semantic misinterpretations because of this elimination technique:

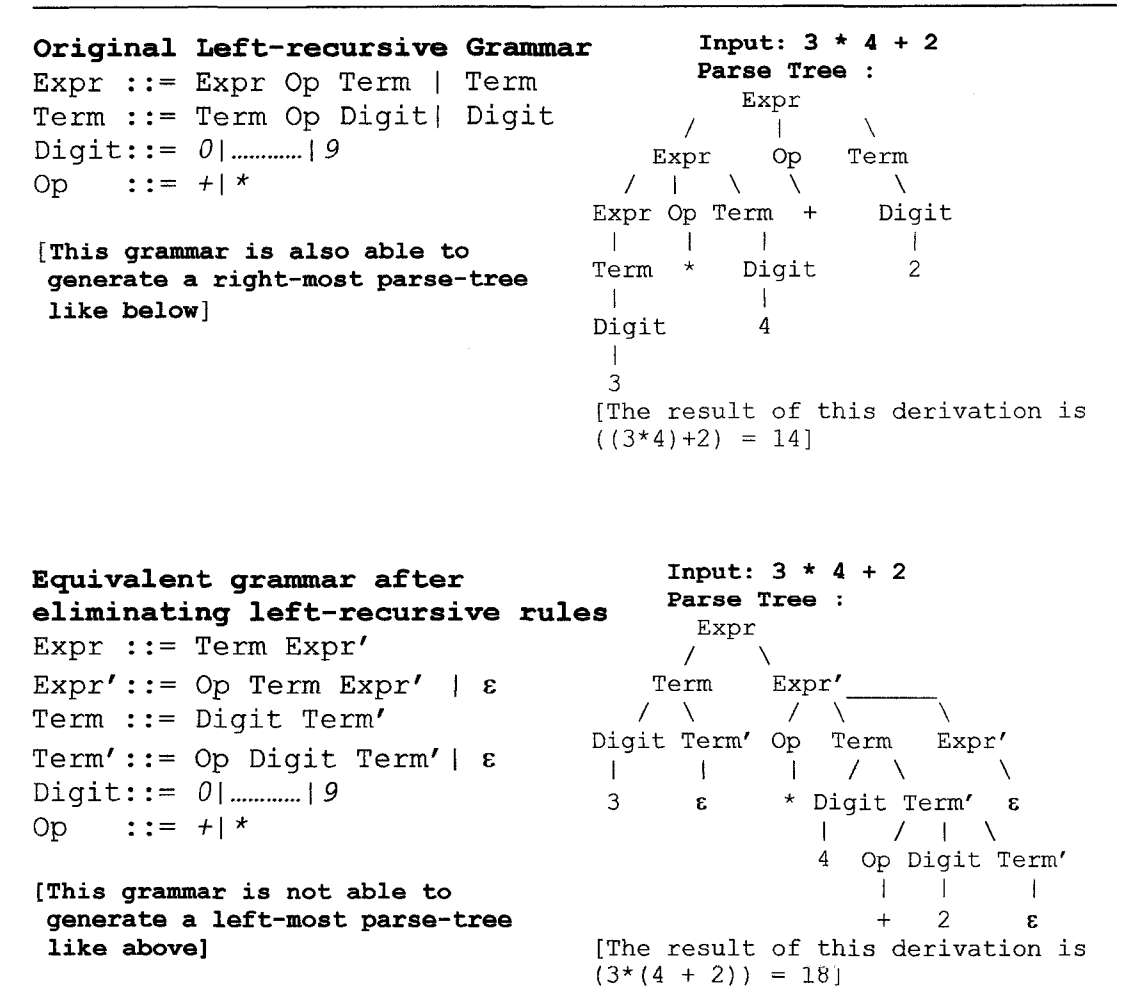


Figure 2.2: Elimination of left-recursion and possible problem

2. If the production rules of the grammar are in left-recursive form, sometimes it is straight-forward and easier to add semantic-meanings or *attributes* to a grammar – that is used in a language-processor. For example, attributes can be easily added to the following left-recursive grammar, where as, there is no such simple way to add attributes to an equivalent right-recursive grammar (**bold** fonts are attributes):

```
Left-recursive attribute-grammar
number   ::= digit                                    digit   ::= '0'
number.VAL = digit.VAL                                digit.VAL = 0
           | number' digit                                      | '1'
number.VAL = (10 * number'.VAL + digit.VAL)           digit.VAL = 1

Equivalent non-left recursive attribute-grammar
number   ::= digit
number.VAL = digit.VAL
           | digit number'
number.VAL = ?????
```

3. As mentioned in [11], if left-recursive grammars could be used with top-down parsing, they would provide a better framework for investigating NL theories in order to achieve more efficient natural-language interfaces. For example, to test and investigate compositional Montague-like theories (for processing verb adjuncts such as "When and with what did Hall discover Phobos?") the parsing-system needs to achieve all possible ambiguous leftmost and rightmost derivations.

2.3 The Problem

To utilize the advantages of a 'recursive-descent backtracking' top-down combinatory parser for natural-language parsing (and for other applications involving ambiguity), we require a new algorithm to accommodate ambiguity and left-recursive production rules in polynomial time. Many researchers have tried to address this problem with different approaches, but none of them completely satisfies the requirements.

It is widely believed 1) that top-down parsers require exponential computational-time, 2) that they fall into infinite-loop while processing left-recursive grammars, and 3) that it is not practical to implement a modular top-down parser. However, the following list of researchers' work demonstrates that the above problem can be partially addressed:

1. Norvig (1991, [32]) showed that it is possible to achieve polynomial complexity for top-down recognizers by use of the memoization technique.
2. Shiel [34] and Kuno's [23] algorithms, though terminating for left-recursion by utilizing the length of the input string, are based on the similarities between chart-parsing and top-down parsing, but have exponential complexity.

3. Leermakers' [24] claim to solve the problem using a "recursive ascent" functional approach appears to be achieved compromising modularity and clarity of the code, and is not really a top-down approach.
4. Frost's [7] "guarded-attribute" approach solves the left-recursive problem, but exhibits exponential time at worst case.
5. Nederhof and Koster's approach of "cancellation parsing" [30] to process left-recursive rules is exponential at the worst-case and the resulting code is less clear as it contains additional production rules and code to insert the special tokens.
6. Lickman's use of purely functional set-monadic fix-point parser-combinator approach to accommodate left-recursion for recognition is exponential [27].
7. Johnson integrated memoization with continuous-passing style-programming [20] to resolve the problem. It appears that this approach is too complicated for practical use. Also, as pointed by the Johnson, this approach might be too difficult to modify for compact-representation of resulting parse-trees.
8. Camarao, Figueiredo, and Oliveiro's [3] monadic compiler-generator may accommodate left-recursion but fails to accommodate ambiguity.

Even though the above approaches partially solve the well-known drawbacks of top-down parsing, for last 40 years no one has been able to provide full support for all of the following requirements within a single parsing-system:

1. Complete support for direct and indirect left-recursive grammar.
2. Accommodating ambiguity.
3. Accommodating any form of CFG (including empty, cyclic, 'densely' rules etc).
4. Maintaining at least polynomial time and space complexity at the worst case.
5. Clarity and modularity of the implementation for efficient practical use.
6. Not only recognition but also working as a complete parser – that is able to represents the resulting parse-trees using least possible space.

This thesis fulfills the requirements above.

2.4 Thesis Statement

2.4.1 The Statement

"Top-down parsing can accommodate ambiguity and left-recursion and can create a compact (polynomial-size) representation of parse trees in polynomial time at worst case."

2.4.2 Why This Thesis is Important

Much work has been done and many theories have been proposed to analyze, investigate and compute different aspects and problems related to natural-language semantics. But all of the existing syntax-analysis systems share some common shortcomings (as mentioned in section 2.3); it is not completely possible to accommodate and easily investigate all semantic-analysis theories within current platforms. The algorithm described in this thesis will allow combinator-parsing to be used with any form of ambiguous left-recursive CFGs whilst maintaining polynomial time and space complexity. Hence, this work will enable the full potential of existing work on natural-language semantics to be integrated with and investigated within syntactic-analysis, thereby providing a useful environment for natural-language investigation. This work also allows constructing natural-language processors as executable specifications by being highly modular, structured and easily alterable. According to the proposed algorithm, the result of parsing is represented as a densely-compact form, which will help the potential users to retrieve a particular parse-tree by spending less time. Overall, investigating and implementing different theories and aspects of computational-linguistics will benefit significantly.

2.4.3 Why it is Not Obvious

As top-down parsing attempts to find the left-most derivation for a given input sequence, it has been assumed that its 'recursive-descent' phase would never terminate while processing a CFG that contains left-recursive production rule(s). As 'backtracking' is important to achieve the desired ambiguity (especially for natural-language parsing) and requires sophisticated care during implementation, an unstructured and naïve implementation may result loss of possible parses. Also, even if the CFG doesn't have

any left-recursive production rules, top-down parsing is generally considered to exhibit exponential time-complexity in the worst-case – mainly because of its 'backtracking' characteristic. Reviews from literature suggest that it's been universally assumed in the functional-programming community that combinator-parsing cannot accommodate left-recursion at all - because the left-most 'parser' would always keep executing its own definition again and again. Therefore, despite fulfilling some partial requirements (as mentioned in section 2.3), for last 40 years no one has been able to accommodate ambiguity and left-recursion within a complete top-down parsing system in polynomial time.

2.4.4 How the Thesis will be Proven

The following steps have been taken to justify that the proposed algorithm satisfies the thesis-statement:

- Studying related-works thoroughly to identify whether they have addressed the following all requirements or not:
    1. Complete support for direct and indirect left-recursive grammar.
    2. Accommodating ambiguity.
    3. Accommodating any form of CFG (including empty, cyclic, 'densely' rules etc).
    4. Maintaining at least polynomial time and complexity at the worst case.
    5. Not only recognition but also working as a complete parser – that is able to represent the resulting parse-trees using least possible space.
    6. Maintaining modularity, clarity and flexibility to accommodate different theories and applications for NLP including integrating semantic-rules with syntax-structure correctly.
- Proposing a new algorithm to fulfill the above requirements.
- Implementing the algorithm in a lazy-functional language – Haskell.
- Termination-analysis of the algorithm.
- Complexity-analysis – to justify the claim of polynomial time and space complexity.
- Conducting experiments to test the practicality of the algorithm.

2.5 Brief Description of the Solution

Frost and Szydlowski [14] proposed a framework to the utilize memoization technique for improving the complexity of purely-functional parser-combinators. Later, in 2003 [9], Frost showed how recognition of natural-languages can be achieved in polynomial-time by memoizing parser-combinators in a systematic way using 'state- monads'. But the basic drawback of this approach is that it doesn't 'terminate' while processing left-recursive grammars. The proposed algorithm of this thesis uses memoization in such a way that it can accommodate indirect and direct left-recursive grammars. In order to do so, the new algorithm keeps track of the depth of a particular parse and the length of the input which is currently being processed. Each time a particular parser is being called during recursive-descent while processing a particular input, a 'counter' (we call it 'left-rec-counter') is incremented by one – indicating the depth of the parser. This parser is 'curtailed' when its left-rec-counter exceeds the length of the remaining input and the process backtracks up the parse-tree to apply another alternative parse, if exists any. When the parser computes a result, it saves it to a 'memo-table' along with a reference to the position in current input.

Though this treatment solves the 'direct left-recursive' problem, the indirect or hidden left-recursive productions may still skip some valid 'parses'. This is mainly because when an intermediate parser (of an indirect left-recursive parser) tries to look-up a previously stored result in the 'memo-table', it may retrieve only a partial result which is less then if the parser were applied again in the new context. The solution to this is to provide 'context-based' update and lookup for memoization. Now, when a parser goes down during recursive-descent, it keeps records of all others parses on its way and their 'left-rec-count' along with its own. If any parser is 'curtailed' at a particular position (i.e. if a parser is left-recursive), it passes its 'reference' upward during recursive-ascent. When a parser computes a result, it saves the result to 'memo-table' along with its current-context w.r.t. the reference of the 'curtailed' parser(s), if any. Subsequently, while performing a lookup, the current parser judges whether to reuse a saved result or not by comparing the 'saved-context' with its 'current-context'. If it finds that it has recursively descended enough, then it is eligible to re-use, otherwise it has to perform more recursive-descent operations.

The new algorithm stores the resulting parse-trees in the memo-table as a forest of one-level depth, n-ary branches. Each branch has implicit pointers to determine 'where to go next' in its nodes and is shared between ambiguous parses. This densely-compact representation of resulting parse-trees ensures the cubic space-complexity, even though the total number of parses could be exponential.

The detailed description of the algorithm is given in chapter 6 and 7. Section 11 contains experimental results (based on different grammars of the appendix with varying number of inputs). The implementation is described in section 8 and theoretical analysis is given sections 9 and 10.

CHAPTER 3: LAZY FUNCTIONAL PROGRAMMING

3.1 Introduction to Lazy Functional Languages

A functional-program, in general, consists of a set of function definitions – which follow regular mathematical properties. Execution of a conventional program, written in a conventional language, is based on processing a set of 'hidden stores of named locations' by sequences of assignment statements, whereas execution of a functional program is based on computation of functions and application of them to data. Frost [10] has defined pure functional programming as programming in an environment where "function composition and function application are the only forms of control structure" and any form of looping and iteration must be performed through recursive function calls. In the purest form of functional programming (known as lazy functional-programming, LFP), the languages are polymorphically typed and embedded with automatic type checkers, "the evaluation of arguments to functions is delayed until those values are required" [10]. Assignment statements are not allowed in functional programs. So 'variables' don't change their values during the program-life. Hence, nothing changes the value of an expression and function calls don't have any other effects other than executing themselves. In other words, lazy languages don't have any "side-effects". As a result, the order of execution of any function is not important (i.e. functional programs have no flow-control). The properties above reduce possible causes of errors in programs and program-executions. As functional programs exhibit 'natural-parallelism' more than conventional programs, they are well suited for the latest computer-architecture where different processes operate simultaneously while communicating and cooperating with each other. Well-structured and modularity (achieved through 'higher-order functions' and laziness) make a functional-language more efficient than conventional languages. Commonly-used LFP languages are Haskell, Miranda and SML. Lisp and Scheme represent strict-version of FL. Hughes's paper [16] answers the question "Why Functional Programming Matters?" in detail.

3.2 Elements of Lazy Functional-Programming

Important components and attributes of LFP are described below briefly:

1. Lambda Calculus

Church invented lambda-calculus in 1930 to demonstrate that it is impossible to find a general algorithm which, for some given first-order statements, decides whether they are universally valid or not. Since then, it has been used for investigating function-definition, function-application, recursion and has influenced the basic implementation-mechanism of LFP languages. The basic building block of lambda calculus is computable-function formation (which is obtained by *abstracting* an expression), and *variable* substitution. A simple lambda-expression or term is, for example $\lambda x.(x^3 + 1)$, where λx abstracts the name-less expression $(x^3 + 1)$ w.r.t. x. We evaluate this λ-term by substituting the variable x with a given constant. For example, in $(\lambda x.(x^3 + 1))2 \equiv 2^3 + 1 \equiv 9$ the variable x is replaced by the constant 2. A variable x is 'bound' if its occurrence in the λ-term is preceded by λx, otherwise x is 'free'. For example, in the expression `λx.(x + y)`, x is bound and y if free. λ-calculus is considered as the 'universal programming language'. As any computable function can be expressed and evaluated using λ-calculus, it is the central issue of the LFP paradigm.

2. Higher-Order Function

A 'higher-order function' (HOF) is a function which can take other function(s) as its input-argument and also can produce some other function(s) as its output. For example, most LFP languages provide a HOF `map` that receives a function and a list as its input parameter and returns a list by applying the input function to each element of the input list, for example, `map (*2) [1,2,3]` ⇒ `[2,4,6]`. HOFs are 'first-class object' of LFP language. Use of higher-order functions as infix-operators is the basic for constructing parser-combinators as such use of functions can mimic the BNF notation of a CFG. For example, a CFG rule `'s::= a s | empty'` can be interpreted in English as

"`s is either 'a then s' or empty`"

Using higher-order functions (`or` and `then`), parser-combinators[1] can be written as:

```
empty input          = [input]
a (x:xs)             = if x =='a' then [xs] else []

(p `or` q ) input    = p input ++ q input
(p `then` q) input   = if r == [] then []
                       else map q r
                            where r = p input
```

and used, as for example:

```
s input = (a `then` s `or` empty) input
```

3. Pattern-matching

Pattern-matching enriches function-definitions' readability and the structure of the program a great deal. Standard patterns (such as variables, constants, wildcard-pattern, patterns for tuples, lists, algebraic constructors etc) match against the syntactic-structure of an argument while maintaining the 'lazy-evaluation' attribute. LFP supports another type of pattern – known as 'application pattern' or 'n+k-pattern', which matches the semantic-structure of the arguments instead of syntactic-structure. As described in [33], a pattern n+k, where k is a constant, matches against an actual function-argument - a, if a can be considered as the result of an expression (λn.n+k) b. If so, then n is bound to b. Clearly, b can be calculated by evaluating the inverse expression (λn.n-k) a.

4. Polymorphic Types

LFP languages are 'strongly-typed' – which prevents users to use ill-typed values in function application, equipped with 'statically type-checking' system – that tries to detect types automatically, checks for type-mismatch and identifies type-errors during compilation. Also, in LFP, functions' input and output arguments may have 'polymorphic-type' – that means these function-definitions are common for any type of values. This facility makes function-definitions more general and reusable. For example, a simple function 'tail' – which is defined to pick the last element of a list - works on lists of integers, strings etc:

```
tail [1,2,3] ⇒ 3,
tail ["aa","bb","cc"] ⇒ "cc".
```

[1] Explained in section 3.3

5. Currying

Frege, in 1893, observed that it is sufficient to define any function with a single argument. "Currying" (named after Haskell B. Curry) is a way that converts an n-argument function to a 1-argument function and returns another function (if more arguments are needed). Modern LFP languages use "currying" (and "uncurrying") as a default method of evaluation (by treating all functions as higher-order functions) and provide abstraction to user through 'syntactic-sugar'. For example, the function `add` (of type `add :: Int → Int → Int`) adds two integers. When '`add 2 3`' is executed, first '`add 2`' is evaluated and returns a function (of type `Int → Int`). Then this function is applied to 3 and returns 5 as the final result.

6. Lazy-evaluation

Lazy-evaluation delays the computation of a function until the result is required. This unique feature allows LFP languages to process 'infinite data-structure'. For example, assume already defined `pick_5th` function selects 5^{th} element of a list. When this function is applied on an infinite list of integers, it doesn't go into non-terminating state but retunes the fifth element from the list. For example, `pick_5th [1..]` ⇒ `5`

7. Monads

Monadic-computation is another unique feature of LFP languages. The underlying idea is derived from categorical-theory. Using monads, a computation could be constructed as sequential block of computations and the 'block of computations' may be constructed using other sequential block of computations too. Monads make programs much structural and modular by ensuring sequential execution of computations. A detailed description of monadic-computation is described in section 4.2.

3.3 Haskell – A Purely-functional Language

Haskell (Hudak, 2000) is a purely-functional, lazy, polymorphically typed, widely used programming language. The latest version *Haskell 98* is the most stable implementation and enriched with an expressive syntax, user-defined algebraic data-types and standard libraries with a wide range of built-in primitive data-types, functions and type-classes. Haskell 98's features and functionalities are documented in "The Haskell 98 Report"

(Peyton-Jones, 2002). In addition to supporting all of the features mentioned in previous section, it also provides a novel type-system that supports a systematic form of 'overloading'. Different variations of Haskell are also introduce such as: GPH, pH (both are parallelizable version of Haskell), Haskell++, O'Haskell ('object-oriented' Haskell), Mondrian (a 'mixture' of Haskell and Java – can be used in .NET platform) etc. Haskell has been used frequently in academia and in industry simultaneously. Many NLP-relates systems have been implemented using Haskell. Following table summarizes Haskell's use in NLP:

System	Implementer and Year of Implementation	Purpose
LOLITA	Garigliano, R., Natural Language Engineering Group, University Of Durham, 1989	• Natural language processing • Content scanning • Implementing plausible reasoning model • Chinese language tutoring • Connexion to speech input and output • Natural language generation system for English and Spanish • Discourse planner • Information extraction system for equity-derivation trading
Grammatical Framework (GF)	Ranta, A. 1998	• Multilingual authoring/ Multilingual Syntax Editing • Proof text editor • Software specifications • Controlled language • Dialog system • Technical document editor
Ontology Construction	Khun, W., 2001	• Building ontologies for natural language text to describe human activities
Functional Morphology	Ranta, A. and Forsberg, M., 2004	• Constructing morphologies of Swedish, Italian, Russian, Spanish, and Latin
Combinator Parsing for NL	Frost, R and Hafiz, R, 2006	• Accommodating ambiguity and left-recursion in polynomial time • Compact-Representation of resulting parses

Table 3.1: Use of Haskell for constructing NLP-related systems

As the new algorithm - proposed and documented in this thesis-report - is implemented in Haskell, knowledge of elementary Haskell notations [5, 16] would be useful for the better understanding of the rest of the report.

CHAPTER 4: COMBINATORY PARSING

4.1 General Concept

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 these combinators, is called 'combinatory-parsing' (as higher-order functions 'combine' different parsers together). A complete language-processor can be constructed by combining small processors with combinators. Though the concept of combinatory-parsing was introduced by Burge in 1975 [2], it was Wadler (1985, [37]) 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'. Most of time, 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. Combinatory parsers are written and used within the same programming language as the rest of the program. As function application in LFP is juxtaposition, a language-processor written using combinators can represent BNF representation of any CFG. 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. Semantic-meaning and extra functionalities can be added to the respective production-rules effortlessly. Frost and Launchbury (1989, [13]) showed how to construct Natural-Language Interpreters in Miranda[1] using higher-order functions. Based on this work, Frost later constructed an attribute grammar-programming environment – **W/AGE** (Windsor attribute grammar-programming environment) (2002, [10]). Huttton (1992, [19]) also used parser-combinators to demonstrate a complete parser construction - that addresses parsing problems caused by white-space, special characters etc. Koopman and Plasmeijer (1999, [21]) used continuation to improve the efficiency and performance of parser-combinators.

[1] Miranda is a trademark of Research Software Limited of Europe

The following step-by-step example demonstrates how a CFG can be represented as a language-processor using parser-combinators.

4.1.1 Example of a Simple Parser-Combinator

A production-rule of a CFG may have one or more 'alternatives' and each alternative may consist of a sequence of non-terminal(s) and/or terminal(s), or the alternative may consist of a single non-terminal or terminal or 'empty'. In order to build a recognizer for a CFG using parser-combinators, we need to construct some basic combinators and use them to 'glue' different terminals and non-terminals to form a complete rule. These combinators work as infix operators and non-terminals (and terminal) work as operands to these operators. (Note that in this example we just work with 'recognizers' instead of 'parsers') Consider a CGF that generates a limited subset of natural-language:

```
Sentence    ::= Noun_Phrase Verb_Phrase
Noun_Phrase ::= Del Noun | Adjective Noun | Del
                Noun Verb
Verb_Phrase ::= Verb PP NP | Verb | empty
Det         ::= the | a | an
Noun        ::= universe | planets |solar-system
Adjective   ::= earth-like | finite
Det         ::= the | a | an
Verb        ::= exist | finds | expands
PP          ::= in | on
```

This grammar recognizes a given sentence (or parts of it) if the sentence's syntactic-structures match some rules of the grammar. We denote the 'sentence' as a list of strings (or tokens). If some parts of the sentence (starting from the beginning) have been recognized, then the result of recognition is the 'rest of the sentence'. That implies if the whole sentence is recognized successfully, the result is just a list of an empty string. If the recognition fails, the result is an empty list. If the same input can be recognized in more then one ways, then the result contains multiple entries.

For example, application of `Noun_Phrase` to `["the", "universe"]` and `["the", "milky-way"]` results `[[]]` (indicating 'success') and `[]` (indicating 'failure') respectively.

So, the type of the basic recognizer is:

```
type Recognizer = [String] -> [[String]]
```

We define four basic-combinators to construct the complete recognizer-set.

1. The **empty** recognizer always succeeds and simply returns the input.

```
empty input = [input]
```

2. Any terminal is constructed in terms of combinator - **term**, which matchers the first token of the input-sequence with its own token. If a match is found, it returns the rest of the input-sequence, otherwise returns an empty list.

```
term :: String -> Recognizer
term w [] = []
term w (t:ts) | w == t     = [ts]
              | otherwise = []
```

3. We call the 'alternative' combinator **orelse**, which is used as an infix operator between two recognizers. The orelse applies both of the recognizers on the same input-sequence and sums up the results returned by of both of the recognizers, which is eventually returned as the final result. It can be defined as:

```
orelse:: Recognizer -> Recognizer -> Recognizer
(p `orelse` q) inp = unite (p inp) (q inp)
```

We assume the function `'unite'` combines the results returned by the two recognizers and removes the duplicate values.

4. The sequencing of recognizers is done with the **then** combinator. Like 'orelse', it is also used as an infix operator between two recognizers. But it applies the first recognizer to the input-tokens and if there is any successful result of this application, then the second recognizer is applied to the result – returned by the first recognizer, otherwise the final result is an empty list – indicating a failure. One way of defining it is:

```
then:: Recognizer -> Recognizer -> Recognizer
(p `then` q) inp = apply_to_all q (p inp)
     where
     apply_to_all q []     = []
     apply_to_all q (r:rs) = unite (q r)
                             (apply_to_all q rs)
```

The function `apply_to_all` ensures that the second recognizer - `q` – is being applied sequentially to all possible results returned by first recognizer – `p`.

Using these four basic combinators, we now represent the previously mentioned CFG as a combinatory-parser for simple subset of English. Basically the sequencing terminal and/or non-terminals are glued together by **then** combinators, alternatives are represented with **orelse** combinators, terminals and empty recognizers are created by **term** and **empty** combinators. Each non-terminal definition works as an executable-function, which is simple enough to construct, understand and modify.

```
Sentence      = Noun_Phrase `then` Verb_Phrase

Noun_Phrase   = Del `then` Noun `orelse` Adjective
                `then` Noun `orelse` Del `then` Noun
                `then` Verb

Verb_Phrase   = Verb `then` PP `then` NP `orelse` Verb
                `orelse` empty

Det           = the `orelse` a `orelse` an

Noun          = term "universe" `orelse` `orelse` term
                "planets" `orelse` term "solar-system"

Verb          = term "exist" `orelse` term "finds"
                `orelse` `term "expands"

Adjective     = term "earth-like" `orelse` "finite"
PP            = term "in" `orelse` term "on"
```

Figure 4.1: A combinatory-parser representation of a CFG for NL

Below is a list of sample applications of these recognizers to some natural-language inputs:

1. `Sentence ["earth-like", "planets", "exist", "in", "the", "universe"] => [[""]]`
   (A completely successful recognition)
2. `Sentence ["earth-like", "planets", "may", "exist"] => [["may", "exists"]]`
   (A partially successful recognition)
3. `Noun_Phrase ["the","universe", "expands", "uniformly"] => [["expands", "uniformly"],["uniformly"]]`
   (Two different ways of recognitions for the same input – shows ambiguity)
4. `Sentence ["andromeda", "is", "next", "to", "milky-way"] => []`
   (A failed or unsuccessful recognition)

4.2 Use of Monads for Combinatory-Parsing

4.2.1 Monads to Structure Program

As non-strict functional-programming languages 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 monads appear as an easy solution of these kinds of problems. The concept of 'monad' in computing is derived from Category-Theory – a branch of mathematics, which abstractly describes mathematical-structures (*categories*) and relations between them. Moggi [8, 9] showed how monads can be used efficiently to structure semantic-computations. Moggi (1989) and Spivey (1990) demonstrated that maintaining states, raising exceptions, error handling, continuations etc can be performed structurally using monads. Inspired by their works, Wadler established monads as a convenient tool for structuring functional programs [38, 39].

4.2.1.1 Definition of Monad

Our discussion about monads is restricted within its use in functional programming as a software-engineering tool. A monad consists of a triple `(M, unit, bind)`.

M is a polymorphic type constructor.

Function **`unit`** (of type `a → M a`) takes a value and returns the computation of the value. Function **`bind`** (of type `M a → (a → M b) → M b`) applies the computation (`a → M b`) to the computation `'M a'` and returns a computation `'M b'`. The **`bind`** ensures sequential building blocks of computations. To be considered as a valid monad, the triple has to obey following three laws:

Left Unit :: `unit a `bind` k = k a`

Right Unit :: `a `bind` unit = a`

Associative :: `a `bind` (\b → (k b) `bind` (\c → h c))`

`≡ (a `bind` (\b → (k b)) `bind` (\c → h c)`

By adding simple changes to an existing monadic-definition, one can perform complex operations with added requirements in a fairly easier way. Monads, to some extent, mimic an imperative-style programming environment within the scope of purely-

functional language. Monad helps programmers to construct a 'bigger' computation combining sequential alterable blocks of 'smaller' computations. It abstracts the smaller computations and the combination-strategies from the main computation. As monad separates the type of computation from the type of resulting value, it's easier to add new changes to an existing monad to fulfill different computational requirements.

Two most commonly used monads are the **identity-monad** and the **state-monad.** We are particularly interested in the 'state-monad' as this form of monad has been employed in this thesis to construct the combinatory parsers.

Identity-Monad:

This simplest monad just returns the value without attaching any information to it.

```
type Id x = x

unit :: a -> Id a
unit x = x

bind:: Id a -> (a -> Id b) -> Id b
x `bind` f = f x
```

State-Monad:

As maintaining updateable variables (in other words – 'different states') are not permitted in LFP languages, every function-definition, which requires latest state-value, must have a 'state' as input-parameter. For complex functions, maintaining this explicit 'state' is complicated, error-prone and results unstructured and cluttered code. By using state-monads, function-definitions can 'abstract away' the updated 'state' as a function-parameter implicitly. A continuously-changing state-variable can float around within a monadic function-definition without forcing the function explicitly operates on it. One way to define a state-monad is:

```
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** takes a value of any type (along with an initial state of type S, which is abstracted within the definition) and returns a pair that consists of the input-value and the initial state. In other words, the type of the output of **unit** is `'State a'`. The **bind** takes two parameters - the first one, m, is of type `'State a'` and the second one, k, which is a computation that takes a value of type a and returns a value of `'State a'`. The output of **bind** is of type `'State a'`. According to the type-definition of `'State a'`, when x of type S is supplied to m, it returns a pair `(p, y)` where p is the value that 'container' m was holding and state y is of type S. Then **bind** takes p and y from the output of `'m x'`, applies k on p, which returns something of type `'State a'` (which contains a value inside). When output of k is applied to the previously calculated state y, it returns `(q, z)` - by following the definition of `'State a'`. Here q is the new value that the output of `'k p'` was holding and z is the new state (of type S).

4.2.1.2 Example of Monadic Computation

The following simple-but-illustrative example shows how adding some changes to the existing monadic-definition can perform different computational tasks.

With the non-monadic definition for reversing a list, it's quite tedious to retrieve other information about the list. But by converting the naïve definition of 'reverse list' to a monadic definition, we can perform additional tasks in a structured manner.

The original definition for reversing a list:

```
revList []     = []
revList (e:es) = revList (es) ++ [e]
```

Example 1:

This basic monadic-definition (using identity-monad) of reversing a list and the original definition do not have any difference w.r.t. their functionalities. This monadic-definition can be considered as the basic building block for the other monadic-definitions. The type-constructor `'M1 a'` has one data constructor - a. The **unit1** and **bind1** are defined according to the previous identity-monad definition.

```
revList1 [] = unit1 []
revList1 (e:es) = revList1 (es) `bind1` f
                  where f a =  unit1 (a ++ [e])
```

Sample output:

```
*Main> revList1 [3,7,9,0]
[0,9,7,3]
```

Example 2:

This version of 'monadic reverse-list' uses 'state-monad' and is changed in such a way that along with reversing a list, it is also able to return the length of the input list. It was done by changing the type constructor `'M2 a'` so that it can maintain a state of integer type. The definition of **`bind`** was also changed to ensure that the recursive calls can have the latest state.

```
type M2 a = Int -> (a, Int)

unit2 :: t -> M2 t
unit2 x = f where f t = (x,t)

bind2 :: M2 t1 -> (t1 -> M2 t2) -> M2 t2
m `bind2` k = f'
              where f' x = (b,z)
                    where (b,z) = k a y
                          where (a,y) = m x

revList2 [] c          = unit2 [] c
revList2 (e:es) count = (revList2 es `bind2` f)
                        (count+1)
                         where f a
                         =  unit2 (a ++ [e])
```

Sample output:

```
*Main> revList2 [3,7,9,0] 0
([0,9,7,3],4)
```

Example 3:

This version of 'monadic reverse list' is changed in such a way that along with reversing a list, it is also able to detect if the input-list has multiple occurrences of one or more elements. The same state-monad is used but the type constructor `'M3 a'` is changed in such a way that it can print some information. The monad gives us the chance to add some helper functions like 'findDup' and 'chkForDup' for computational flexibility.

```
type S     = [Char]
type M3 a = S -> (a, S)

revList3 []       = unit3 []
revList3 (e:es)   = (drevList3 es `bind2` f)
                     where f a =  unit2 (a ++ [e])

drevList3 lis str = findDup revList3 lis str
findDup revList3 lis str
                  = (out, chkForDup out)
                  where (out, none)
                  = revList3 lis str
chkForDup resList = if (resList == nub resList)
                    then "Has No Repetition"
                    else "Has Repetition"
```

Sample output:

```
*Main> drevList3 [3,7,4,9] ""
([9,4,7,3],"Has No Repetition")
*Main> drevList3 [3,7,9,4,9] ""
([9,4,9,7,3],"Has Repetition")
```

4.2.1.3 Monads in Haskell

Haskell is equipped with many built-in monads (such as: list, maybe, IO etc) and the Prelude[1] contains some monadic classes (such as: Monad, MonadPlus, Functor etc). The standard monad class in Haskell is defined as

```
class Monad m where
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

(where 'return' and '(>>=)' are equivalent to 'unit' and 'bind' of previous discussion)

A basic monad can be constructed by creating an instance of this class. Haskell also provides a special syntax for monad (the 'do' notation – an expressive short-hand notation), which gives programmers a touch of imperative-style-programming in Haskell. Instead of using the built-in monads, we shall use hand-written monads to maintain the clarity of the function-definitions.

[1] The standard library of Haskell

4.2.2 Monadic Parser-Combinators

Wadler (1990, [38]) first noticed that using monads, combinatory-parsers can be represented in an organized manner. Hutton (1996, [19]) described a step-by-step procedure to form functional-combinators using monads. In 2003, Frost demonstrated how monadic parser-combinators can be used to maintain updatable 'state' efficiently during 'memoization'. The main purpose of transforming regular combinators to monadic combinators is to abstract out the underlying computation mechanism in order to add new functionalities in a structured and modular way. In our case, however, the primary requirement is maintaining a changing 'state' securely. We begin by transforming the non-monadic combinators of section 4.1.1 to monadic combinators using identity-monad. Though these definitions don't serve any useful purpose except modularity for now, we show in following sections, how we can systematically replace the identity-monad with a state-monad to provide systematic method for memoization.

```
--   Identity-monad definition
     type Recognizer x = x
     unit :: a -> Recognizer a
     unit x = x
     bind:: Recognizer a -> (a -> Recognizer b) ->
                                Recognizer b
     x `bind` f = f x

--   Basic (identity) Monadic-combinators

     term c []              = unit [""]
     term c (r:rs)| r == c = unit [rs]
                  | otherwise = unit []
     empty x = unit [x]

     (p `orelse` q) inp = p inp `bind` f
                          where f m = q inp `bind` g
                             where g k = unit(union m  k)

     (p `then` q) inp   = p inp `bind` f
                          where f m = q m `bind` g
                                where g k = unit k
```

Figure 4.2: Simple monadic-combinators

In **orelse**, recognizers p and q are applied to the given input **inp** and their results are bound to the variables m and k using **bind**. The **union** of m and k is added to a 'container' of computation through **unit** and returned as the result of orelse. In **then**, recognizer p is applied to the given input and its result is bounded to m with **bind**. Then recognizer q is applied to m and its result is bounded to k, which ultimately is returned as a computation using **unit**. The **term** and **empty** combinators' results are added to a 'container' of computation using **unit** and returned afterwards. Using these combinators, we can form identical recognizers of figure 4.1.

4.3 Shortcomings of Combinatory-Parsing

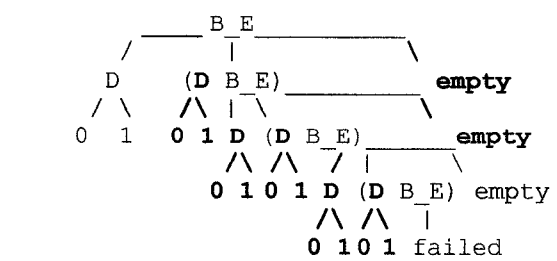
Combinators are very effective for constructing modular top-down recursive-descent backtracking language-processors and to accommodate ambiguity. But they exhibit exponential time-complexity in worst-case and they don't terminate when used to represent a left-recursive production rule.

4.3.1 Exponential Time-complexity

If no precaution is taken, a top-down parser normally exhibits exponential time-complexity while processing an ambiguous grammar. As combinator-parsing follows recursive-descent with backtracking top-down parsing technique, it is inherently exponential. The simple ambiguous grammar - G_1 from section 2.1 that can generate binary numbers – can be expressed using combinators as:

```
B_E  = D `orelse` (D `then` B_E) `orelse`  empty
D    = term 0 `orelse` term 1
```

And when executed on input "000", the execution tree with repeated computations would look like:



(the bold parses are repeated computation)

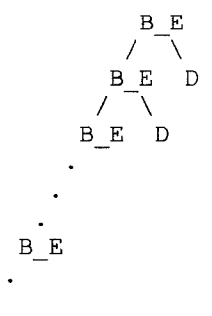
When a non-terminal e.g. `B_E` (or `D`) is applied on the input, according to the definition of `` `orelse` `` combinator, all the alternatives of `B_E` (or `D`) are being applied on the current input without knowing whether the same operation for same input was performed previously or not. For a highly ambiguous grammar, the rate of backtracked alternative re-computations grows exponentially with respect to the length of the input.

4.3.2 Non-Termination for Left-Recursion

If we express the above grammar in a left-recursive form, then the combinator-parser would be:

```
B_E  = D `orelse` (B_E `then` D) `orelse`  empty
D    = term 0 `orelse` term 1
```

The second alternative of the non-terminal `B_E` (`B_E = B_E `then` D`) is a left-recursive rule. When this particular rule is applied on an input `"000"` the following parse will be ever-growing:



When the non-terminal `B_E` is executed on any input, according to the definition of the combinator `` `then` ``, its left-most operand (which is the `B_E` function itself) is being applied on the input first. When this second `B_E` executes its definition, the same scenario occurs again. As `B_E` rewrites itself at the left-most position and doesn't introduce any 'terminal', the parser doesn't get the chance to 'consume' any input-token and therefore the process never terminates.

CHAPTER 5: RELATED PREVIOUS WORKS

5.1 Use of memoization and monads to accommodate ambiguity in polynomial time

The technique of re-using 'previously-stored results' has been used to improve parsing and recognition efficiency by many. Earley's well-known chart-parsing algorithm [6], which uses 'dynamic-programming', requires $O(n^3)$ time in the worst case. Leermakers and Augusteijn [25] used 'memoization' to improve their parsing-algorithm though their explanations are slightly abstract in terms of modularity. It was Norvig [32] who first demonstrated how to construct modular and efficient parser-combinators using a strict functional language (Lisp) with the help of 'memoization'. Inspired by his work, Frost and Szydlowski (1995) constructed a purely-functional versions of memoized language-processors. In 2003, Frost extended the previous work by changing the general-parser combinators to state-monadic parser-combinators to ensure correct systematic memoization. By using memoization, this approach also ensures cubic time-complexity at worst case for recognition. In this section, we discuss **Frost and Szydlowski** [14] and **Frost**'s [9] work briefly.

5.1.1 Basic Concept of Memoization

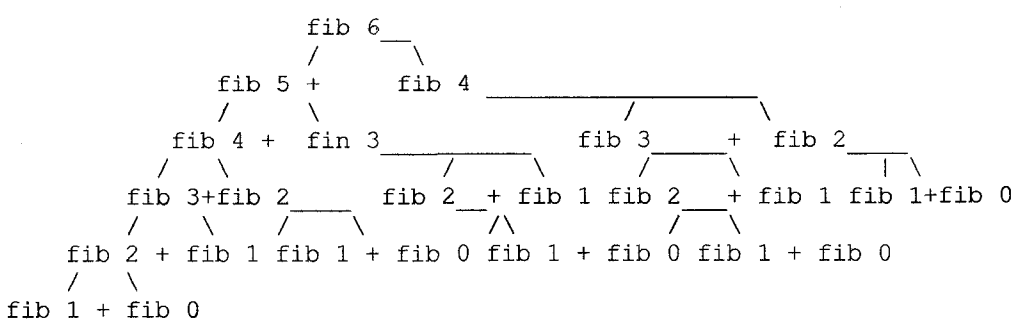
Many recursive programs can be "memoized" to improve efficiency. Memoization (also known as 'top-down dynamic-programming') computes a 'sub-problem' once, saves the result in a storage (we shall refer this storage as 'memo-table') and reuses this result (instead of re-computing it) when the identical sub-problem is required to be solved again. Time-complexity of most of the recursive computations can be reduced from exponential to linear or polynomial using memoization. The whole process is based on two operations:

Update: whenever a result is computed for a sub-problem, it is saved in the memo-table (with a unique identifier (id)) during recursive-ascent - only once.

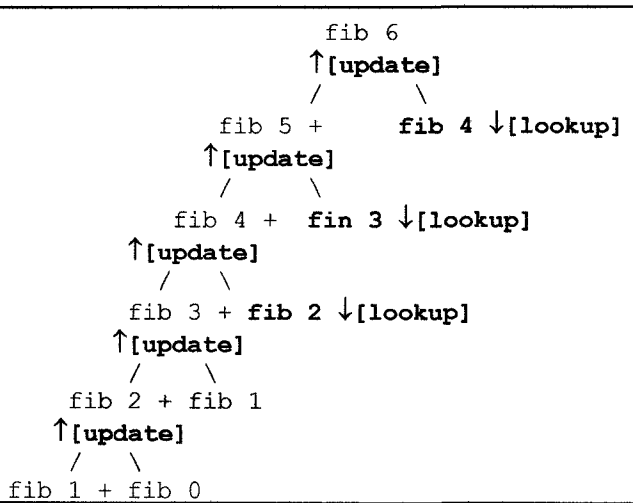
Lookup: if the recursive-process meets the same sub-problem again somewhere during recursive-descent, then the memo-table is being checked with this problem's id and if a match is found, then the saved result is returned, otherwise, the problem has to be

computed. Different variant of 'update' and 'lookup' can be used – according to the needs. For example, consider the recursive definition 'fib' (for computing Fibonacci number) and the execution-tree for input 6:

```
fib n|n == 0 = 0
     |n == 1 = 1
     |otherwise =  fib (n-1) + fib (n-2)
```



Clearly, the time requires to compute 'fib 6' is exponential ($O(2^n)$). But, if the results of sub-problems were saved in a memo-table for future use, then the required time would be reduced to linear:



Memo-table for 'fib 6'

n	result
0	0
1	1
2	1
3	2
4	3
5	5
6	8

Figure 5.1: Memoized Computation of Fibonacci

In an imperative-programming (or in a strict-functional) language, a global memo-table can be used to store the previously computed results, but as there are no updatable variables in lazy-functional languages, the latest copy of the memo-table has to be passed-around as an input argument of recursive function-calls.

5.1.2 Frost and Szydlowski – Memoized Language-processors

Frost and Szydlowski constructed memoized versions of basic combinators (as described in section 4.1.1) for building polynomial-time language-recognizers. Their method differs from Norvig's approach by being implemented in a purely-functional programming language – Miranda. As Norvig used Lisp (a strict functional language), he was able to maintain a globally accessible memo-table within the program-scope, which could be updated or looked up by any function independently. But, as updatable dats-structure is not permitted in LFP languages, Frost and Szydlowski implemented language-recognizers in such a way that they can receive the recent memo-table as input-argument and can also return it as a part of the output. Basically the whole memo-table is threaded through the all recursive calls of the recognizers as an input-argument.

To illustrate their approach, we step-by-step construct memoized parser-combinators that represents the CFG rule "`S::= a S S | ε`". In their original paper, Frost and Szydlowski described the recognition-procedure by identifying inputs as integer indices for improved efficiency. For simplicity and better readability of the function-definitions, in this section we assume that

- input-tokens are represented as characters.
- unique-identifier of a recognizer is a string (i.e. recognizer's name) and
- 'result' of recognition is a list of 'remaining inputs' (or a string).

So, the memo-table can be defined as:

```
type memo-table = [(String,[( String,[String])])]
```

which represents "memo-table = {(recognizer id,{(recognized input-token, {different results of recognition})})}".

In order to perform a **lookup** or an **update** operation, recognizers are 'memoized' by applying a higher-order function – `memoize` – to all recognizers. Through this function, a recognizer first looks up the memo-table, if look-up fails, then the recognizer executes its own definition and whenever it finds a result, it updates the memo-table with appropriate id and newly-computed result. If a similar computation is required at some point later, the recognizer just looks up the table to retrieve the result – instead of re-computing it. The lookup and update functions can be defined as:

```
lookup name inp table
 |res_in_table == [] = []
 |otherwise = res|(i, res) <- (res_in_table !! 0),i == inp]
        where
        res_in_table = [pairs|(n,pairs) <- table,n == name]

update [] name inp res = [(name,[(inp,res)])]
update ((key, pairs):rest) name inp res
 |key == name =(key,(inp,res):pairs):rest
 |otherwise   =((key,pairs): update rest name inp res)
```

The 'lookup' function scans through the memo-table with given recognizer 'name' and current input 'inp'. If there exists a result in memo-table, 'lookup' returns that, otherwise it returns an empty list – indicating a lookup-failure. The 'update' function adds a newly computed result 'res' to the end of the result-set for a particular recognizer 'name' and specific input 'inp'.

```
memoize rec-name recognizer (inp, table)
 |table_res == [] =(res, update newtable rec-name inp res)
 |otherwise =  (table_res!!0, table)
              where
              table_res = lookup rec-name inp table
              (res, newtable) = recognizer inp table
```

The 'memoize' function takes a recognizer-name (unique id), recognizer function-definition, input-string and a memo-table as its input-parameters. From the definition, it is obvious that 'memoize' first performs a 'lookup', if 'lookup' returns an empty list, it permits the 'recognizer' to compute new results. When a result is found, it is updated to the memo-table for later use. To pass-around the memo-table as an input-argument (or as the part of the output) of a recognizer, the definitions of basic combinators can be modified as follows:

```
(p `orelse` q)(inp, memo_tab) =(merge_result[1] p_r q_r,n_tab)
                    where
                    (p_r, n_tab)= p (inp, memo_tab)
                    (q_r, n_tab)= q (inp, n_tab)
```

[1] Assuming already-defined 'merge-result' function adds two sets of results by removing duplicates.

```
(p `thenS` q) (inp, memo_tab)
                | if n_tab /= [] = q (p_r, n_tab)
                | otherwise      = ([],n_tab)
                  where
                  (p_r, n_tab) = p (inp, memo_tab)

empty (inp, memo_tab)        = ([inp], memo_tab)

term c ([], memo_tab)        = ([], memo_tab)
term c (inp, memo_tab)
             | c == head inp = ([tail inp], memo_tab)
             | otherwise     = ([], memo_tab)
```

In `orelse`, recognizer p is applied to the given input and table pair whereas q is applied to given input and table returned by p pair. Recognizers p and q's result-sets are merged to form the result of `orelse`. The combinator `then` applies p to the given (input, table) pair and q is applied to the output and table pair - returned by p. The result of `then` is simply the q's final result. Using these new combinators and the memoize function, the CFG 'S::= a SS | ε' may now be expressed as:

```
s  = memoize "s" (a `then` s `then` s `orelse` empty)
a  = term 'a'
```

This approach results in polynomial time-complexity when implemented correctly. However, it was found in practice, that errors often were made in implementation, resulting in unexpected exponential complexity.

5.1.3 Frost– State-Monadic Language-processors

Frost [9] solved the shortcomings of the previous work by transforming the basic-combinators into state-monadic-combinators and this approach allows the systematic threading of memo-table systematically throughout all recursive recognizer-executions. In addition, use of the state-monad improves the modularity of the language-processor and provides flexibility to add different functionalities (e.g. adding semantic meaning to the production rules etc) to the recognizers. Experiments suggest that this monadic-version is less error-prone and ensures cubic time-complexity for ambiguous grammars flawlessly. To explain Frost's approach, we can simply change definitions of the basic-monad from section 4.2.2 to a state-monad for building basic-combinators. From the discussion of last sections, we know that a memo-table has to be passed around as an input-argument and output of all recognizer-executions for real-time 'update' and 'lookup' operations. By using same 'type' of the last memo-table (represented here as `'S'`) we formulate the state-monad as:

```
type S = [(String,[( String,[String])])]
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)
```

Operations of `unit` and `bind` are identical to the description of state-monad in section 4.2.1.1.We can now reuse the definitions from section 4.2.2 to build the state-monadic combinators:

```
term c []              = unit [""]
term c (r:rs)| r == c = unit [rs]
             | otherwise = unit []

empty x = unit [x]
```

```
(p `orelse` q) inp = p inp `bind` f
                     where f m = q inp `bind` g
                        where g n = unitS(union m  n)

(p `then` q) inp = p inp `bind` f
                   where f m = apply_to_all q m

apply_to_all q [""] = unit [""]
apply_to_all q []   = unit []
apply_to_all q (r:rs) = q r `bind` f
            where f m = apply_to_all q rs `bind` h
                        where h n = unit (union m  n)
```

Beside the use of state-monad, the other change is the definition of 'then' combinator. A function 'apply_to_all' is introduced in 'then' so that recognizer q is allowed to be applied on all possible results returned by recognizer p. All the possible results returned by q are united together, added to 'container' of computation through 'unit' and returned as the result of 'then'. The combinator-recognizers, which represent a CFG grammar can be 'memoized' identically using the same memoize, update and lookup functions - defined in the last section.

For example, the CFG 'S::= a SS | ε' may again be expressed as:

```
s  = memoize "s" (a `then` s `then` s `orelse` empty)
a  = term 'a'
```

A test-execution 's "aaa" []'returns:

```
(["","a","aa","aaa"],
[("s",[("aaa",["","a","aa","aaa"]),
       ("aa",["","a","aa"]),
       ("a",["","a"]),
       ("",[""])])])
```

Both of the above mentioned approaches are for 'recognition' of the given input-sequence, no parsing system was constructed. Moreover, even though Frost's last approach ensures accommodation of ambiguity in polynomial, it is not capable of processing any form of left-recursive grammar.

5.2 Approaches to Accommodate Left-recursion

Kuno (1965) appears to be the first to have used the length of the input to force-termination of left-recursive descent in top-down processing. 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. However, Kuno's method is exponential in the worst case.

Lickman (1995, [27]) showed how Wadler's (1992) idea of 'using (monadic) fixed-point operator to terminate left-recursive recognizer' can be achieved practically. He described a program that takes a BNF representation of a CFG as a input and automatically converts it into a combinator-parser using fixed-point operator. However, as mentioned by Lickman, this approach may not be able to result all possible results (i.e. not complete) and exhibits exponential time-complexity with respect to the length of the input during recognition.

Other attempts, includes Johnson's approach (1995) of integrating memoization with continuous-passing-style (CPS) programming to handle left-recursive grammars appears to solve the problem for recognition in polynomial time. He mentioned that simply memoizing a recognizer (as introduced by Norvig) doesn't help to terminate a left-recursive recognizer, as memoization is 'delayed' due to left-growing parse. His approach to solve this problem, to some extent, is similar to the chart-parsing techniques - developed by Shiel (1976) and Leermakers (1993). According to this approach, the central idea of terminating a left-recursive memoized CPS recognizer is to make sure that 'no un-memoized procedure is ever executed twice with the same arguments'. Johnson mentioned that this approach may be too complicated to convert the recognizers into a parsing-system and a straight_forward implementation would not have enough information for compact-representation of resulting parse-trees.

CHAPTER 6: THE NEW ALGORITHM – FOR RECOGNITION

The proposed-algorithm uses memoization to accommodate ambiguity and left-recursion in polynomial time. It utilizes the state-monadic computation-technique (section 5.1.3, Frost, 2003) for modular and structured construction of parser-combinators and for threading the memo-table correctly trough out all parser-executions. The memoization process has been defined in such a way that it may forcefully terminate a branch of a parse by performing a 'bound-check' with respect to the length of the input-sequence and the depth of the parse. Also, the 'lookup' process of memoization is strictly conditional, which ensures proper re-use of the saved results. If a parser tries to retrieve a result from the memo-table, its current 'context' is compared with its saved 'context' of memo-table with respect to the 'reason' – that curtailed the underlying left-recursive parse, if any. The memo-table is currently able to represent the resulting ambiguous parse-trees in a highly-compact format, which can be viewed as a forest of directed-acyclic-graph (DAG). The definitions of the basic-combinators are redefined (utilizing the flexibility of the state-monad) to maintain n-ary branching of a non-terminal and to generate a list of reasons for curtailment, if any. We first describe the algorithm for recognition (in this chapter) and then parsing (in the next chapter) from a theoretic point of view.

6.1 Basic Definitions

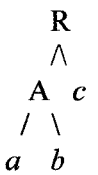
Some definitions - related to the algorithm – are discussed informally in this section:

Algorithm: An algorithm is a *procedure* (a 'method' of executing a series of finite number of instructions) that *halts* or *terminates* (or runs out of instructions) after executing a finite number of instructions in a finite time using finite effort on any number of inputs. An algorithm may have any number of inputs and can produce any number of outputs.

Recognizer: A recognizer simply indicates whether an input can be identified by a given CFG or not. It can be viewed as a decision maker, which, if successfully identifies a part of the input-sequence, points the beginning of the remaining input-sequence, otherwise

returns null. For example, a recognizer can identify the first three characters of the input "abcde" using R::= *abc* | *xy*, so as a result, recognizer returns {"de"}.

Parser: A parser not only indicates the how far of the input-sequence is identified using a given CFG, but also tells 'how' it identifies the part or whole of the input. In other words, a parser results parse-tree(s) as the output of parsing. For example, while processing an input "abcde" using a CFG "R::= Ac | *xy*, A::= *bc* | *pq*", a parser returns:



(In this report, we generally refer to a definition of a non-terminal/terminal using combinators, as a recognizer or parser)

Recursive Recognizer/Parser: A recognizer r is **left-recursive** if the left-most recognizer in any of r's 'alternatives' either immediately (**direct left-recursive**) or through some other recognizer-execution (**indirect/hidden left-recursive**) rewrites to r again without performing any 'recognition'. For example, R ::= R *a*|*b* is a direct left-recursive recognizer, whereas R′::= A *a*|ε , A::=R′*a*|*b* is an indirect left-recursive recognizer. For a **direct left-recursive recognizer,** (1) at least one of the alternatives has to rewrite to a terminal or 'empty' (through a terminal or a non-terminal) at its **left-most position**. For an **indirect left-recursive recognizer** either (1) is true and/or one of the 'causing' immediate non-terminal's one of the alternatives has to rewrite to a terminal or 'empty' (through a terminal or a non-terminal) at its left-most position. Any other forms of recursive-recognizers are **non left-recursive recognizer,** which recursively call themselves following some 'other' terminal(s)/non-terminal(s). The 'other' terminal(s)/non-terminal(s) must rewrite to a 'terminal' or 'empty'. For example, R′′::= *a* R′′| ε is a non-left recursive recognizer. Any form of non left-recursive recognizer's one of the alternatives has to rewrite to a terminal or 'empty' (through a terminal or a non-terminal) at its **right-most position**. Same definitions are applicable to the 'parsers' too.

6.2 Overview of Basic Recognition

For computational efficiency, we define the recognition-process in terms of integer-indices. Assume that the sequence of input-token is represented by `'input'` and the length of it is `'#input'`. Each input-token can be accessed by an integer-index or start-position. The result of recognition is expressed with a set of a pair of integers `(i, j)` - where `i` is the 'start-position' and `j` is the 'end-position + 1' of the character-sequence, which has been 'recognized' by a particular recognizer. An 'empty' result-set indicates that the recognizer has failed to recognize the given input-sequence successfully. For example, if an `input` of length 5 - `"abcde"` - is to be recognized by a recognizer `r::= a b c | ε`, then the result of the recognition would be `{(1,4), (1,1)}`.

We now provide a set-theoretic definition of previously-discussed basic recognizers in terms of using indices. We use set-theory notation to simplify proofs of termination and complexity (given later). The simplest recognizer that recognizes a single terminal (or a character, in our case) is **`'term'`** that takes an integer i, which indicates the i^{th} position of the input-sequence, and a terminal 'c' as input. If i is greater than the length of the input, the recognizer returns an empty set. Otherwise, it checks to see if the character at position i in the input is equal to the terminal 'c'. If so, then it returns a singleton-set containing a pair `(i, i + 1)`, otherwise it returns the empty set. The basic recognizer for any terminal 'c' is defined as follows:

```
term1 i c = {}            , if i > #input
          = {(i, i + 1)}  , if token at position i == c
          = {}            , otherwise
```

The next simpler recognizer is **`'empty'`** that takes a single index i and returns a singleton-set containing a pair `(i, i)` - indicating no-action.

```
empty1 i = {(i, i)}
```

The next recognizer-operator **`'orelse'`** is responsible for representing production-rules having different alternatives (i.e. `r::= p | q`) in their definitions. The **`'orelse'`** takes two alternative recognizers (p and q) and an index i that indicates the start-position as input-arguments, applies each recognizer individually to i and unites the results returned by both recognizers. It can be defined as:

```
(p orelse1 q) i = (p i)∪(q i)
```

The sequencing of one terminal or non-terminal after another in a production-rule (i.e. `r::= p q`) is achieved with the **`'then'`** recognizer-operator. It takes two sequencing recognizers (p and q) and an index i that indicates the start-position as input-arguments, applies the first recognizer to i, then applies the second recognizer to a set of end-positions (which are paired with start-positions) returned by the first one - p. At the end **`'then'`** returns the union of each of the results returned by the applications of the second recognizer q. We can define it as:

$$(p\ then1\ q)\ i = \bigcup(map\ (q\ pick\_2^{nd})\ (p\ i))$$

In order to avoid exponential behavior (caused by repeated same computation) of a recognizer that represents an ambiguous CFG, we define the 'memoization' procedure (as described in section s 5.1.1, 5.1.2, 5.1.3) as follows:

memoize1
```
Input : recognizer name, recognizer, start position i
Output: (a set of (start-pos, end-pos+1) pairs, memo-table)
Method: if lookup succeeds,
          return memo-table result
        else
          apply recognizer to i update table with results
        return (results, updated memo-table)
```

lookup1
```
Input : recognizer name, start position i
Output: a set of (start-pos, end-pos+1) pairs
Method: if memo-table has result for rec_name at i
          return result
        else
          return empty set
```

update1
```
Input : recognizer name, new result, start position i
Method: if an entry exists in memo-table for
        rec_name at i,
          union {add the new-result to the end}
        else
          create new entry for rec-name with
          new-result at i
```

We assume, for now, that the memo-table of `memoize` procedure is globally-stored and has a type of

```
{{recognizer name, {start position, {(start position, end position + 1)}}}}.
```

An update-procedure is only executed during the recursive-ascent phase – when the recognizer has computed a result already and ready to pass the control to the next one. And the lookup procedure is executed during recursive-descent phase – when the recognizer checks the memo-table for previously saved results before 'going down'. Using these basic building-blocks and procedures, an example CFG `'S::= a S S | ε'` can be expressed as

```
S = memoize1(((term1 a then1 S) then1 S) orelse1 empty1)
```

and an execution of `S` on an input-sequence `"aaa"` at the start-position 1 results:

```
{(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4)}.
```

As mentioned earlier, each pair is in the result-set has a type `(start-position, end-position + 1)`. For example, a pair `(1, 3)` implies that the recognizer `S` has successfully identified first two characters of the input `"sss"`. Notice that, as we are performing recognition only, it is sufficient to have only one copy of the (start, end) pair in the result set. But there might be more then one resulting (start, end) pairs – in other words there could be ambiguous results, which are not necessary to detect with the recognition-procedure. It is parser's job to identify all possible combinations of ways in order to detect different syntactic-structures of any given sequence of input. But if the above mentioned CFG's equivalent left-recursive version `'S::= S S s | ε'` is expressed as a left-recursive recognizer -

```
S = memoize1 (((S then1 S) then term1 s) orelse1 empty1)
```

and is executed on the same input at the same start-position, then the procedure won't terminate (section 4.3.2).

6.3 Accommodating Direct Left-Recursion

6.3.1 Condition for Curtailment

Our approach for handling left-recursion is to impose an upper-bound limit on the number of recursive calls of a left-recursive recognizer at each start-position while processing a particular input. More specifically, to retrieve all possible results, a recognizer r_i is only required to call itself at most n times at position j, where

```
j = any start-position of the input, input# ≥ j ≥ 1
n = input# - (j -1)
```

That implies, to curtail a recursively descending left-recursive recognizer r_i from ever-growing when applied at a start position j, the following condition-check is sufficient:

```
if     rij > n
then   ri is 'curtailed'
else   ri performs another recursive-descent operation
where  rij = number of time ri has been called at position j.
             it increases each time ri is called at the same
             start-position. We shall refer this counter as
             'left-rec-counter'.
```

We attach an r_{ij}-value for every recognizer (non-terminal) of the grammar. For any non-left recursive recognizer, the value of r_{ij} will never be more then one. This is because if a recognizer r_i' is non-left recursive then it will never apply itself again at the same start position, as either it will 'fail' to recognize the input-sequence or will 'consume' some input-token from the input before applying itself again. Hence, a non-left recursive recognizer will never be forcefully curtailed. These observations introduce the following assumptions and lemmas:

Assumption 6.1

Iff every alternative of a recognizer is tried on each input-token then the recognizer's **attempt** to compute ambiguous-recognition is correct.

(This assumption stays valid for memoized recognizers too because re-using a result from the memo-table for a particular recognizer r at a particular start-position j is equivalent to computing a result by executing r on j.)

Lemma 6.1

If any non-left recursive recognizer's (r'_i) left-rec-counter value (r'_{ij}) at any start-position (j) is 1, then the recognizer's attempt to compute ambiguous-recognition is correct.

Direct Proof:

Let a non left-recognizer, $r'_i ::=$ ..... | a..... r'_i .....| b..... r'_i..... be applied to an input of length n. So, the start-positions (j) of input are $\{1, 2, ..., i, ..., n\}$ and initially $r'_{ij} = 0$.

Regardless of the value of n and $\forall j \in \{1, 2, ..., i, ..., n\}$, initially r'_i goes 'down' one step and sets $r'_{ij} = 1$. According to the definition of a **non-left recursive recognizer**, r'_i then applies the left-most 'symbol' (a or b) of its definition on j and this symbol (either a terminal or a non-terminal) does not introduce r'_i again without 'consuming' at least one input-token. If a or b fails to consume any input at j, the process terminates, otherwise all next applications of any processor is applied to $j+1$ position, if any. That implies, at position j, control of the processor goes to a different recognizer (a or b) or completely terminates leaving r'_i' s r'_{ij} value at 1 and other alternatives are applied to j sequentially (definition of **combinators**). Hence, recognizer's attempt to compute ambiguous-recognition is correct (assumption 6.1). □

Lemma 6.2

If any left-recursive recognizer's (r_i) left-rec-counter value (r_{ij}) at any start-position (j) is equal to n (where n = #input-token - that r_i is currently processing), then the recognizer's attempt to compute ambiguous-recognition is correct.

Proof by induction on #input:

Let a left-recursive recognizer, $r_i ::=$ a..... r_i .....| r_i.....r'.....| b..... r'_i..... be applied to an input of length n. So, the start-positions (j) of n are $\{1, 2, ..., i, ..., n\}$ and initially $r_{ij} = 0$.

Base Case:

For $n = 1$ and $\forall j \in \{1\}$, when r_i is applied at $j=1$, initially r_i goes 'down' one step and sets $r_{ij} = 1$. According to the definition of a **left-recursive recognizer** and

parser-combinators, r_i then applies the left-most 'symbol' of one of its alternatives, which is rewritten to r_i again (directly or indirectly), if we let it go 'down' one more step then r_{ij} eventually would be 2 and at this point r_i's growth is curtailed by indicating this alternative as 'failed' .This lets other alternatives of r_i to be applied at $j=1$ (definition of combinators). As $n = 1$ and at least one of alternatives consumes the input, there will no input-token be left for further processing. Hence, r_i's attempt to compute ambiguous-recognition is correct (assumption 6.1).

Hypothesis:

Assume that the claim be true for $n = k$. That implies, r_i's r_{ij}-values $\forall j \in \{1,2,...,i,...,k\}$ are equal to $\{k,k-1,...,i,...,2,1\}$ respectively and this ensures r_i's attempt for ambiguous-recognition is correct. We now show that the claim is true for $n = k+1$.

Inductive Step:

For $n = k+1$, the start-positions are $\{1,2,...,i,...,k,k+1\}$ and when j = 1, the length of remaining input-token = $k+1$. Up to k^{th} token, r_{ij}-value is k (hypothesis). And then from the base-case, r_i needs to go 'down' one more step (hence increasing r_{ij}-value by 1) for allowing other alternatives to recognize the $(k+1)^{th}$ token. That implies, up to k^{th} token at $j=1$, r_{ij}-value of r_i is $k+1$. It can be shown in the similar way that r_i's r_{ij}-values $\forall j \in \{1,2,...,i,...,k, k+1\}$ are equal to $\{k+1,k,k-1,...,i,...,2,1\}$ respectively, which ensures r_i's correct attempt for recognition.

Theorem 6.1

Any recursive-recognizer's left-rec-count value at a particular start-position can be at-most equal to the length of the input it is currently processing for its correct recognition attempts.

Direct Proof:

Directly proven from Lemma 6.1 and Lemma 6.2, as all recognizers fall either one of these two categories. □

[This theorem is applicable for parsers too, as number of recursive-calls remains same in both cases]

6.3.2 Modified Memoization for Direct Left-recursion

To accommodate theorem 6.1, we define the **memoize** procedure as follows:

```
memoize2
Input : recognizer name, recognizer (ri), start position j
Output: ({(start-pos, end-pos)}, memo-table)
Method: if lookup2 succeeds,
          return memo-table result
        else
          if rij > #input - (j -1)
             return {}
          else
               increment rij counter by 1
               apply ri to j & update2 table with new-results
               return (results, updated memo-table)
```

lookup2 = lookup1

```
update2
Input : recognizer name, new result, start position j
Method: if an entry exists in memo-table for
        rec_name at j,
          replace the old-entry with the new result
        else
          create new entry for rec-name with
          new-result at j
```

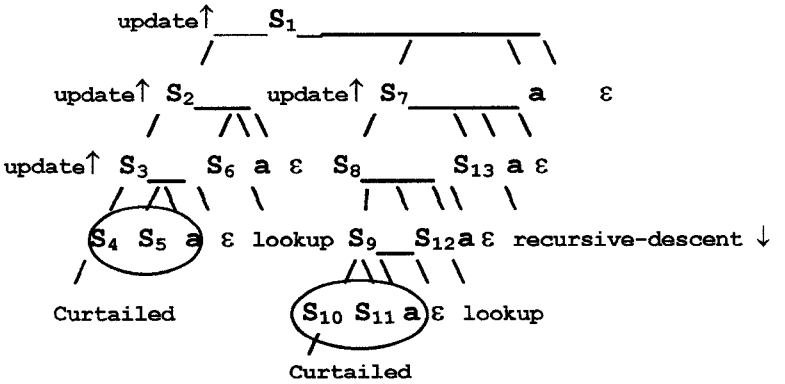
We **memoize** every recognizer in order to check the 'curtailment-condition', which ensures a recognizer's attempt for recognition is correct. As the new-result is computed on recursive-ascent - by applying all possible alternatives of a recognizer-definition, it contains the older-results too (if any). Hence, it is sufficient to replace the older result with the new result in **update**. Memoization reduces the number of recognizer-execution at a same start-position from exponential to polynomial.

Example:

Consider the same left-recursive grammar '`S::= S S a | ε`' that now can be expressed as a memoized combinator-parser as

```
S = memoize2 "S" (((S then1 S) then1 term1 a) orelse1 empty1)
```

and when recognizer S is applied on input `"aaa"` at start-position `j= 1` we have the execution-tree of figure 6.2.



1. At j =1, S recursively descending until its left-rec value > n = 3. At this point (S_4), one alternative 'S->S S a' is curtailed but S->ε is applied that returns a result {(1,1)} for S at j=1 and S_3 **updates** this in memo-table.
2. On the way up, at S_6 ,S **looks-up** the memo-table for j=1 and retrieves the result {(1,1)}. At S_2, S computes new result {(1,2),{1,1}} and replaces the old result for j=1 in memo-table through **update**.
3. At S_7, S has two start-positions j= 2,1. As there is no result for j=2 in memo-table, S now goes recursive-descent phase for j=2 and get curtailed at S_{10}. This lets S->ε rule to be executed and S **updates** {(2,2)} to memo-table for j=2.
4. At S_{12}, S performs a successful lookup and at S_8, S updates a new result {(2,2),(2,3)} for j=2. At S_{13}, S **looks-up** for j=2 successfully but goes to recursive-descent again for j=3 and **updates** result {(3,3),(3,4)}.
5. This same process is performed repeatedly and at S_7, S **updates** {(2,2),(2,3),(2,4),(3,3),(3,4)} and eventually at S_1, s returns the final set of results:
{(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4)}.

Figure 6.1: 'Condition for curtailment' for left-recursive recognition

6.4 Accommodating Indirect Left-Recursion

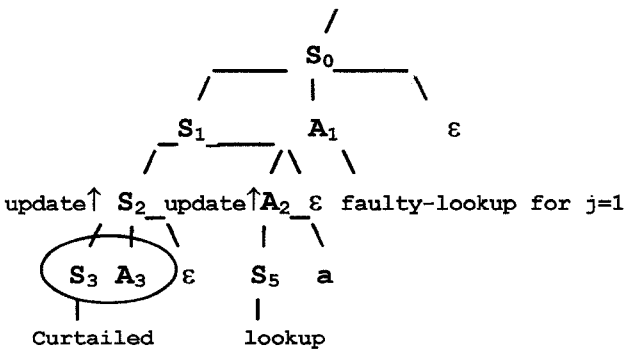
6.4.1 The Problem

Though the 'curtailment-condition' ensures right growth of the correct applications of all alternatives for recognition, for indirect left-recursive recognizers the process tends to do too little work – which eventually causes some missing results (or parses – in terms of parsing). This problem results from *premature* or *out-of-place* lookup-operations during memoization. If there is no memoization involved in recognizer-executions, then there wouldn't be any problem.

Consider the recognizer

```
S = memoize2 "S" ((S then1 A) orelse1 empty1)
A = memoize2 "A" (S then1 term1 a)
```

(which is the equivalent Chomsky Normal Form (CNF) of `'S = memoize2 (((S then1 S) then1 term1 a) orelse empty1)'`). The result of executing `S` on input `"aaa"` at start-position `j=1` and according to section 6.2.2.1, a part of the execution-tree is:



1. According to the 'curtailment-condition', S is curtailed at S_3.
2. S_2 computes a result {(1,1)} using 's->ε' for j=1.
3. A_2 computes a result {(1,2)} using 'A->((S-> **lookup**)a)' for j=1 and updates to memo-table.
4. S_1 updates {(1,1),(1,2)} for j=1.
5. A_1 tries to compute for j=1,2 and for j=1, A already has a result in memo-table (step 3). But that result {(1,2)} was computed in a different **context** in terms of A's curtailed left-recursive predecessor S. To get the right result {(1,2),(1,3)} at j=1, A needs to perform another recursive-descent operation at A_1.

Figure 6.2: Faulty 'out-of-context' lookup

As a left-recursive recognizer's attempts to find the 'left-most derivation' is 'delayed' until the curtailment-condition is satisfied, we can say its process of recognition occurs on recursive-ascent. If a recognizer falls within the parse of a curtailing indirect left-recursive recognizer and if it tries to re-use a result, it needs to make sure that its parse has recursively descended enough when the result was stored to re-use the result later. In the previous example (figure 6.2), when A saved a result {(1, 2)} at A_2 for j=1, its left-recursive predecessor S was called twice (at S_0 and S_1) and the same left-recursive recognizer S was curtailed at S_3 as A's or any of its siblings' successor. And when A at A_1 tries to re-use a result (which was computed at A_2), A's curtailed left-recursive predecessor S was called only once (at S_0). Therefore A's contexts are not equal at A_1 and A_2 in terms of how many times A's or any of its siblings' curtailed-left-recursive successor was called as A's predecessor. It is obvious because at A_2, when A started to compute a result, it appeared as S_0->(S_1 -> (S_2 A_2)) and at A_1, A appeared as S_0->(S_1 A_1). Hence, it is not correct for A to re-use a result at A_1 and in order to re-use, A_1 has to grow one more step down. On the other hand, when S computed a result at S_2, S (the curtailed-successor) was called twice (at S_0 and S_1) as predecessor and when S_5 performs a lookup, it should be allowed to re-use the result because S (the curtailed successor) was called twice (at S_0 and S_1) too as the predecessor of S_5. Hence, S's contexts at S_2 and S_5 are the same. This scenario doesn't apply for a non left-recursive parser's attempt to re-use a result, because of its attempts to find the 'left-most derivation' is not 'delayed' or is 'at the right place' and it won't try to re-use a result before growing correctly. That means, for a particular start-position, a non-left recursive recognizer's growth of parse is not context-dependent.

6.4.2 Context-Based Re-use, Modified Combinators and Memoization

From the previous-section's analysis, it is understood that

1. We need to know a parse-result's **reason(s) - r** for curtailment, if any. Also we need to pass the current-context (of the current-recognizer) downwards during recursive-descent.
2. If a result is to be saved, we need to save the context in which it was computed, which we call the **left-rec-context – lc**, with respect to the reason(s) for curtailment, if any.
3. Before re-using any saved-result, we need to make sure that a recognizer's **current-context – cc** is appropriate or the recognizer is at right place - with respect to **left-rec-context.**

6.4.2.1 Generating and Passing 'Reason for Curtailment' and 'Current-context'

During recursive-descent, a recognizer needs to pass down its id and left-rec-count (which we call a **context**) so that at a particular position, any recognizer can have its predecessors' 'context'. For a specific start-position, a recognizer's predecessors' context and its own context form **current-context - cc** for this recognizer. A subset of this context (if applicable) will be stored with the recognizer's computed-result as **left-rec-context – lc** (explained later).

```
current-context - cc
            = {(start-position, {( recognizer- name,
                left-rec-counter)})}
```

The memo-table is also changed to accommodate the reason and the saved results' left-rec-context.

```
memo-table = {( recognizer name,
               {(start-position,
                 (left-rec-context,
                  {(start position, end position + 1)})})}
reasons    = { recognizer-name }
```

For each result, which is computed by a recognizer with sequencing and alternating combinators, we need to know if its successor(s) or siblings' successor(s) contains left-recursive recognizer(s) – that has been curtailed according to the 'condition for

curtailment', and if so, which recognizer(s) caused the curtailment. Each result should be paired with a list of recognizer-ids (i.e. **reasons - r**) which caused any curtailment in the sub-tree below the result. The **reason** is passed up on recursive-ascent together with the result as a pair. As memoization enforces the curtailment-condition, we have to modify the **memoize** procedure for producing the **reason** after curtailing a parse.

```
memoize3
Input : recognizer name, recognizer (r_i), start-position(j),
        current-context (cc)
Output: ((reasons,{(start-pos, end-pos + 1)}), memo-table)
Method: if lookup3 succeeds,
          return memo-table result
        else
          if r_ij > #input - (j -1)
             return {( {recognizer-name}, {})}
          else
               increment r_ij counter by 1
               apply r_i to j & update3 memo-table with
               results and left-rec-context (if applicable)
          return ((reason, results), updated memo-table)
```

We also need to modify the definitions of the **sequencing** and **alteration** combinators for allowing them to 'pass-up' the **reason**. The combinators merge the recognizer-ids, which caused curtailment, as follows:

```
(p orelse3 q) i cc = (reason_p ∪ reason_q,
                      result_p ∪ result_q)
                      where
                     (reason_p, result_p) = p i cc
                     (reason_q, result_q) = q i cc

(p then3 q) i cc   =
            (reason_p  ∪  reason_q, result_q)
             where
            (reason_q, result_q)
             = fold (⋃ over first and second set of the result-pair respectively ) ({},{}) map (q' cc) result_p
             where
             q' cc i                = q i cc
             (reason_p, result_p)   = p i cc
```

Recognizers consist of `term` and `empty` don't have any underlying 'reasons', hence the reason-part of their result is empty:

```
term3 i c cc = ({},{})            , if i > #input
             = ({},{(i, i + 1)}) , if token at
                                     position i == c
             = ({},{})            , otherwise

empty3 i cc  = ({}, {(i, i)})
```

6.4.2.2 Storing the Result with 'Left-rec-context' During 'Update'

Whenever a recognizer computes a result at a specific start-position, the reasons for curtailment (if any) of the result - generated during the computation (comes from recursive-ascent) and the current recognizer's current-context (computed during recursive-descent) are examined. If any 'reason' exists in the current recognizer's 'current-context' at current position, then that context (the **left-rec-context – lc,** which includes the recognizer-id(s) and respective left-rec-counts at current position) is updated to the memo-table with the newly computed result. The new **`update`** operation consists of the following procedure:

```
left-rec-context - lc
              = {( recognizer- name, left-rec-counter)}

update3
Input : recognizer_name, (reason - r, new result - res),
        start position j, current-context - cc, memo-table


Method: let left-rec-context = {}
        if (there is any r paired with res)
        then (∀x ∈ {cc.j.recognizer-name} at position j)
             if x == ∃y ∈ {r. recognizer-name}
             then (x,x.left-rec-count): left-rec-context
             else -- do nothing
        else -- do nothing

        if an entry exists in memo-table for
           recognizer_name at j,
        then replace the old-entry with
             (left-rec-context, res)
        else create new entry for rec-name with
             (left-rec-context, res)  at j
```

The only part of the current-context which is stored with a 'result', is a list of those recognizers and their left-rec-counts that had an effect on curtailing the result.

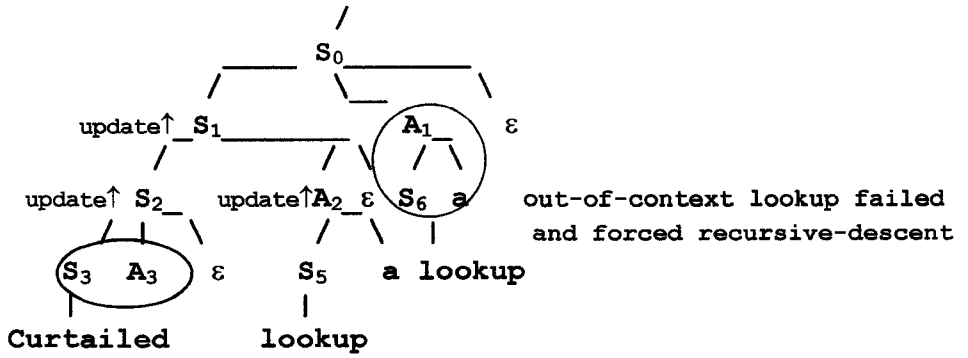
6.4.2.3. Condition for Re-using the Saved Result During 'Lookup'

Whenever a memo-table result is being considered for re-use by a recognizer at a particular start-position j, the **left-rec-context - lc** – saved with the result - is compared with the **current-context - cc** of the current recognizer at j for start-position j. The result is reused if, every recognizer-id of `lc` exists in `cc` (for j) and all of the left-rec-count of `lc`'s recognizer-id is equal or greater to the left-rec-count of `cc`'s recognizer-id. If there were no curtailments (in case of non-left recursive recognizers), the left-rec context of a result would be empty and that result can be reused irrespective of the current-context. So, the changed **`lookup`** procedure is:

```
lookup3
Input : recognizer-name, start position j, memo-table
        (contains left-rec-context - lc and result - res) ,
        current-context (at j) - cc
Method:
  if memo-table has entry for recognizer-name at j
  then
      if lc == {}
      then re-use the result res
      else (∀x ∈ {lc.j.parser-id}),(∃y ∈ {cc.j.parser-id})
          If (x == y ∧ x.left-rec-count >= y.left-rec-count)
          then re-use the result res
          else recognizer goes to 'recursive-descent' phase
               by returning empty-set
  else return empty-set
```

This makes sure that a result - stored for some recognizer at start-position j - is only reused by a subsequent application of the same recognizer, at the same position, if the left-rec context of the later executions of the recognizer would constrain the result equally as much as it has been constrained by the left-rec context for the previous application of the same recognizer at j.

According to this modified context-based memoization, the example-recognition of figure 6.2 now can be computed correctly as follows:



For example:

S at j=1

S_2's lc = {(1,{(S,2)})}, (r, res) = ({S},{(1,1)})

S_5's cc = {(1,{(S,2)})}

As lc.S.left-rec-counter == j.cc.S.left-rec-counter, S_5 re-uses S_2's result

A at j=1

A_2's lc = {(1,{(S,2)})}, (r, res) = ({s},{(1,2)})

A_1's cc = {(1,{(S,1)})}

As lc.S.left-rec-counter ≠ j.cc.S.left-rec-counter , for A at A_1, instead of 'lookup', A goes to a 'recursive-descent' phase and eventually computes a new result for j=1 and updates to memo-table:

{(1, 2),(1,3)}

Figure 6.3: Restricted re-use of result when recognizer is 'out-of-context'

6.4.3 Results in Memo-table

Up to this point, our discussion is limited to only 'recognition'. An application of a memoized-recognizer

```
S = memoize3 "S" ((S then3 A) orelse3 empty3)
A = memoize3 "A" (S then3 a)
a = memoize3 "a" (term3 a)
```

} G_2

to the input-sequence "aaa" saves all possible ways to recognize "aaa" in the memo-table by indicating the starting and ending position of the recognized-tokens (as a pair of (start- position, end-position + 1)).

Start positions

Recognizer names	1	2	3
S	(1,1) (1,2) (1,3) (1,4)	(2,2) (2,3) (2,4)	(3,3) (3,4)
A	(1,2) (1,3) (1,4)	(2,3) (2,4)	(3,4)
a	(1,2)	(2,3)	(3,4)

Figure 6.4: Memo-table represents results of recognition using G_2

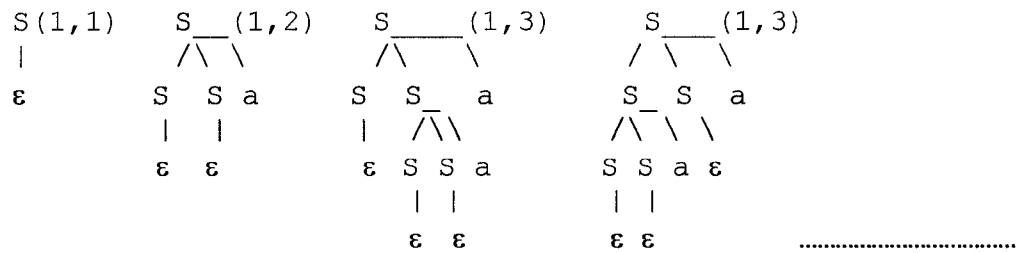
A sample "snapshot" of the memo-table is shown in figure 6.4. As this representation only informs us how far the input-token has been recognized using which recognizer, in the next chapter we transform the algorithm in to a parser to indicate the syntax-structure(s) of the recognized input-tokens i.e. the parse-trees. Note that as we have united the results obtained through sequencing and alteration, the duplicate results (more then one result having same start and end position) are not indicated in the memo-table for recognition.

CHAPTER 7: THE NEW ALGORITHM – FOR PARSING

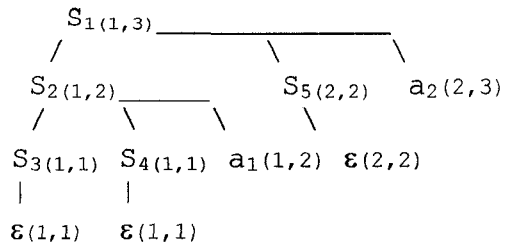
7.1 Overview

Recognition is not sufficient to identify the grammatical-structure of an input sentence. From NLP point-of-view, parse-trees generated by a parser, are essential to incorporate semantic-meanings or theories to the syntax-structures of an input-sentence. It is also important for a parser to generate all possible ambiguous parse-trees so that all possible 'meanings' of a sentence can be retrieved. For example, an application of

`S = memoize3 "S" (S then3 term3 a orelse3 empty3)` to input `"aaa"` would generate `{(1, 1), (1, 2), (1, 3), (1, 4)}`. Instead of this, we would like to have a set of parse-trees:



We need to add some extra information to the memo-table for constructing a parse-tree. A memoized-recognizer – constructed using alteration and sequencing – descends downwards until it recognizes some tokens and then ascends upwards to recognize another token. Throughout this interchanging recursive-process, a recognizer actually visits all required 'nodes' to construct a parse-tree. The extra pointing-information should be able to indicate 'where to go next' from one point of a parse. A parser can keep track of this pointing-information by saving some information about its 'previously-visited' nodes during recursive-ascent, along with the information about the end-points. For example, consider the 4th parse-tree of the above example again:



1. At S_2, non-terminal S tries its one of the alternatives "S -> S S a". When S recognizes "empty" at S_3 (for start-position 1), it can store a reference of the rule it used on its way up with its recognition-result in form of 'S (1, 1)->ε (1,1)'. It then uses same rule again to recognize another "empty" at S_4 (for start-position 1), hence as entry 'S (1, 1)->ε (1,1)' can be stored. In the sequence, "a" is recognized by "term3 a". So, at the end of the rule "S -> S S a", S can save an entry 'S (1, 2) -> S (1, 1) S (1, 2) a (1, 2)' where each pointer attached with a terminal or non-terminal keeps the information of 'where to go next' if we look at from top-down.
2. Similarly, at the end of recursive-ascent when S reaches at S_1, it can save an entry 'S (1, 3) -> S (1, 2) S (2, 2) a (2, 3)'.
3. The stack of saved results, according to the order of computation, would be:

   5. S (1, 3) -> S (1, 2) S (2, 2) a (2, 3)
   4. S (1, 1) -> ε (1,1)
   3. S (1, 2) -> S (1, 1) S (1, 2) a (1, 2)
   2. S (1, 1) -> ε (1,1)
   1. S (1, 1) -> ε (1,1)

   And if we follow the pointers from top (S (1, 3)), we'll have the above-mentioned parse-tree.

Figure 7.1: Basic idea of constructing a parse-tree

7.2 Concepts of Compact-representation

It is desired that a top-down parser should identify an exponential number of parse-trees for an ambiguous grammar. If no precaution is taken, the space-requirement for representing ambiguous-parses would be exponential, which is inefficient for practical uses. To avoid this, we represent the resulting parse-trees as a forest of n-ary one-level-

depth branches (and leaves) with pointers attached at each sub-node, which ensures polynomial space-complexity by sharing common sub-trees and grouping ambiguities. The overall representation acts as a directed-graph (to some extent similar to the Tomita's [35] compact-representation for general LR parsing algorithm). A branch of an example non-terminal

(R::= A_1 A_2 A_3 | b_1 B_2 | C_1) represents one of R's alternatives' (which has been used recently to compute a result) terminals and/or non-terminals in a sequence. Each of the nodes of the branch has pointers to their own entries in the memo-table. If one of R's alternatives "A_1 A_2 A_3" is used to recognized part of current input (for example, start-position = 2 to end-position+1 = 8), we represent it as:

Node R (2, 8)
↓
Branch [Sub-node A_1 (2, 3) Sub-node A_2 (3,4) Sub-node A_3 (4,8)]

For the same start-end position, R may have more (ambiguous) parses with the same alternative and/ or with different alternatives:

Node R (2, 8)
↓
Branch [Sub-node A_1 (2, 4) Sub-node A_2 (4, 5) Sub-node A_3 (5, 8)]
Node R (2, 8)
↓
Branch [leaf b_1 (2, 3) Sub-node B_2 (3, 8)]

Node R (2, 8)
↓
Sub-node C_1 (2, 8)]

Each of the non-terminals (pointed sub-nodes) has its own entry in the memo-table under the particular start-end position. The terminals (leaves) indicate the "bottom" of a parse. If another non-terminal (for example D ::= R | A_2 A_3) also parses from start-position = 2 to end-position+1 = 8 at the upper level of R, then D needs to have four different entries in the memo-table to point to R's four different results.

Node D (2, 8)
↓
Sub-node R (2, 8)]

Node D (2, 8)
↓
Sub-node R (2, 8)]

Node D (2, 8)
↓
Sub-node R (2, 8)]

Node D (2, 8)
↓
Sub-node R (2, 8)]

Though this representation is still 'compact', we can achieve a more densely-compact form by grouping branches of a non-terminal together, which have been used to parse a specific start-end position, also merging the common branches (of a same non-terminal for same start-end position) in to a single one. This representation reduces the space-requirement significantly.

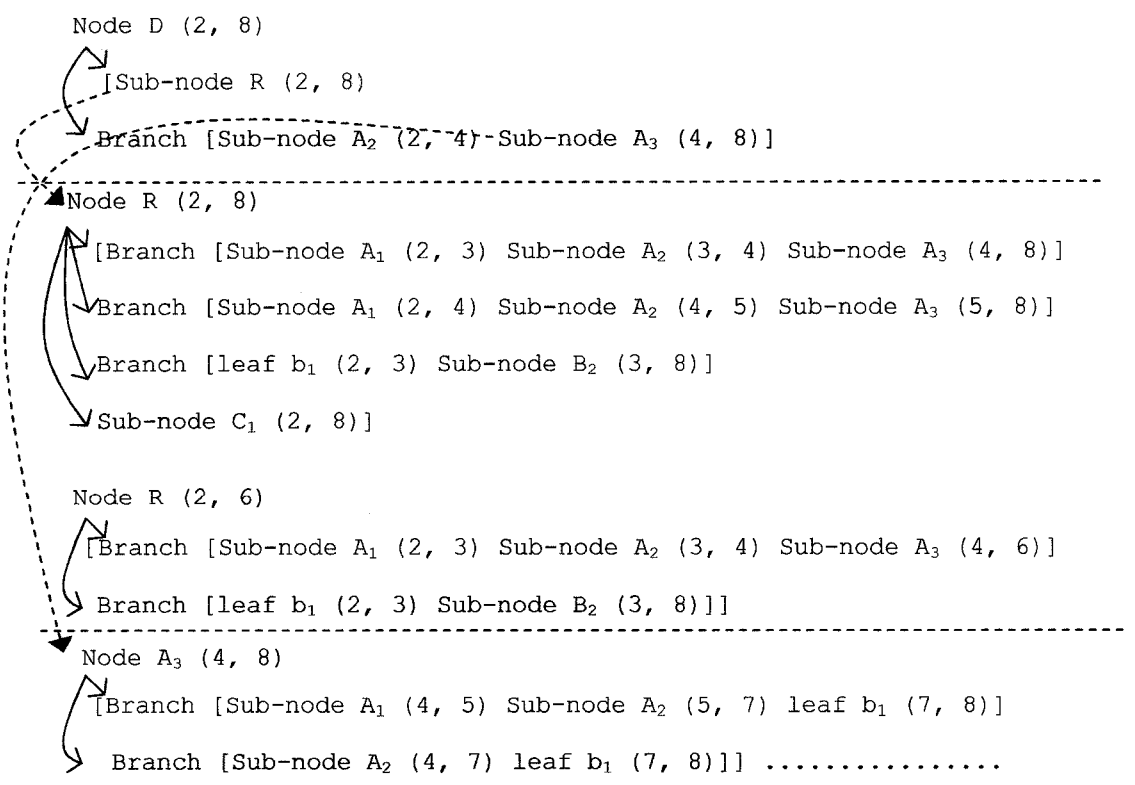


Figure 7.2: Example of a densely-compact representation

If it is required to retrieve parse-trees for start-position = 2 to end-position+1 = 8, one has to just follow the pointing notations of sub-nodes, from the root, until all leaves are reached. We shall discuss formally how this representation ensures polynomial space-requirement in chapter 11

7.3 The Modified Algorithm

In this section we modify the recognition-algorithm (described in chapter 6) step-by-step to accommodate parsing. We replace all occurrences of the term 'recognizer' of the last version of the algorithm with term 'parser'.

7.3.1 Type of Result and Memo-table

Instead of having only pairs of start-end positions as the results saved in memo-table, it is also required to save the n-ary sub-nodes (sequences of terminal/non-terminals) of one-level-depth branches for a node (non-terminal) at a specific start-end position. So the new memo-table type is:

```
memo-table = {( parser name, {(start-position,(left-
               rec-context[1], {result}))}}

result     = ((start-position, end-position + 1), {tree} )
tree       =  leaf or sub-node or branch
leaf       =  terminal name
sub-node   =  (non-terminal name, (start-position, end-
                                    position + 1) )
branch     = {tree}
```

A 'tree' can be considered as a 'name-less' leaf, sub-node or a set of leaves and/or sub-nodes (a branch), where each nodes have pointers to indicate 'where to go next'. The 'name' of a tree is added during the memoization process – as a part of the pointing-reference.

7.3.2. Modified Combinators

When a parser - constructed using an **alternative** combinator – is applied to the current start-position of the input, it may return multiple one-level-depth **trees** – indicating different or same ending-positions, which were computed using different alternatives. So the alternative combinator simply unites the returned **results** of the two operands of `'orelse'`, which is – as before – paired with the united **reasons** of two operands of

[1] If not pointed out, then any previously-mentioned procedure or data-type's definition remains same. For example, the type of 'left-rec-context' is same as section 6.4.2.

'orelse'. So the definition remains same for the alternative combinator – only the type of result has changed.

```
(p orelse4 q) i cc = (p orelse3 q) i cc
```

The **sequence** combinator 'then' is responsible for creating one-level-depth n-ary branches of the terminal and/or non-terminals (which are in a sequence) of an alternative-definition of a non-terminal. It ensures that all the sub-nodes of a branch are properly pointed (i.e. having appropriate start-end position). The left-operand of 'then', p, is applied on start-position i, it returns a result-set {((start-pos, end-pos+1), {tree})}. Then right-operand of 'then', q, is applied on every 'end-pos+1' returned by p. Every application of q returns a result-set {((start-pos, end-pos+1), {tree})}. As mentioned before, the type of tree is leaf or sub-node or branch. For every element '((start-pos_p, end-pos+1_p), {$tree_p$})' of the result-set of p and for each element '((start-pos_q, end-pos+1_q), {$tree_q$})' of the respective result-set of q, we form a new result '((start-pos_p, end-pos+1_q), name-less one-level-depth branches involving {$tree_p$} and {$tree_q$})'. At the end, all newly created one-level-depth n-ary branches are united and returned.

```
(p then4 q) i cc
    = (reason_p  ∪  reason_q, result_p-q)
      where
      (reason_q, result_p-q)
       =fold ( ⋃ over first and second set of the result-pair respectively ) ({},{})(map (q' cc) result_p)
        where
        q' cc (end-pos_of_p,tree_p)
          = (reason_q',create-branch (end-pos_of_p,
                                      tree_p) result_q')
            where
            (reason_p, result_p)     = p i cc
            {(end-pos_of_p,tree_p)} = result_p
            (reason_q', result_q')   = (q end-pos_of_p cc)
            where
            result_q'is of type
            {(end-pos_of_q,tree_q_at-end-pos_of_p)}
```

```
create-branch
Input : (end _ p, tp) set_tq
Method: let set_tp_tq = {}
        (∀(end _ q, tq)  ∈  set_tq)
          (form name-less branch as
           (tp's pointer as left-node,
            tq's pointer as right-node) and add
          ((end_p, end_q), (left-node tp, right-node tq)
              to set_tp_tq)
        return set_tp_tq
```

Parsers which consist of `term` and `empty` return a `tree` of type `leaf` and other functionalities remain same:

```
term4 i c cc = ({},{})                , if i > #input
             = ({},{((i, i + 1),{ leaf "c"})})
                                      , if token at
                                        position i == c
             = ({},{})                , otherwise

empty4 i cc  = ({}, {((i, i), {leaf "empty"})})
```

Note that, for all combinators, the functionalities related to 'context' and 'reason' remain unchanged.

7.3.3. Modified Memoization

When a memoized-parser (r_1) computes a 'result' using one of its alternatives (which may have sequences of terminals and/ or non-terminals) for the current start-position, the set of result may have multiple entries for an identical start-end position – due to ambiguous parsing. Each of these entries has the same start-end positions but will have different 'name-less' trees. As mentioned before, when this parser (r_1) is referred by another parser (r_2) for this same start-end position, r_2 needs to refer r_1 multiple times – for each ambiguous result individually. To avoid this extra space-requirement, we **group** these multiple ambiguous trees of r_1 as a 'set of trees' having only one start-end pointer. This grouping also unites the identical trees into one tree so that this single 'common-tree' can be shared by any parser that has a reference to it. Now r_2 needs to refer r_1 only once for this specific start-end position. This grouping takes place when a parser

computes a result for a specific start-position for the first time. The grouped ambiguous result-set is a set of 'name-less trees with pointers' and it is updated in the memo-table under current parser's name and start-position. During recursive-ascent, when current-parser r_1 passes the control to a preceding-parser (either after successfully performing a 'lookup' or freshly computing and updating a new result), it only requires to pass-up its own reference (pointer) of the memo-table entry, instead of returning the actual entry of the memo-table. In other words, for the other parsers to refer r_1's entry in the memo-table, it is required to **point** the r_1's computed set of result as r_1's **name and start-end position.** This requirement ensures the one-level-depth structure of the trees, which eventually reduces the space-requirement. Hence, during the creation of branches in the sequencing-combinator, it is only required to refer to this added 'pointing' node of the current parser, instead of the whole set of results. To accommodate these requirements, we modify the **memoize** procedure as follows:

```
memoize4
Input : parser name, parser (ri), start-position(j),
        current-context (cc)
Output: ((reasons,{((start-pos, end-pos),{tree})}), memo-
table)
Method: if lookup4 succeeds,
          return (create_pointer parser name memo-table
                  result)
        else
          if rij > #input - (j -1)
             return {( { parser-name}, {})}
          else
               increment rij counter by 1
               apply ri to j (that returns new_results)
               update4 memo-table (grouping_ambiguity
               new_results) with left-rec-context (if
               applicable)
        return ((reason, create_pointer parser name
                    new results), updated  memo-table)

grouping_ambiguity
Input : result-set = {(start-pos,end-pos+1), trees)}
Method: unite the trees of identical (start-pos,end-pos+1)
        pairs as a new set of trees under a single (start-
        pos,end-pos+1) pair.
        Return the altered result-set.
```

```
create_pointer
Input : parser name (p_name), result-set = {(start-pos,end-
                                            pos+1), trees),
Method: let pointer_set = {}
        (∀((s,e),t) ∈ result-set)
        add ((s,e),{sub-node (p_name, (s,e))} to
             pointer_set
        return pointer_set
```

lookup4 = lookup3, update4 = update3

The other functionalities (i.e. creating context and reason, comparing contexts w.r.t reason etc) remain the same in `memoization`, `update` and `lookup`, the only difference in new `update` and `lookup` procedure is the new type of the 'result'.

7.4 Memo-table as a Forest of n-ary Branches

For secured and correct operations on the memo-table, we use a state-monad to thread the memo-table within different recursive parser-calls (section 4.2.2 and 5.1.3).

According to the modified parsing-algorithm, an application of a memoized-parser

```
S = memoize4 "S" ((S then4 A) orelse4 empty4)
A = memoize4 "A" (S then4 (term4 a))
```

to the input-sequence "aaa" saves ambiguous and common 'results' under a parser's start-position entry as a set of n-ary branches of 'sub-trees', where each sub-node (represents another non-terminal) of a branch has appropriate pointers to its own entry in the memo-table (figure 7.3).

To retrieve a complete parse-tree (for example, based on some semantic-interpretations) one has to follow the directed-pointers of a node – starting from the root and continuing expanding from its left-most sub-node - until all 'leaves' are retrieved.

Start positions

Parser names

	1	2	3
S	S(1,1) | ε	S(2,2) | ε	S(3,3) | ε
	S(1,2) /\ S(1,1) A(1,2)		
	S(1,3) /\ S(1,1) A(1,3) S(1,3) /\ S(1,2) A(2,3)	S(2,3) /\ S(2,2) A(2,3)	
	S(1,4) /\ S(1,1) A(1,4) S(1,4) /\ S(1,2) A(2,4) S(1,4) /\ S(1,3) A(3,4)	S(2,4) /\ S(2,2) A(2,4) S(2,4) /\ S(2,3) A(3,4)	S(3,4) /\ S(3,3) A(3,4)
A	A(1,2) /\ S(1,1) a(1,2)	A(2,3) /\ S(2,2) a(2,3)	
	A(1,3) /\ S(1,2) a(2,3)	A(2,4) /\ S(2,3) a(3,4)	A(3,4) /\ S(3,3) a(3,4)
	A(1,4) /\ S(1,3) a(3,4)		

Figure 7.3: Memo-table represents results of parsing as a packed-forest

CHAPTER 8: IMPLEMENTATION IN HASKELL

Haskell has been used for the implementation of the algorithm described in the previous chapters. (See the [41] for basic notation of Haskell). The description in this section covers both recognition and parsing. The algorithm is implemented mostly in self-explanatory descriptive fashion.

8.1 Data-types and State-monadic Combinators

As discussed before, we have utilized Frost's (section 5.13) approach of constructing combinators using state-monads for secure threading of the memo-table. For convenience, we repeat the definition of state-monad according to the description of section 4.2 :

```
unitS :: t -> StateM t
unitS x = f where f t = (x,t)

bindS :: StateM t1 -> (t1 -> StateM t2) -> StateM t2
m `bindS` k = f
              where f x = (b,z)
                          where (b,z) = k a y
                                        where (a,y) = m x
```

The state or memo-table **Mtable** keeps a record of a parser's results at every start-position of the input. The general-term **Context** is a pair of **reason** for curtailing a left-recursive parser (a list of parser-names) and **left-recursive-context** – a list of (parser, parser's left-recursive count) pairs at different start-positions. An element of the list of **Result** consists of a pair of (start-position, end-position + 1) and a list of trees.

```
type State     = Mtable
type StateM t  = State -> (t, State)
type Mtable    = [(String,[(Int,(Context,[Result]))])]
type Context   = (Reason, Left-recursive-context)
type Reason    = [String]
type Left-recursive-context
               = [(Int,[(String, Int)])]
type Result    = ((Start, End),[Tree String])
```

We define a tree data-type utilizing Haskell's facility of constructing user-defined algebraic and recursive data-type. The data-type **Tree** could be either a **Leaf** (represents a terminal), a **SubNode** (represents a non-terminal with pointer – node name and start-end position in the memo-table) or a **Branch** (consists of a list of any form of trees) to represent sequencing.

```
data Tree a = Leaf a
            | Branch [Tree a]
            | SubNode (NodeName, (Start,End))
              deriving (Eq,Ord,Show)
type NodeName = String
type Start    = Int
type End      = Int
```

The combinator **empty** simply returns a tree of type `Leaf` – with the same start-end position. But combinator **term** checks whether the token at given input's start-position (**r**) has a match with its own. If so, then it returns a `Leaf` with **(r,r+1)** that makes the next parser move-ahead to parse next token. Either `empty` or `term` has no effect on the current descending `left-reccontext` **(l)** and none of them produce any `reason`.

```
empty x l = unitS (([],[]),[((x,x), [Leaf "empty"])])
term c r l |r - 1 == length input     = unitS (([],[]),[])
           |input !! (r - 1)  == c    =
                     unitS (([],[]),[((r,r+1),[Leaf [c]])])
           |otherwise                 = unitS (([],[]),[])
```

The **orelse** combinator individually applies parsers **p** and **q** to the given start-position **inp** and current-context **cc** and returns back the united `reasons` with `'union (fst l1) (fst l2)'` and summed `results` with `'(m ++ n)'`. Note that as both **l1** and **l2** are pairs of type `(Reason, Left-recursive-context)`, we just need to unite the `Reasons` to pass upwards. The **then** combinator first applies **p** to the given start-position `inp`, which returns a set of `results`. Then parser **q** is applied to every end-positions returned by `p` sequentially using **apply_to_all**.

```
(p `orelse` q) inp cc
     = p inp cc  `bindS` f
       where f (l1,m) = q inp cc  `bindS` g
                 where g (l2,n)  = unitS ((union (fst l1)
                       (fst l2),[]) ,(m ++ n))
 (p `thenS` q) inp cc
         = p inp cc `bindS` f
                   where f (l,m) = apply_to_all q m l cc
apply_to_all q [] l cc      = unitS ((fst l,[]),[])
apply_to_all q (r:rs) l cc
 = (q `add_P` (r,cc,l)) `bindS` f
    where f (l1,m)  = ((apply_to_all q rs l cc) `bindS` h)
          where h (l2,n)
          = unitS ((union (fst l1) (fst l2),[]),( m ++  n))
```

8.2 Forming 'name-less' n-ary branches for Parsers in Sequence

With **add_P** function of **apply_to_all**, the end-positions of the p's result-set are selected and individually passed to q for sequencing applications of q on them, which returns `(((s2,e2),t2):restQ)` on each application. At the end, `add_P` unites reasons of current p and current q and with **addp** function (of `add_P`), current result of `p - ((s1,e1),t1)-` creates **branches** with every results of q by executing **addToBranch** function . The `addToBranch` function creates sequencing results of p and q as `((p's start-position, q's end-position), p's result as left-node & q's result as right-node)`. At the end `apply_to_all` function unites all reasons with `'(union (fst l1) (fst 12),[])'` and sums all sequencing results with `'( m ++  n)'` of every q's application on p's result-set. Like `orelse`, only `reasons` for curtailments are united.

```
q `add_P` (rp,cc,l)
= (q (pickEnd rp) cc) `bindS` f
     where f (l1,m)
           = unitS ((union (fst l) (fst l1),[]),(addP m rp))

pickEnd ((s,e),t) = e -- selecting the end-position

addP [] ((s1,e1),t1)                    = []
addP (((s2,e2),t2):restQ) ((s1,e1),t1)
= ((s1,e2), addToBranch ((s2,e2),t2)  ((s1,e1),t1))
  : addP restQ ((s1,e1),t1)
```

For creating 'name-less' n-ary branching with **addToBranch** function, there could be 9 cases. If a branch already exists (either p or q's result), we just add the new candidate to the end. If two candidates are two branches, we append them. Otherwise, we form a new branch with non-branch candidates.

```
addToBranch ((st2,en2),((SubNode (name2,(s2,e2))):ts2))
            ((st1,en1),((SubNode (name1,(s1,e1))):ts1))
= [Branch [(SubNode (name1,(st1,en1))),(SubNode (name2,(st2,en2)))]]
addToBranch ((st2,en2),((Branch t2):ts2))
            ((st1,en1),((Branch t1):ts1))
= [Branch (t1++t2)]
addToBranch ((st2,en2),((Branch t2):ts))
            ((st1,en1),((SubNode (name1,(s1,e1))):ts1))
= [Branch ((SubNode (name1,(st1,en1))):t2)]
addToBranch ((st2,en2),((SubNode (name2,(s2,e2))):ts2))
            ((st1,en1),((Branch t1):ts))
= [Branch (t1++[(SubNode (name2,(st2,en2)))])]
addToBranch ((st2,en2),((SubNode (name2,(s2,e2))):ts2))
            ((st1, en1),[Leaf x])
= [Branch [(SubNode (("Leaf "++x), (st1,en1))),
           (SubNode (name2,(st2,en2)))]]
addToBranch ((st2,en2),[Leaf x])
            ((st1,en1),((SubNode (name1,(s1,e1))):ts1))
= [Branch [(SubNode (name1,(st1,en1))),
           (SubNode (("Leaf "++x),(st2,en2)))]]

addToBranch ((st2,en2),((Branch t2):ts)) ((st1,en1),[Leaf x])
= [Branch ((SubNode (("Leaf "++x),(st1,en1))):t2)]
addToBranch ((st2,en2),[Leaf x]) ((st1,en1),((Branch t1):ts))
= [Branch (t1++ [(SubNode (("Leaf "++x),(st2,en2)))])]
addToBranch ((st2,en2),[Leaf x2])((st1,en1),[Leaf x1])
 = [Branch [(SubNode (("Leaf "++x1),(st1,en1))),
            (SubNode (("Leaf "++x2),(st2,en2)))]]
```

8.3 Lookup, Update and Computing New-result using Memoization

When a memoized-parser is executed, the function **memoize** is applied with the parser-name (**name**), the parser-definition (**f**), starting input-position (**inp**), the descending-down context (**context**) and the initial memo-table (**mTable**).

```
memoize name f inp context mTable
| table_res /= []
  = ((fst1 table_res,(addNode name inp (snd1 table_res))), mTable)
| (funccount (snd context)) > ((length input) - (inp-1) )
  = ((([name],[]),[]), mTable)
| table_res == []
  = (( l1 ,(addNode name inp newRes)),udtTab)
            where
            fst1 [(a,b)] = a
            snd1 [(a,b)] = b
```

8.3.1 Lookup Operation

The `memoize` first looks in the memo-table to find whether there already exists a reusable-result for **`name`** at **`inp`** by checking the content of **`table_res`**, which in terns executes **`lookupT`** operation.

The **`lookupT`** fails if:

1. there is no entry for `name-inp` in `mTable (failure 1 & 2)`[1],

2. the saved-entry's reason (re in **`checkUsability`**) is not empty but left-rec-context is empty at `inp` (`failure 3`) or

3. some entries in saved left-rec-context are not present or have less left-rec-counter value (`failure 4 & 5`).

The **`lookupT`** succeeds if:

1. the saved-entry's reason (re in `checkUsability`) is empty (`success 1`),

2. the descending current-context or saved left-rec-context is empty (`success 2 & 3`) or

3. all members of saved left-rec-context exists in current left-rec-context and all of them have equal or greater number of left-rec-count (`towards_success 4 to 10`).

[1] All possible `failures` and `successes` are marked in the code of the next page

```
table_res   = lookupT name inp (snd context) mTable
lookupT name inp context mTable
| res_in_table == [] = []                                          --failure 1
| otherwise          = checkUsability inp
                       context (lookupRes (res_in_table !! 0) inp)
  where res_in_table = [pairs|(n,pairs) <- mTable,n == name]
lookupRes [] inp                        = []
lookupRes ((i,res):rs) inp | i == inp  = [res]
                           | otherwise = lookupRes rs inp

checkUsability inp context [] = []                                 --failure 2
checkUsability inp context [((re,sc),res)]
| re == []  =  [((re,sc),res)]                                     --success 1
| otherwise =  checkUsability_ (findInp inp context) (findInp inp sc)
               [((re,sc),res)]
findInp inp []                        = []
findInp inp ((s,c):sc) | s == inp  = c
                       | otherwise = findInp inp sc

checkUsability_ [] [] [(sc,res)]             = [(sc,res)]     --success 2
checkUsability_ ((n,cs):ccs) [] [(sc,res)]   = []             --failure 3
checkUsability_ [] ((n1,cs1):scs) [(sc,res)] = [(sc,res)]     --success 3
checkUsability_ ((n,cs):ccs) ((n1,cs1):scs) [(sc,res)]
| and (memCheck ((n,cs):ccs) ((n1,cs1):scs)) = [(sc,res)]
                                  --towards_success 4, if true for all
| otherwise = []                                                 --failure 4

memCheck [] ((n1,cs1):scs) = []                      -- towards_success 5
memCheck ((n,cs):ccs) ((n1,cs1):scs)
= condCheck (n,cs) ((n1,cs1):scs) ++ memCheck ccs ((n1,cs1):scs)

condCheck (n,cs) ((n1,cs1):scs)
| (notElemCheck (n,cs) ((n1,cs1):scs)) == [] = []    -- towards_success 6
| any (==(n,cs)) ((n1,cs1):scs)              = []    -- towards_success 7
| otherwise                                  = [False]       -- failure 5

notElemCheck (n,cs) []                        = []     -- towards_success 8
notElemCheck (n,cs) ((n1,cs1):scs) | n /=n1 = notElemCheck (n,cs) scs
                                                -- towards_success 9
                                   | otherwise = [False]
                                                -- towards_success 10
```

If the `lookupT` fails, `memoize` then checks the 'condition for curtailment' `'(funccount (snd context)) > ((length input) - (inp-1))'` in the descending-context.

```
funccount []                    = 0
funccount ((key,funcp):rest) | key == inp = findf funcp
                             | otherwise  = funccount rest
          where
          findf  []            = 0
          findf  ((tk,fc):rx) | tk == name = fc
                              | otherwise  = findf rx
((l,newRes),mft) = ((fst res,packAmb $ sort (snd res)),newtable)
 where
 (res, newtable)  = f inp ([],(incContext (snd context) name inp))mTable

incContext [] name inp  = [(inp,[(name,1)])]
incContext ((st,((n,c):nc)):sn) name inp
      | st == inp = ((st, (addNT ((n,c):nc)) name inp ) :sn)
      | otherwise = ((st,((n,c):nc)): incContext sn name inp )

addNT []  name inp                      = [(name,1)]
addNT ((n,c):nc) name inp  | n == name = ((n,(c + 1)):nc)
                           | otherwise = ((n,c):addNT nc name inp)
```

If the current parser is left-recursive and if the 'condition for curtailment' fails (3rd guarded condition of **memoize** function), then the left-rec-counter of the current parser is increased by one for the current starting position of the input. At this point the left-recursive parser starts recursively descending with `'f inp ([],(incContext (snd context) name inp))mTable'` until it satisfies the 'condition for curtailment'.

8.3.2 Update Operation

When the 'condition for curtailment' is satisfied, the left-recursive parser is curtailed (2nd guarded condition of **memoize** function) by adding its `name` to the 'reason for curtailment' and on the recursive-ascent, eventually computes a new result **(res, newtable)** for the current starting-position. This new-result **res** is added to the recent memo-table **newtable** with function **udtTab**. Before saving any result, we need to group and unite the ambiguous result-set – `res` (described next section) and also have to make sure that the correct left-rec-context is saved w.r.t the 'reasons for curtailment' `'(fst l)'` for `res`. With the function **makeContext**, we compare the current parser's descending current-context **(findContext (snd context))** at current start-position

`inp` with `(fst l)`. We only keep entries from the `(findContext (snd context))` which has a match in `(fst l)` and remove the other entries. These selected entries are then placed at the appropriate position (w.r.t `inp`) and paired with `(fst l)`. These operations are carried through **makeContext**, **makeContext_** and **makeContext__** functions. When the appropriate left-rec-context **l1** is created, it is paired with grouped and united result-set **newRes** and updated to the latest memo-table **mft** with function **udt**.

```
udtTab      = (udt ((l1,newRes),mft) name inp)

l1 =  makeContext (fst l) (findContext (snd context))
          where
          findContext []                              = []
          findContext ((st,rest):sr) | st == inp = [(st,rest)]
                                     | otherwise = findContext sr

     makeContext [] [(st,((n,c):ncs))]     = ([],[])
     makeContext (r:rs) []                 = ((r:rs),[])
     makeContext [] []                     = ([],[])
     makeContext (r:rs) [(st,((n,c):ncs))] = ((r:rs), [(st,makeContext_
                                              (r:rs) ((n,c):ncs))])

makeContext_ [] ((n,c):ncs)      = []
makeContext_ (r:rs) ((n,c):ncs)  = makeContext__ r ((n,c):ncs) ++
                                   makeContext_ rs ((n,c):ncs)

makeContext__ r [] = []
makeContext__ r ((n,c):ncs) | r == n     = (n,c): makeContext__ r ncs
                            | otherwise = makeContext__ r ncs

udt (res, mTable) name inp
              = update mTable name inp res

update [] name inp res                    = [(name,[(inp, res)])]
update ((key, pairs):rest) name inp res
               | key == name = (key,my_merge inp  res pairs):rest
               | otherwise   = ((key, pairs): update rest name inp res)


my_merge inp  res [] = [(inp, res)]
my_merge inp  res ((i, es):rest)
               |inp == i  = (i, res):rest
               |otherwise = (i, es): my_merge inp res rest
```

The update-function **udt** simply searches through the memo-table to find an entry for **name** at **inp**, and if there exits a previous result, **udt** replaces that with the new result. Otherwise **udt** creates a new entry for name at **inp** and places the new result in it. These operations are performed with **update** and **my_merge** respectively.

8.3.3 Grouping Ambiguities and Adding Pointers

When a memoized-parser creates a new-result, according to the algorithm, multiple trees (either identical or different) for a specific start-end position are grouped into a list of trees – with a single entry indicating the whole set's start-end position. The newly-created result **(snd res)** is first sorted[1] and passed to the function **packAmb,** which searches for the common start-end positions **(s1,e1) == (s2,e2)**, and if found, it then groups their respective results (which are name-less list of one-level-depth trees) together. If for identical start-end position, there exists some identical trees, they are also united into a single one – under a single start-end position – so that they can be shared by other parsers with a single reference. This grouped-result - **newRes** – is used in the update-operation, which was described in the last section.

```
{-- repeted segment of code - for convenience
((l,newRes),mft) = ((fst res,packAmb $ sort (snd res)),newtable)
where (res, newtable) = f inp ([],(incContext (snd context) name
inp))mTable --}

packAmb []                  = []
packAmb [((s1,e1),t1)]      = [((s1,e1),t1)]
packAmb [((s1,e1),t1),((s2,e2),t2)]
    | (s1,e1) == (s2,e2) = [((s2,e2), t1++t2)]
    | otherwise          = [((s1,e1),t1),((s2,e2),t2)]

packAmb (((s1,e1),t1):((s2,e2),t2):xs)
    | (s1,e1) == (s2,e2) = packAmb (((s2,e2), t1++t2):xs)
    | otherwise          = ((s1,e1),t1):packAmb (((s2,e2),t2):xs)
```

On ascending, the memoized-parser, which either computes a new result or successfully looks up a previous result, returns a pointer (consists of its name and the start-end position) to upwards – instead of returning the complete set of results. It does so by simply replacing every trees of the result-set with its name and start-end position – through the function **addNode**.

```
{-- repeted segment of code - for convenience
memoize name f inp context mTable
| table_res /= []
  = ((fst1 table_res,(addNode name inp (snd1 table_res))), mTable
.....................
| table_res == [] = (( l1 ,(addNode name inp newRes)),udtTab)--}

addNode name inp []  = []
addNode name inp (((s,e),t):rs)
     = ((s,e),[SubNode (name,(s,e))]):addNode name inp rs
```

[1] For sorting, we have used Haskell's library-function **sort** and its definition varies depending on the use of a particular interpreter or compiler i.e. Hug 98 uses a variation of inset-sort and GHCi uses stable quick-sort.

CHAPTER 9: TERMINATION ANALYSIS

9.1 Basic Concept

In the following, the terminations of recursively-defined procedures are described in terms of 'parsers' – which also justifies the terminations of recognizers, as recursively-defined recognizers and parsers have same number of recursive calls for a particular input. We discuss termination analysis of the algorithm by adopting a well-practiced technique for 'termination analysis of recursive functions' – where the central idea is to ensure that there exits a well-founded ordering so that the argument of each recursive call is 'smaller' (or 'greater') then the corresponding inputs. This comparison is done in terms of a 'measure' (an element of the well-founded set), which decreases (or increases) after each recursive-procedure execution. A 'measure-function' needs to be defined so that it can map a data-object (which is related to the corresponding recursive-function's input) to a member of a well-founded ordered set. For example, consider a recursive function definition:

$$f(x^1) = \ldots\ldots f(x^i)\ldots\ldots.$$

To show `f` terminates, the first task would be to define a measure function ($\| . \|$) that maps some type of data-object (in this example, 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[1] of decreasing 'measures' for all executions of `f` until $f(x^k)$, which is the last recursive call:

$$\|x^1\| > \ldots\ldots > \ldots\ldots \|x^i\| > \ldots\ldots\ldots > \|x^k\|$$

If the above inequality holds, then the function `f` terminates. The inequality could be formed the other way around too (based on the 'semantic' of the recursive function). The important property is that every two consecutive 'measures' must be related with a well-founded order. For example, in the case of the above inequality, $x^{i+1} \leq x^i$ holds for each pair of consecutive measures – that ensures termination. Giesl (1997, [15]) employed this basic approach to establish an automated termination-proof technique for

[1] In this case, the standard ordering $\leq$ of the Natural-number – that contains the 'least element' and ensures there exists no infinitely decreasing sequence of non-negative Integers.

nested and/or mutual recursive algorithms. But as we already know how our algorithm works (i.e. the semantics of the algorithm), we have the flexibility to prove the termination by following four 'general-steps' [15]:

1. Generating a measure-function $\| . \|$ and a well-founded ordering $\prec$
2. Generating an Induction Lemma `IL`: $\|x\| \prec \|g(x)\|$
3. Proving the Induction Lemma
4. Proving the inequality $\|x\| \prec \|x'\|$

The induction lemma is required for the recursive parsers of the form `f(t)` = ...`g(x)`...., where parsers `f` and `g` are both recursively defined (but can be different).

9.2 Cases for Combinatory-Parsers' Termination

Non-recursive basic parsers constructed with `term`, `empty` or other non-recursive parsers (constructed with `term` and `empty`) - terminate for a finite input in case of success or failure, as they are not recursively calling themselves or other functions again.

If a memoized nested and/ or mutually recursive parser (`p`) has a previously-computed re-usable entry in the memo-table for the current start-position (`j`), then instead of recursively descending, it simply retrieves the result and terminates (definition of `lookup` operation). If there is no entry in the memo-table, then the parser is bound to descend downwards and uses its alternatives to parse the current input. At this point, the following cases may occur:

Case 1: p is a non-left recursive parser.

a. If the memoized `p` fails to parse the input-token at `j` using all of its alternatives, then it terminates, without trying other sequential parsers of its alternatives (definition of `term`, `then` and `orelse`).

b. If memoized `p` parses the input-token at `j` successfully using any of its alternatives' first symbol, then the next parser (which could be recursive w.r.t `p`) is applied at start-position `(j+1)` as the input-token at `j` is consumed by now.

Case 2: p is a left-recursive parser.

Memoized p has to call itself (directly or indirectly) n (length of the input) times before determining a success or failure. This growth is tracked with a counter – `left-rec-counter`, which increases by one after each left-recursive call at same start-position. After descending down n times, the left-recurring branch is curtailed and the next alternative is being applied on the input-token at starting-position `j` (definition of `condition for curtailment`). If the next-available symbol consumes the current input, the start-position changes to `(j+1)` - indicating a success (hence, all subsequence parser-applications are on `(j+1)`). Otherwise, the alternative fails – indicating a failure.

A measure-function needs to be defined so that it can map the `start-position` and `left-rec-counter` (of each recursive call of a parser) to a natural-number (which is increased by at least one or remains the same after each recursive call) in order to form a well-founded order. From the above discussion, it is sufficient to show the termination of Case 1.b and Case 2 to prove that any recursively-defined parser terminates if it follows the algorithm described in chapter 6 and 7.

9.3 Proof of Termination

Definitions

9.1 The length of the finite sequence of input-tokens is $input_{\#}$.

9.2 P is a finite set of recursively-defined memoized parsers of size $P_{\#}$ which have been constructed by finite application of `empty`, `term`, `orelse`, and `then`. The members of P are denoted by p_i, $1 \le i \le P_{\#}$.

9.3 R is a finite set of `left-rec-counters`, the members of which are denoted by r_{ij} where $1 \le i \le P_{\#}$ and $1 \le j \le input_{\#}$. The counter r_{ij} represents the `left-rec-counter` for parser p_i applied to the input at the `start-position` given by the index `j`. The r_{ij}'s value is passed down only during the recursive-descent phase and

temporarily saved within the second element of `context` (section 6.4.2.1 and 8.1)[1]. For left-recursive parsers, r_{ij} is incremented by 1 during each recursive call up to $input_{\#}$ - (j -1) and j remains unchanged till this point. For non-left recursive parsers, r_{ij} stays to 1 and j is incremented by 1 after successfully parsing each input-token.

9.4 The measure-function $\| . \|$ maps a memoized recursive parser (p_i)'s input-argument (start-position (j), context, memo-table[2]) to a natural-number as follows:

$$\|j, context, memo\text{-}table\| = 0 \text{ , if context's second element doesn't have any entry for } p_i \text{ (i.e. } r_{ij} = null)$$
$$= r_{ij}, \text{ if } j = null$$
$$= j + r_{ij} \text{ , otherwise}$$

9.5 The well-founded order, $\prec$ is formed by relation $\leq$ on natural-numbers, which has the least element = 0 and greatest element = $input_{\#} + 1$.

Assumptions

9.1 All parser applications are memoized and the initial parser is applied to start-position, j = 1 with an empty context ({},{}) and an empty memo-table {}.

9.2 An application of p at (j context memo-table) returns (result, memo-table'), were result = {((start-position (j), end-position + 1 (j')),{Tree})}. The end-position, (j'-1) indicates how far the parser has parsed the input starting from start-position, j i.e. j' = start-position for next parser in sequence (if exists any).

9.3 All non-recursive parsers terminate. (as there is no recursion involved)

9.4 If a non-left recursive parser fails to parse an input-token then it terminates. (as there won't be a case to introduce any recursion and definition of term, then and orelse)

---

[1] The second-element of context has a type of {(start-position (j), {(parser-name (p_i), left-rec-count (r_{ij}))}}.

[2] Type of memo-table = {(parser-name,{(start-position,(Context',{Result}))}}) Context' = part of left-rec-context that has a match with 'reason' (section 6.4.2)

Lemmas

9.1 (∀start-position, j) $1 \leq j \leq input_{\#}$. It directly follows from the definition of term, which increases j by 1 until $input_{\#}$ upon each successful parsing.

9.2 (∀left-rec-counter, r_{ij}) $0 \leq r_{ij} \leq input_{\#} - (j - 1)$. It directly follows from the definition of memoization, which, according to condition of curtailment (section 6.3.1), increments p_i's r_{ij} by one if $r_{ij} \leq input_{\#} - (j - 1)$.

9.3 The measure-function $\| . \|$ ensures a well-founded ordering $\prec$ as it has minimum value = 0 (1st alternative of the definition of $\| . \| = 0$) and maximum value = $input_{\#} + 1$ (2nd alternative of the definition of $\| . \| = j + r_{ij} = input_{\#} + 1$ (definition 9.3, 9.4 and lemma 9.1, 9.2)).

Induction Lemma IL P

(∀$p_i \in P$)
memoize (p_i start-position (j) context memo-table) returns (result, memo-table') and the corresponding r_{ij} is updated in context' through memoization during recursive-descent
⇒
IL (1). result = {} ∨
((j ≤ ∀ j' ∈ map pick_2nd result) ∧ ($r_{ij} \in$ context (if any)
≤ $r'_{ij} \in$ context')).
[where pick_2nd (a, b) = b]

IL (2). map ($\|$start-position (j) context memo-table$\|\leq$)
(map ($\|.\|$ context' memo-table') map pick_2nd result)

IL (1). Proof by Induction on $P_{\#}$

Base Case

IL (1) P = {empty, term any} (definition of empty and term)

Hypothesis

Assuming IL (1) P = S is true.

Inductive step

Have to show IL (1) P = S ∪ {p_i}.

$(\exists\ p_1, p_2, p_3 \in S)(p_i$ = memoize "p_i" (p_1 then p_2) orelse p_3). The IL $\{p_i\}$ directly follows from base-case, hypothesis, definitions of then, orelse, memoization and definition 9.2, 9.3. In practice, a parser may be defined in terms of various combinations of other parsers using then and orelse combinators. But as the total number of parsers is constant (definition 9.2), this lemma still holds for any parser constructed over any combination of then and orelse. Also, from definition 9.3, assumption 9.2, lemma 9.1 and 9.2, this lemma is applicable for any left-recursive and non left-recursive parser. Hence, IL (1) $P = S \cup \{p_i\}$. □

IL (2). Direct Proof

$\|$start-position (j) context memo-table$\|$ = j + r'_{ij}. ..... b

Mapping $\|.\|$ context' and memo-table' to (map pick_2nd result) returns a set of values $B = \{(j'_1 + r'_{ij1}), (j'_2 + r'_{ij2}), \ldots\ldots, (j'_{input\#} + r'_{ijn})\}$ (definition of $\|.\|$). It follows from lemma 9.1, 9.2, induction lemma 1, definition 9.2 and 9.3 that $b \leq \forall b' \in B$. Hence mapping (b $\leq$) to B results

map ($\|$start-position (j) context memo-table$\|$ $\leq$)
(map ($\|.\|$ context' memo-table') map pick_2nd result). □

Proof of Termination

Theorem 9: We have to show that, using the induction-lemma, any recursively defined parser terminates.

Direct proof

Let a recursive parser $p_i \in P$ such that $p_i = p_{i'}$ then $p_{i''}$ and parsers p_i and $p_{i'}$ are applied on 'j context memo-table' (definition of then and semantically p_i inp = ($p_{i'}$ then $p_{i''}$) inp $\equiv$ p_i inp = $p_{i''}$($p_{i'}$ inp)). Suppose $p_{i'}$ returns (result, memo-table') and updates its left-rec-counter at context' during recursive-descent. According to IL (1) (which includes both left and non-left recursive parsers):

Case 1: result = {} i.e. $p_{i'}$ fails – that implies $p_{i'}$ and eventually p_i fails too (definition of then). From assumption 9.4, IL (2) and definition of $\|.\|$

$\| p_i$'s argument $\| \leq \| p_{i'}$'s argument $\| \leq \| p_{i''}$'s argument $\|$,

where minimum value of $\|.\| = 0$ (if $j = 1$ and r_{ij} = null) and maximum value of $\|.\| = 1$ (if p_i is non left-recursive) or $input_{\#} - (j - 1)$ (if p_i is left-recursive , lemma 9.2). Hence,

$\| p_i$'s argument $\| \prec \| p_{i'}$'s argument $\| \prec \| p_{i''}$'s argument $\|$

(definition 9.5 and lemma 9.3) .

Case 2: result $\neq$ {} i.e. $p_{i'}$ succeeds. From the definition of then, $\prec$ and $\|.\|$, $\| p_i$'s argument $\| \prec \| p_{i'}$'s argument $\|$

$\Leftrightarrow (j + r_{ij}) \prec (j + r_{ij} + 1)$ c

As $p_{i'}$ succeeds, $p_{i''}$ is applied to $\forall j' \in$ (map pick_2nd result) with memo-table' (definition of then). According to IL (2) , definition of $\|.\|$, the following is true for

$\forall j' \in$ (map pick_2nd result):

map ($\|j'$ context' memo-table' $\| \leq$)
(map ($\|.\|$ context' memo-table') map pick_2nd result)
$\Rightarrow$
map (($j' + r_{ij'}$) $\leq$)
(map ($\|.\|$ context' ($p_{i''}$' argument)))
$\Rightarrow$
map (($j' + r_{ij'}$) $\leq$) $\|(p_{i''}$' argument)$\|$ ..... d

But as $p_{i'}$ succeeds, $j \leq \forall j'$ and $(r_{ij} + 1) \leq \forall r_{ij'}$ (lemma 9.1 and 9.2).

Therefore, from c and d, lemma 9.3, definition of $\|.\|$ and $\prec$:

$(j + r_{ij}) \prec (j + r_{ij} + 1) \prec \|(p_{i''}$' argument)$\|$

$\Rightarrow$

$\|p_i$'s argument $\| \prec \| p_{i'}$'s argument $\| \prec \| p_{i''}$'s argument $\|$

Well-founded order of any number of parser-sequencing of p_i with then can be shown according to the above argument and if p_i has more then one alternative, all of their individual termination ensures p_i's complete termination.

Hence, all recursive parsers terminate. □

CHAPTER 10: COMPLEXITY ANALYSIS

10.1 Time Complexity of Recognition – w.r.t the length of input

In this section, we show that the worst-case complexity of recognizing an input-sequence (of length n) is $O(n^3)$ for a non-left recursive recognizer and $O(n^4)$ for a left-recursive recognizer - w.r.t n. The complexities of individual building blocks are analyzed first in order to prove the complexity of a complete recognizer (proof by construction).

Assumptions

10.1 R = non-terminals, nts ∪ terminals, ts - is a finite set of recognizers of a given grammar and size of this set is $R_\#$. $\forall r_i \in$ nts applications are memoized and the initial recognizer is applied on (start-position j = 1, context ({},{}), memo-table {}). An application of a recognizer returns (result, memo-table'), were result = {(start-position (j), end-position + 1 (j'))}, where j' = start-position for next recognizer in sequence (if exists any). On ascending, this result is paired with a set of reasons for curtailment (first element of context), if any (section 6.4.2).

10.2 For recognition, size of the memo-table = $R_\#$*n*n = $O(n^2)$ and size of the second element of context = n*$R_\#$ = O(n) (definition of memo-table and context for recognition (section 6.4.2)) .

10.3 The following operations have constant time-complexity:
comparison of two values, extracting a value from a tuple, adding an element to the front of a list and retrieving i^{th} value from a list whose length depends on $R_\#$ not on n.

Lemmas

10.1 Merging two result-sets, curtailment-condition check, incrementing left-rec-counter requires O(n) time

Follows from the definition of (++) – which is the only operation used for merging result-sets and from assumption 10.2 (as curtailment-condition check and incrementing left-rec-counter are performed on second element of context).

10.2 Operations related to manipulating context and reason need O(n) time

According to section 6.4.2, forming the left-rec-context, comparison between left-rec-context and current-context etc. take place at start-position j of each context's second element and actual operations are dependent on $R_{\#}$. Also, creating reason for curtailment is independent of n too. Hence, time required for manipulating context and reason is O(n).

10.3 Basic Recognizers require O(n) time

Recognizers constructed with **term** require O(n) time at the worst case as the start-position j could be the last index of the input (definition of term) and recognizers constructed with **empty** need O(1) time as its only purpose is to return {(j, j)} (definition of empty).

10.4 Memo-table update and lookup require O(n) time

The **lookup** requires a search for the current recognizer's set of saved results (which is paired with reason and left-rec-context, if any) at the current start-position j in the memo-table of size $O(n^2)$ (assumption 10.2), which needs O(n) time. Then lookup performs the re-usability test by comparing left-rec-context with current-context w.r.t reason, if any (section 6.4.2). These operations are sequential linear operations w.r.t the length of input n (lemma 10.2). Therefore the worst-case complexity remains O(n).

The **update** operation constructs appropriate left-rec-context with O(n) time (lemma 10.2) and saves the newly computed result by replacing the old result in the memo-table (it does so instead of merging so that there exists no duplicates), which requires a search for the current recognizer and the current start-position j by spending O(n) time (section 6.4.2). Hence the worst-case complexity remains O(n).

10.5 Recognizer with Alteration requires O(n) time

Application of memoize (r_p orelse r_q) at start-position j involves the following steps (assuming recognizers r_p and r_q had already been applied on j and their results are available):

1. One memo-table lookup – 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 and reason returned by r_p and r_q – requires O(n) (merging reasons depends on $R_\#$)
        2.2.3 Updating the new result to memo-table – requires O(n)

All the above time complexities follow from lemma 10.1 to 10.4. Hence, the worst case complexity remains to O(n).

10.6 Recognizer with Sequencing requires $O(n^2)$ time

In case of memoize (r_p then r_q) at start-position j, at worst-case r_p may returns a set of results of length n and according to the definition of then, r_q has to be applied to every (end-position +1) of r_p's result-set. Application of memoize (r_p then r_q) at start-position j involves following steps (assuming recognizers r_p and r_q had already been applied on j and $\forall j' \in$ (map pick_2^{nd} r_p's result-set) respectively and their results are available):

1. One memo-table lookup – 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 r_q on $\forall j' \in$ (map pick_2^{nd} r_p's result-set) and merging their results and reasons to form new result – requires O(n*n) = $O(n^2)$
        2.2.2 Updating the new result to memo-table – requires O(n)

All the above time complexities follow from lemma 10.1 to 10.4. Hence, the worst case complexity remains to $O(n^2)$.

Theorem 10.1

Non-left recursive recognizers require $O(n^3)$ time at the worst case.

Direct proof:

Given an input of length n and a recognizer-set (grammar) R of size $R_\#$, each non-left recognizer, $r \in R$ is applied to a particular start-position $j \in n$ at most once, as at least one left-most input-token of current input would be consumed before recursive execution of r again.

..... **a**

In practice, a recognizer may have multiple combinations of `then` and `orelse` to form a bigger recognizer. Multiple occurrences of `then` in a recognizer-definition $(r_1$ `then` r_2 `then`$\ldots . r_i)$ doesn't change the time complexity $O(n^2)$ of lemma 10.6 because each subsequent recognizers $(r_2 \ldots . r_i)$ can be applied sequentially to at most n start-positions and this cost of time depends on $R_\#$ not on n. Also multiple occurrences of `orelse` in a recognizer-definition $(r_1$ `orelse` r_2 `orelse`$\ldots . r_i)$ maintains time complexity $O(n)$ of lemma 10.5 as all alternative recognizers are applied sequentially to a same start-position and their underlying number of computations depend on the number of alternatives not on n. Therefore, irrespective of how many times `then` and `orelse` combinators have been used in a recognizer-definition, it's worst-case time complexity would be $O(n^2)$ when applied to one input (from lemma 10.3, 10.5 and 10.6).

..... **b**

Hence, from **a** and **b**, worst-cast time complexity of a non-left recursive recognizer = $n * O(n^2) = O(n^3)$. □

Theorem 10.2

Left recursive recognizers require $O(n^4)$ time at the worst case.
Direct proof:
Given an input of length n and a recognizer-set (grammar) R of size $R_{\#}$, each **direct left-recognizer**, r is applied to a particular start-position $j \in n$ at most n times – follows from the definition of 'condition for curtailment' (section 6.3.1).

..... **a'**

If r is an **indirect left-recursive recognizer** and its lookup fails due to re-usability checking then at the very worst-case r may be applied to any $j \in n$ at most $n*nt_{\#}$ times, where $nt_{\#}$ = is the number of non-terminals in R (section 6.4.2).

..... **a''**

This worst-case may happen when every nt of R is involved within the path of a indirect left-recursive recognizer towards its recursive call.
Hence, from **a'**, **a''** and **b** (of theorem 10.1), worst-cast time complexity of a left recognizer = $nt_{\#}*n*n*O(n^2) = O(n^4)$. □
It follows from lemma 10.3, theorem 10.1 and theorem 10.2 that $\forall r \in R$ terminates. □

10.2 Time Complexity of Parsing – w.r.t the length of input

We gradually show that worst-case time complexity of parsing an input-sequence (of length n) is $O(n^3)$ for a non-left recursive parser and $O(n^4)$ for a left-recursive parser w.r.t n – length of the input.

Assumptions

10.4 P = non-terminals, nts ∪ terminals, ts – is a finite set of parsers of a given grammar and size of this set is $P_{\#}$. $\forall p_i \in$ nts applications are memoized and the initial parser is applied on (start-position j = 1, context ([],[]), memo-table []). An application of a parser returns (result, memo-table'), were result = {((start-position (j), end-position + 1 (j')), {tree[1]})},

[1] From section 7.3.1
tree = leaf *or* sub-node *or* branch
leaf = terminal name
sub-node = (non-terminal name, (start-position, end-position + 1))

where `j'` = `start-position` for next recognizer in sequence (if exists any). On ascending, this result is paired with a set of `reason for curtailment` (first element of `context`), if any (section 6.4.2).

10.5 For parsing, size of the `memo-table` increases by a factor of `n` = $R_{\#}$`*n*n*n` = $O(n^3)$ and size of the second element of `context` = `n*`$R_{\#}$ = `O(n)` (definition of `memo-table` and `context` for parsing (section 7.3.1)) .

Assumption 10.3 and Lemma 10.1, 10.2, 10.3 and 10.4 remain unchanged except the term `parser` replaces the term `recognizer`.

Lemma 10.7 Functionalities for 'creating pointers' and 'grouping ambiguity' need $O(n)$ and $O(n^2)$ time respectively

From the discussion of 'modified memoization' of section 7.3.2 and 7.3.3, the following two sequential operations may need to be performed:

1. A pointer is being created for a set of results of the current parser at current start-position that refers to the actual set of results in the memo-table. It basically involves searching the result-set, which requires `O(n)` time (definition of `create_pointer`).
2. Grouping ambiguity involves uniting the trees of identical (start-pos, end-pos+1) pairs as a new set of trees under a single (start-pos, end-pos+1) pair. According to current implementation of `group_ambiguity` (section 8.3.2), the new-results are sorted first before the actual grouping (that requires `O(n)` time) takes place. As the library function 'sort' is interpreter/ compiler dependent (i.e. worst case is $O(n^2)$), the overall worst-case complexity for ambiguity-grouping is $O(n^2)$.

Lemma 10.8 Creating n-ary branches requires $O(n)$ time

From the discussion of 'modified combinators' of section 7.3, creation of name-less n-ary branches between a single pointer and a set of pointers (of length `O(n)`) requires `O(n)` time in the worst-case.

Lemma 10.9 Parsers with Alternation requires $O(n^2)$ time

Application of `memoize` (p_p `orelse` p_q) at `start-position j` involves following steps (assuming parsers p_p and p_q had 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 and reason returned by p_p and p_q – requires $O(n)$ (merging reasons depends on $R_\#$)
        2.2.3 Ambiguity packing of new result + updating the packed result to memo-table + create pointer – requires $O(n^2)$

All the above time complexities follow from lemma 10.1 to 10.4 and 10.7. Hence, the worst case complexity remains at $O(n^2)$.

Lemma 10.10 Parser with Sequencing requires $O(n^2)$ time

In case of `memoize` (p_p `then` p_q) at `start-position j`, at worst-case p_p may returns a set of results of length `n` and according to the definition of `then` (of section 6.3.3), p_q has to be applied on every (end-position +1) of p_p's result-set and each pointers of p_p's result-set needs to create n-ary branch with pointer-set returned by p_q's application on each (end-position + 1) of p_p. Application of `memoize` (p_p `then` p_q) at `start-position j` involves following steps (assuming p_p and p_q had already been applied on `j` and $\forall j' \in$ (map `pick_2nd` r_p's result-set) respectively 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 Application of p_q on $\forall j' \in$ (map `pick_2nd` p_p's result-set) + forming n-ary branching between each pointer of p_p with corresponding

pointer-set of p_q on current `j'` + merging their results and reasons to form new result – requires $O(n*n) = O(n^2)$

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

All the above time complexities follow from lemma 10.1 to 10.4, 10.7 and 10.8. Hence, the worst case complexity remains at $O(n^2)$.

Applying same arguments of theorem 10.1 and 10.2, we can conclude that a non-left recursive parse and a left-recursive parser require $O(n^3)$ and $O(n^4)$ time respectively. □

10.3 Space Complexity – w.r.t the length of input

According to section 6.4.2, the memo-table used for **recognition** is of type `{(recognizer-name,{(start-position,(left-rec-context,{(start-position, end-position + 1)}))}}`. As described in section 6.4.3 and shown in figure 6.4, each recognizer has at most `n` entries and each of these entries may have at most a result-set of size `n`. So the size of the final memo-table would be $O(n^2)$ after complete recognition.

Similarly, according to section 7.3.1, the memo-table used for **parsing** is of type `{(parser-name,{(start-position,(left-rec-context, {result}))}}`, `result` is of type `(start-position, end-position + 1), {tree})`. As described in section 7.4 and shown in figure 7.3, each parser has at most `n` memo-table entries and each of them has a result-set of size at most n. But each entry of result-set can be paired with a tree of size at most `n * k` (where `k` is a constant that depends on number of symbols - `r` on the right-hand side of a rule). If the grammar is in Chomsky Normal Form (i.e. `r = 2`) or `r ≥ n` then `k = 1`. The reason is if `r = 2` or `r ≥ n` then there could be at most `n` number of ambiguous results (branches) for a particular start/ end pair. Hence, the size of the final memo-table would be $O(n^3)$ (if `r = 2` or `r ≥ n`) after complete parsing.

An Analysis for polynomial space requirement due to ambiguity-packing

Consider the grammar

```
S = memoize4 "S" ((S then4 A) orelse4 empty4)
A = memoize4 "A" (S then4 (term4 s))
```

Say for start-position 1 and end-position 4 on input "ssss", S creates 4 different parses.

Pointers with one-level depth branches but without ambiguity grouping, there would be 4 different memo-table entries for S:

```
"S", 1, {((1,4), tree1),
         ((1,4), tree2),
         ((1,4), tree3),
         ((1,4), tree4)}
```

If A is to refer S (1, 4) for its parses, then A creates 4 different entries in the memo-table for S:

```
"A", 1, {((1,4), S(1,4)..),
         ((1,4), S(1,4)..),
         ((1,4), S(1,4)..),
         ((1,4), S(1,4)..)}
```

If A (1,4) if needed to be referred by S again somewhere in the parse, each 4 of A's entries has to be added to S's list. For total 64 complete parse-trees, space requirement is 12 cells – still compact

With ambiguity grouping, S's 4 ambiguous results are grouped together in a single list:

```
"S", 1, {((1,4), {tree1, tree2, tree3, tree4})}
```

Now A needs to create only one entry that refers to S's memo-table entry (1, 4) cell:

```
"A", 1, {((1,4), S(1,4).. )}
```

S now can now refer to A by adding only one entry to its list:

```
"S", 1, {((1,4), A(1,4).. )}
```

Now, for 64 parse-trees space requirement is only 6 cells – densely compact. It is needless to say that if there was no one-level depth pointing branches and no ambiguity-grouping, it would take 64 cells in the memo-table.

(The example 5 of the appendix contains the actual result)

CHAPTER 11: EXPERIMENTAL RESULTS

To justify the complexity-analysis of the previous chapter, we have tested different versions of a highly ambiguous grammar (`S :: S S 's' | ε`) by applying our implemented algorithm (chapter 8) on various lengths of inputs (n). According to Aho and Ullman [1]'s equation $\binom{2n}{n} / (n+1)$, the above grammar generates enormous number of parses, for example:

Length of input, n	No of parses
3	5
6	132
12	20,812
24	128,990,414,734
48	1.313278982422e+26

We have used four different parsers - representing four versions of the above grammar:

1. Un-memoized non-left recursive parser

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

2. Memoized non-left recursive parser (example 2 of the appendix)

```
s1 = memoize "s1" (((term 's') `thenS` s1 `thenS` s1) `orelse` empty)
```

3. Memoized left-recursive parser (example 3 of the appendix)

```
s2 = memoize "s2" (s2 `thenS` s2 `thenS` (term 's') `orelse` empty)
```

4. Memoized left recursive parser in CNF[1] (example 4 of the appendix)

As the algorithm is not restricted to only CNF, we memoize every components of the previous parser to represent it in a CNF. This is only possible because of the modularity of combinator-parsers.

```
s3 = memoize "s3" (s3 `thenS` memoize "s3'"  (s3 `thenS` (term 's'))
                  `orelse` empty)
```

Parsers `s2` and `s3` are equivalent, according [1].

It is worth mentioning that any practical grammar for a Natural Language would be much less ambiguous than the above grammars.

[1] In a CNF grammar, each rule has at most two symbols in sequence for each alternative.

We have collected 'number of seconds' and 'number of reductions' required for generating compact-representation of packed forest per input-length – using built-in functionalities of GHCi[1] and Hugs'98[2]. The experiments were performed on a PC with 0.5 GB of RAM and the results are listed in the following tables:

n = No of 's' in input	Parser s time required (using GHCi)	No of reduction (using Hugs)	n^x where $x = \log_n$ (no of reductions)
3	0.05 secs	14470	8.719939617
6	1.22 secs	627678	7.450655517
12	1006.27 secs	(out of space)	

Table 11.1: Time and no of reductions for parser s

n = No of 's' in input	Parser s1 time required (using GHCi)	No of reduction (using Hugs)	n^x where $x = \log_n$ (no of reductions)
3	0.02 secs	7407	8.11039606
6	0.15 secs	36415	5.861688601
9	0.32 secs	106899	5.270121162
12	0.52 secs	240206	4.985801848
15	1.07 secs	457662	4.813014985
24	4.24 secs	1847653	4.540334278
30	7.66 secs	3628761	4.440907166
35	13.31 secs	5825128	4.381481436
40	20.96 secs	8769200	4.333770279
48	32.65 secs	(out of space)	

Table 11.2: Time and no of reductions for parser s1

n = No of 's' in input	Parser s2 time required (using GHCi)	No of reduction (using Hugs)	n^x where $x = \log_n$ (no of reductions)
3	0.07 secs	12188	8.563719191
6	0.20 secs	102908	6.441484397
9	0.33 secs	486526	5.9598121
12	0.80 secs	1613858	5.752384358
15	1.38 secs	(out of space)	
24	5.84 secs		
30	13.30 secs		
35	24.02 secs		
40	45.91 secs		

Table 11.3: Time and no of reductions for parser s2

[1] Glasgow Haskell Compiler is the most widely used standard compiler for Haskell. `www.haskell.org/ghc`

[2] Hugs' 98 is a standard interpreter for Haskell. `www.haskell.org/hugs`

n = No of 's' in input	Parser s3 time required (using GHCi)	No of reduction (using Hugs)	n^x where $x = \log_n$ (no of reductions)
3	0.08 secs	14483	8.720757018
6	0.18 secs	83260	6.323239093
9	0.41 secs	301298	5.741723089
12	0.71 secs	831423	5.485475268
15	1.17 secs	1880703	5.334892313
24	4.28 secs	11761465	5.122738612
30	8.88 secs	28636547	5.048279297
35	16.12 secs	(out of space)	
40	23.62 secs		
48	68.21 secs		

Table 11.4: Time and no of reductions for parser s3

From the above results, it is evident that the required time (using GHCi) increases in a polynomial-rate (except the un-memoized parser, `s` - which fails, after exhibiting exponential behavior, at `n=12`). It also suggests that, though parsers `s2` (non-CNF) and `s3` (CNF) are equivalent left-recursive parsers, time-requirements of `s3` is much less (which is almost equivalent to memoized non-left recursive parser's time requirements) then `s2`. As Hugs 98 allocates less memory then GHCi for each session it runs out-of-space much quicker then GHCi. Even though Hug 98's 'number of reduction count' is a rough-measure, it also suggests polynomial nature of the memoized parsers.

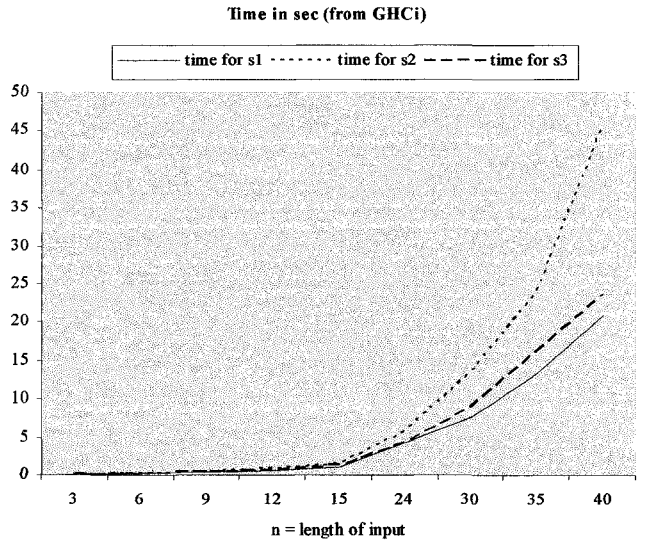


Figure 11.1: time vs. length-of-input plot for memoized parsers

CHAPTER 12: CONCLUSION

In this report we have proven the thesis-statement (section 2.4.2) by proposing a new algorithm, analyzing its termination and complexity, implementing it in Haskell and performing experiments on different Context-Free Grammars with variable length of inputs. The experimental results of chapter 11 (that include highly-ambiguous left-recursive grammar) suggest that the polynomial nature of the algorithm is correct as proven in chapter 10. It is also evident that memoizing all component-parsers of a bigger and highly ambiguous left-recursive parser requires almost the same time to generate a complete parse-forest as it would require for a memoized non-left recursive parser. Our experiments were not only restricted to proper and CNF grammars, but included grammars with cyclic and empty rules. Though monadic facilities and the lazy-evaluation of Haskell assisted to propagate the memo-table systematically, to share computed values within different recursive calls, and facilitated the construction of the compact-representation of parse trees, the described algorithm can be implemented using other programming languages which support recursion and dynamic data-structures.

Future works related to this algorithm includes:

- Analyzing the time and space complexity w.r.t variable length of grammars.
- Improving the Haskell-code by following conventions according to the existing libraries of Haskell and by accommodating 'user-supplied' input for more general use.
- Investigating the use of advanced programming techniques, such as those proposed in [21] to optimize the implementation of the parser combinators.
- Investigating the use of arrays instead of lists to access and search the memo-table and to group ambiguous parses much faster.
- Testing the algorithm on bigger and more practical NL grammars.
- Analyzing extraction-time of a complete parse from the compact representation w.r.t variable length of input and variable size of the grammar.

APPENDIX: EXPERIMENTAL OUTPUT OF COMPACT REPRESENTATIONS

The Haskell implementation of the algorithm has been applied on different Context-Free grammars with variable lengths of input. Some of those sample applications are listed below:

Example 1: CFG for Natural Language

Following is a grammar (similar to the one mentioned in Tomita's paper [35]) that defines a subset of English and its equivalent combinator-parser – written according to the algorithm of this report:

NL grammar

```
S  ::= NP VP | S PP          det  ::= 'a' | 't'
NP ::= n| det n| NP PP  noun ::= 'i' | 'm' | 'p' | 'b'
PP ::= prep NP               verb ::= 's'
VP ::= v NP                  prep ::= 'n' | 'w'
```

where `S` stands for sentence, `NP` for noun-phrase, `VP` for verb-phrase, `PP` for prepositional-phrase, `prep` for preposition, and `det` for determiner.

Equivalent combinator-parser

```
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 original grammar and the equivalent combinator-parser are structurally closely related and each of the sub combinator-parsers can be tested individually. An application of the root combinator-parser `s` on input `"isamntpwab"` (which represents `"i saw a man in the park with a bat"`) results following result, in which exponential numbers of parse-trees are represented in a polynomial space as a forest of one level depth n-ary branches:

```
apply s at start-position 1 of input "isamntpwab" with empty comtext and empty memo-table
⇒
"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 (["np"],[(1,[])])
       ((1,2),[SubNode ("noun",(1,2))])
     3 (["np"],[])
       ((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 (["np"],[])
      ((6,8),[Branch [SubNode ("det",(6,7)),SubNode ("noun",(7,8))]])
      ((6,11),[Branch [SubNode ("np",(6,8)),SubNode ("pp",(8,11))]])
    9 (["np"],[])
      ((9,11),[Branch [SubNode ("det",(9,10)),SubNode "noun",(10,11))]])
"prep" 5 ((5,6),[Leaf "n"])
       8 ((8,9),[Leaf "w"])

"pp" 8 (["np"],[])
       ((8,11),[Branch [SubNode ("prep",(8,9)),SubNode ("np",(9,11))]])
     5 (["np"],[])
       ((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 (["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))]])
"s" 1 (["np","s"],[])
      ((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))]])
```

Example 2: Highly ambiguous non-left recursive CFG

The following is a highly ambiguous non-left recursive grammar.

Original CFG

```
S::= 's' S S | ε
```

Equivalent combinator-parser

```
s = memoize "s" (((term 's') `thenS` s `thenS` s) `orelse` empty)
input = "ssss"

apply s at start-position 1 of input "ssss" with empty comtext and empty memo-table
⇒
"s" 5 ((5,5),[Leaf "empty"])
    4 ((4,4),[Leaf "empty"])
      ((4,5),[Branch [SubNode ("Leaf s",(4,5)),SubNode ("s",(5,5)),SubNode ("s",(5,5))]])
    3 ((3,3),[Leaf "empty"])
      ((3,4),[Branch [SubNode ("Leaf s",(3,4)),SubNode ("s",(4,4)),SubNode ("s",(4,4))]])
      ((3,5),[Branch [SubNode ("Leaf s",(3,4)),SubNode ("s",(4,4)),SubNode ("s",(4,5))],
              Branch [SubNode ("Leaf s",(3,4)),SubNode ("s",(4,5)),SubNode ("s",(5,5))]])
    2 ((2,2),[Leaf "empty"])
      ((2,3),[Branch [SubNode ("Leaf s",(2,3)),SubNode ("s",(3,3)),SubNode ("s",(3,3))]])
      ((2,4),[Branch [SubNode ("Leaf s",(2,3)),SubNode ("s",(3,3)),SubNode ("s",(3,4))],
              Branch [SubNode ("Leaf s",(2,3)),SubNode ("s",(3,4)),SubNode ("s",(4,4))]])
      ((2,5),[Branch [SubNode ("Leaf s",(2,3)),SubNode ("s",(3,3)),SubNode ("s",(3,5))],
              Branch [SubNode ("Leaf s",(2,3)),SubNode ("s",(3,4)),SubNode ("s",(4,5))],
              Branch [SubNode ("Leaf s",(2,3)),SubNode ("s",(3,5)),SubNode ("s",(5,5))]])
    1 ((1,1),[Leaf "empty"])
      ((1,2),[Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,2)),SubNode ("s",(2,2))]])
      ((1,3),[Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,2)),SubNode ("s",(2,3))],
              Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,3)),SubNode ("s",(3,3))]])
      ((1,4),[Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,2)),SubNode ("s",(2,4))],
              Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,3)),SubNode ("s",(3,4))],
              Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,4)),SubNode ("s",(4,4))]])
      ((1,5),[Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,2)),SubNode ("s",(2,5))],
              Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,3)),SubNode ("s",(3,5))],
              Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,4)),SubNode ("s",(4,5))],
              Branch [SubNode ("Leaf s",(1,2)),SubNode ("s",(2,5)),SubNode ("s",(5,5))]])
```

Example 3: Highly ambiguous left-recursive CFG

The following example is the equivalent highly ambiguous left-recursive version of the grammar from example 2.

Original CFG

```
S::= S S 's' | ε
```

Equivalent combinator-parser

```
s = memoize "s" (s `thenS` s `thenS` (term 's') `orelse` empty)
input = "ssss"

apply s at start-position 1 of input "ssss" with empty comtext and empty memo-table
⇒
"s" 1 (["s"],[])
      ((1,1),[Leaf "empty"])
      ((1,2),[Branch [SubNode ("s",(1,1)),SubNode ("s",(1,1)),SubNode ("Leaf s",(1,2))]])
      ((1,3),[Branch [SubNode ("s",(1,1)),SubNode ("s",(1,2)),SubNode ("Leaf s",(2,3))],
              Branch [SubNode ("s",(1,2)),SubNode ("s",(2,2)),SubNode ("Leaf s",(2,3))]])
      ((1,4),[Branch [SubNode ("s",(1,1)),SubNode ("s",(1,3)),SubNode ("Leaf s",(3,4))],
              Branch [SubNode ("s",(1,2)),SubNode ("s",(2,3)),SubNode ("Leaf s",(3,4))],
              Branch [SubNode ("s",(1,3)),SubNode ("s",(3,3)),SubNode ("Leaf s",(3,4))]])
      ((1,5),[Branch [SubNode ("s",(1,1)),SubNode ("s",(1,4)),SubNode ("Leaf s",(4,5))],
              Branch [SubNode ("s",(1,2)),SubNode ("s",(2,4)),SubNode ("Leaf s",(4,5))],
              Branch [SubNode ("s",(1,3)),SubNode ("s",(3,4)),SubNode ("Leaf s",(4,5))],
              Branch [SubNode ("s",(1,4)),SubNode ("s",(4,4)),SubNode ("Leaf s",(4,5))]])
    2 (["s"],[])
      ((2,2),[Leaf "empty"])
      ((2,3),[Branch [SubNode ("s",(2,2)),SubNode ("s",(2,2)),SubNode ("Leaf s",(2,3))]])
      ((2,4),[Branch [SubNode ("s",(2,2)),SubNode ("s",(2,3)),SubNode ("Leaf s",(3,4))],
              Branch [SubNode ("s",(2,3)),SubNode ("s",(3,3)),SubNode ("Leaf s",(3,4))]])
      ((2,5),[Branch [SubNode ("s",(2,2)),SubNode ("s",(2,4)),SubNode ("Leaf s",(4,5))],
              Branch [SubNode ("s",(2,3)),SubNode ("s",(3,4)),SubNode ("Leaf s",(4,5))],
              Branch [SubNode ("s",(2,4)),SubNode ("s",(4,4)),SubNode ("Leaf s",(4,5))]])
    3 (["s"],[])
      ((3,3),[Leaf "empty"])
      ((3,4),[Branch [SubNode ("s",(3,3)),SubNode ("s",(3,3)),SubNode ("Leaf s",(3,4))]])
      ((3,5),[Branch [SubNode ("s",(3,3)),SubNode ("s",(3,4)),SubNode ("Leaf s",(4,5))],
              Branch [SubNode ("s",(3,4)),SubNode ("s",(4,4)),SubNode ("Leaf s",(4,5))]])
    4 (["s"],[])
      ((4,4),[Leaf "empty"])
      ((4,5),[Branch [SubNode ("s",(4,4)),SubNode ("s",(4,4)),SubNode ("Leaf s",(4,5))]])
    5 (["s"],[])
      ((5,5),[Leaf "empty"])
```

Example 4: Memoizing components of CFG

The following example is the equivalent ambiguous left-recursive version of the grammar from example 3, but we memoized sub-components of the grammar for improved performance. In this way any grammar can be represented in CNF.

Original CFG

```
S::= S S 's' | ε
```

Equivalent combinator-parser

```
s1 = memoize "s1" ((s1 `thenS` memoize "s_" (s1 `thenS` (term 's'))) `orelse` empty)
input = "ssss"

apply s at start-position 1 of input "ssss" with empty comtext and empty memo-table
⇒
"s1" 1 (["s1"],[])
       ((1,1),[Leaf "empty"])
       ((1,2),[Branch [SubNode ("s1",(1,1)),SubNode ("s_",(1,2))]])
       ((1,3),[Branch [SubNode ("s1",(1,1)),SubNode ("s_",(1,3))],
               Branch [Sube ("s1",(1,2)),SubNode ("s_",(2,3))]])
       ((1,4),[Branch [SubNode ("s1",(1,1)),SubNode ("s_",(1,4))],
               Branch [Sube ("s1",(1,2)),SubNode ("s_",(2,4))],
```

```
                Branch [SubNode ("s1",(1,3)),SubNode ("s (3,4))]])
        ((1,5),[Branch [SubNode ("s1",(1,1)),SubNode ("s_",(1,5))],
                Branch [Sube ("s1",(1,2)),SubNode ("s_",(2,5))],
                Branch [SubNode ("s1",(1,3)),SubNode ("s(3,5))],
                Branch [SubNode ("s1",(1,4)),SubNode ("s_",(4,5))]])
      2 (["s1"],[(2,[])])
        ((2,2),[Leaf "empty"])
        ((2,3),[Branch [SubNode ("s1",(2,2)),SubNode ("s_",(2,3))]])
        ((2,4),[Branch [SubNode ("s1",(2,2)),SubNode ("s_",(2,4))],
                Branch [SubNode ("s1",(2,3)),SubNode ("s_",(3,4))]])
        ((2,5),[Branch [SubNode ("s1",(2,2)),SubNode ("s_",(2,5))],
                Branch [SubNode ("s1",(2,3)),SubNode ("s_",(3,5))],
                Branch [SubNode ("s1",(2,4)),SubNode ("s_",(4,5))]])
      3 (["s1"],[(3,[])])
        ((3,3),[Leaf "empty"])
        ((3,4),[Branch [SubNode ("s1",(3,3)),SubNode ("s_",(3,4))]])
        ((3,5),[Branch [SubNode ("s1",(3,3)),SubNode ("s_",(3,5))],
                Branch [SubNode ("s1",(3,4)),SubNode ("s_",(4,5))]])
      4 (["s1"],[(4,[])])
        ((4,4),[Leaf "empty"])
        ((4,5),[Branch [SubNode ("s1",(4,4)),SubNode ("s_",(4,5))]])
      5 (["s1"],[(5,[])])
        ((5,5),[Leaf "empty"])
"s_" 1 (["s1"],[(1,[("s1",1)])])
        ((1,2),[Branch [SubNode ("s1",(1,1)),SubNode ("Leaf s",(1,2))]])
        ((1,3),[Branch [SubNode ("s1",(1,2)),SubNode ("Leaf s",(2,3))]])
        ((1,4),[Branch [SubNode ("s1",(1,3)),SubNode ("Leaf s",(3,4))]])
        ((1,5),[Branch [SubNode ("s1",(1,4)),SubNode ("Leaf s",(4,5))]])
      2 (["s1"],[])
        ((2,3),[Branch [SubNode ("s1",(2,2)),SubNode ("Leaf s",(2,3))]])
        ((2,4),[Branch [SubNode ("s1",(2,3)),SubNode ("Leaf s",(3,4))]])
        ((2,5),[Branch [SubNode ("s1",(2,4)),SubNode ("Leaf s",(4,5))]])
      3 (["s1"],[])
        ((3,4),[Branch [SubNode ("s1",(3,3)),SubNode ("Leaf s",(3,4))]])
        ((3,5),[Branch [SubNode ("s1",(3,4)),SubNode ("Leaf s",(4,5))]])
      4 (["s1"],[])
        ((4,5),[Branch [SubNode ("s1",(4,4)),SubNode ("Leaf s",(4,5))]])
      5 (["s1"],[])
```

Example 5: Direct CNF form of CFG

The following example is the equivalent ambiguous left-recursive Chomsky-Normal Form (CNF) version of the grammar from example 3 and 4.

Original CFG

```
S::= S A | ε
A::= S 's'
```

Equivalent combinator-parser

```
s    = memoize "s" ((s `thenS` a) `orelse` empty)
a    = memoize "a" (s `thenS` (term 's'))
input = "ssss"
```

```
apply s at start-position 1 of input "ssss" with empty comtext and empty memo-table
⇒
"s" 1 (["s"],[])
      ((1,1),[Leaf "empty"])
      ((1,2),[Branch [SubNode ("s",(1,1)),SubNode ("a",(1,2))]])
      ((1,3),[Branch [SubNode ("s",(1,1)),SubNode ("a",(1,3))],
              Branch [SubNode ("s",(1,2)),SubNode ("a",(2,3))]])
      ((1,4),[Branch [SubNode ("s",(1,1)),SubNode ("a",(1,4))],
              Branch [SubNode"s",(1,2)),SubNode ("a",(2,4))],
              Branch [SubNode ("s",(1,3)),SubNode ("a",(3,4))])
      ((1,5),[Branch [SubNode ("s",(1,1)),SubNode ("a",(1,5))],
              Branch [SubNode"s",(1,2)),SubNode ("a",(2,5))],
              Branch [SubNode ("s",(1,3)),SubNode ("a",(3,5)),
              Branch [SubNode ("s",(1,4)),SubNode ("a",(4,5))]])
    2 (["s"],[(2,[])])
      ((2,2),[Leaf "empty"])
```

```
      ((2,3),[Branch [SubNode ("s",(2,2)),SubNode ("a",(2,3))]])
      ((2,4),[Branch [SubNode ("s",(2,2)),SubNode ("a",(2,4))],
              Branch [SubNode ("s",(2,3)),SubNode ("a",(3,4))]])
      ((2,5),[Branch [SubNode ("s",(2,2)),SubNode ("a",(2,5))],
              Branch [SubNode ("s",(2,3)),SubNode ("a",(3,5))],
              Branch [SubNode ("s",(2,4)),SubNode ("a",(4,5))]])
    3 (["s"],[(3,[])])
      ((3,3),[Leaf "empty"])
      ((3,4),[Branch [SubNode ("s",(3,3)),SubNode ("a",(3,4))]])
      ((3,5),[Branch [SubNode ("s",(3,3)),SubNode ("a",(3,5))],
              Branch [SubNode ("s",(3,4)),SubNode ("a",(4,5))]])
    4 (["s"],[(4,[])])
      ((4,4),[Leaf "empty"])
      ((4,5),[Branch [SubNode ("s",(4,4)),SubNode ("a",(4,5))]])
    5 (["s"],[(5,[])])
      ((5,5),[Leaf "empty"])
"a" 1 (["s"],[(1,[("s",1)])])
      ((1,2),[Branch [SubNode ("s",(1,1)),SubNode ("Leaf s",(1,2))]])
      ((1,3),[Branch [SubNode ("s",(1,2)),SubNode ("Leaf s",(2,3))]])
      ((1,4),[Branch [SubNode ("s",(1,3)),SubNode ("Leaf s",(3,4))]])
      ((1,5),[Branch [SubNode ("s",(1,4)),SubNode ("Leaf s",(4,5))]])
    2 (["s"],[])
      ((2,3),[Branch [SubNode ("s",(2,2)),SubNode ("Leaf s",(2,3))]])
      ((2,4),[Branch [SubNode ("s",(2,3)),SubNode ("Leaf s",(3,4))]])
      ((2,5),[Branch [SubNode ("s",(2,4)),SubNode ("Leaf s",(4,5))]])
    3 (["s"],[])
      ((3,4),[Branch [SubNode ("s",(3,3)),SubNode ("Leaf s",(3,4))]])
      ((3,5),[Branch [SubNode ("s",(3,4)),SubNode ("Leaf s",(4,5))]])
    4 (["s"],[])
      ((4,5),[Branch [SubNode ("s",(4,4)),SubNode ("Leaf s",(4,5))]])
    5 (["s"],[])
```

Example 6: Cyclic Grammar

The following example is an application of the algorithm on a cyclic-CFG.

Original CFG

```
S1::= S1 'x' | P | 'x' | 'y'| Q
Q ::= R
R ::= P
P ::= S1 'y'
```

Equivalent combinator-parser

```
s1 = memoize "s1" ((s1 `thenS` (term 'x')) `orelse` p `orelse` (term
                   'x') `orelse` (term 'y') `orelse` q)
q  = memoize "q" r
r  = memoize "r" p
p  = memoize "p" (s1 `thenS` (term 'y'))

input = "yyyy"
apply s at start-position 1 of input "ssss" with empty comtext and empty memo-table
⇒
"p" 1 (["s1"],[(1,[("s1",1)])])
      ((1,3),[Branch [SubNode ("s1",(1,2)),SubNode ("Leaf y",(2,3))]])
      ((1,4),[Branch [SubNode ("s1",(1,3)),SubNode ("Leaf y",(3,4))]])
      ((1,5),[Branch [SubNode ("s1",(1,4)),SubNode ("Leaf y",(4,5))]])
"r" 1 (["s1"],[(1,[("s1",1)])])
      ((1,3),[SubNode ("p",(1,3))])
      ((1,4),[SubNode ("p",(1,4))])
      ((1,5),[SubNode ("p",(1,5))])
"q" 1 (["s1"],[(1,[("s1",1)])])
      ((1,3),[SubNode ("r",(1,3))])
      ((1,4),[SubNode ("r",(1,4))])
      ((1,5),[SubNode ("r",(1,5))])
"s1" 1 (["s1"],[])
       ((1,2),[Leaf "y"])
       ((1,3),[SubNode ("p",(1,3)),SubNode ("q",(1,3))])
       ((1,4),[SubNode ("p",(1,4)),SubNode ("q",(1,4))])
       ((1,5),[SubNode ("p",(1,5)),SubNode ("q",(1,5))])
```

REFERENCES

1. Aho, A.V and Ullman, J. D. (1972) The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing. Englewood Cliffs, N.J.: Prentice-Hall.
2. Burge, W. H. (1975) *Recursive Programming Techniques*. Addison-Wesley, Reading, Massachusetts.
3. Camarao, C., Figueiredo, L. and Oliveira, R.,H. (2003) Mimico: A Monadic Combinator Compiler Generator. *Journal of the Brazilian Computer Society.* Volume 9, Issue 1.
4. Chomsky, N. (1956) Three models for the description of language. *IRE Transactions on Information Theory, 2.* Pages: 113-124 .
5. Daume, H. (2004) Yet Another Haskell Tutorial. *http://www.isi.edu/~hdaume/htut/tutorial.pdf.2003-2004.*
6. Earley, J. (1970) An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, Volume 13, Issue 2. Pages:94-102. (52)
7. Frost, R. A. (1993) Guarded attribute grammars. *Software Practice and Experience.* Volume 23, Issue 10, Pages: 1139–1156.
8. Frost, R. A. (2002) W/AGE The Windsor Attribute Grammar Programming Environment. *IEEE Symposia on Human Centric Computing Languages and Environments.* Pages: 96-98.
9. Frost, R. (2003) Monadic Memoization towards Correctness-Preserving Reduction of Search. *Canadian Conference on AI 2003*, Pages: 66-80
10. Frost, R. A. (2006) Non-strict functional programming and natural language interfaces. *ACM Computing Surveys Journal* (in press).
11. Frost, R. and Hafiz, R. (2006) A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time. *SIGPLAN Notices* Volume 41 Issue 5, Pages: 46 - 54 *(May 2006 Article)*
12. Frost, R., Hafiz, R. and Callaghan, P (2006) A General Top-Down Parsing Algorithm, Accommodating Ambiguity and Left Recursion in Polynomial Time. *Tech Report 06-022*. Department of Computer Science, University of Windsor.
13. Frost, R. A. and Launchbury, E. J. (1989) Constructing natural language interpreters in a lazy functional language. *The Computer Journal — Special edition on Lazy Functional* Programming Volume 32, Issue 2. Pages: 108 – 121.
14. Frost, R. A. and Szydlowski, B. (1996) Memoizing purely functional top-down back tracking language processors. *Science of Computer Programming* Volume 27, Pages: 263–288.
15. Giesl, J. (1997) Termination of Nested and Mutually Recursive Algorithms *Journal of Automated Reasoning* Volume 19, Pages:1-29, 1997.
16. Hudak, P., Peterson, J. and Fasel, J. (2000) A gentle introduction to Haskell version 98. *Technical Report, Department of Computer Science*, Yale University.
17. Hughes, J. (1989) Why functional programming matters. *The Computer journal*, Volume 32 Issue 2, Pages: 98–107.
18. Hutton, G. (July 1992) Higher-order functions for parsing. *Journal of Functional Programming*, Volume 2 Issue 3, Pages: 323– 343.
19. Hutton, G. and Meijer, E. (1996) Monadic parser combinators. *Technical Report NOTTCS-TR-96-4*, Department of Computer Science, University of Nottingham.

20. Johnson, M. (1995) Squibs and Discussions: Memoization in top-down parsing. *Journal of Computational Linguistics*. Volume 21, Issue 3, Pages: 405–417.
21. Koopman, P. and Plasmeijer, R. (1999) Efficient combinator parsers. In Implementation of Functional Languages, LNCS, 1595:122 138. Springer-Verlag.
22. Koskimies, K. (1990) Lazy recursive descent parsing for modular language implementation. *Software Practice and Experience*, Volume 20 Issue 8, Pages: 749–772.
23. Kuno, S. (1965) The predictive analyzer and a path elimination technique. *Communications of the ACM*. Volume 8, Issue 7. Pages: 453 — 462.
24. Leermakers, R. (1993) The Functional Treatment of Parsing. *Kluwer Academic Publishers*, ISBN 0–7923–9376–7.
25. Leermakers, R., Augusteijn, L. and Aretz, F.E.J.K. (1992) A functional LR parser. *Theoretical Computer Science*. Volume 104 , Issue 2 , Pages: 313-323.
26. Leijen, D, and Meijer, E. (2001) Parsec: Direct style monadic parser combinators for the real world. *Technical Report UU-CS-2001-35*, Utrecht University
27. Lickman, P. (1995) Parsing With Fixed Points. *Master's Thesis*, University of Cambridge.
28. Moggi, E. Computational lambda-calculus and monads. In Symposium on Logic in Computer Science, Asilomar, California; IEEE, June 1989. (A longer version is available as a technical report from the University of Edinburgh.)
29. Moggi, E. An abstract view of programming languges. Course notes, University of Edinburgh.
30. Nederhof, M. J. and Koster, C. H. A. (1993) Top-Down Parsing for Left-recursive Grammars. *Research Institute for Declarative Systems*, Department of Informatics, Faculty of Mathematics and Informatics, Katholieke Universiteit, Nijmege, Technical Report 93–10.
31. Newbern, J. (2003) All About Monads. A comprehensive guide to the theory and practice of monadic programming in Haskell, Version 1.1.0.
32. Norvig, P. (1991) Techniques for automatic memoisation with applications to context-free parsing. Journal - *Computational Linguistics* Volume 17, Issue 1, Pages: 91 - 98.
33. Oosterhof, N., Hölzenspies, P., and Kuper, J. (2005) Application patterns. *A presentation at Trends in Functional Programming.*
34. Shiel, B. A. (1976) Observations on context-free parsing. *Center for Research in Computing Technology*, Aiken Computational Laboratory, Harvard University. Technical Report TR 12–76.
35. Tomita, M. (1985) Efficient Parsing for Natural Language. Kluwer, Boston, MA.
36. Wadler, P. (1985) How to replace failure by a list of successes, in P. Jouannaud (ed.) *Functional Programming Languages and Computer Architectures* Lecture Notes in Computer Science 201, Springer-Verlag, Heidelberg, 113, Pages: 113 - 128 .
37. Wadler, P. (1990) Comprehending monads. *ACM SIGPLAN/ SIGACT/SIGART Symposium on Lisp and Functional Programming*, Nice, France, June 1990, Pages: 61–78.
38. Wadler, P. (1993) Monads for functional programming. In M. Broy, (ed), *Program Design Calculi, volume 118 of NATO ASI Series F: Computer and System Sciences*, Springer-Verlag, Pages 233-264.

VITA AUCTORIS

Rahmatullah Hafiz was born in 1979 in Dhaka, Bangladesh. He graduated from Government Laboratory High School and Dhaka City College, Dhaka, Bangladesh in 1995 and 1997 respectively. He studied Applied Physics and Electronics at the University of Dhaka for two years before coming to the University of Windsor where he obtained B.Sc. (Honours) in Computer Science in 2004. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor.